# The Role and Impact of Software Coding Standards On System Integrity

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

**Coding standards are an integral part of today's safety-critical computer systems. Software verification and validation (V&V) practices significantly impact the cost of achieving human-rated levels of system integrity. The choices of software used to meet real-time, hard deadline requirements in onboard flight critical systems are relatively narrow. The stringent technical demands and expertise of the domain and the limited and specialized availability of industrial grade software products are the primary limiting factors. Robustness of software is influenced by choices in what programming languages, coding standards and testing, which is the primary form of V&V practice, are used. As a result, in these systems the state-of-the-art software uses less current programming features and techniques than those found, on the whole, in the software industry. Application of coding standards in high integrity systems software development is central to such practices. The goal of these is to make the software robust and safe and thereby contributing to overall system integrity. This paper examines the role and impact of interactions between use of the C++ programming language, the organization's own coding standards, and how these fit within V&V processes and procedures on robustness and testing as recently observed in NASA's EFT-1 project. These observations will help future flight software managers and engineers to make better application of coding standards throughout the V&V life cycle and optimize their impact on robustness and affordability.**

## I. Introduction

Coding standards grew out of the need to manage software from the earliest days of the invention of programming languages. Programmers found that it was essential to have common guidelines for understanding source code that was written by another. Although the nature of computer program source code is a distant cousin to that of literature as prose, both share the notion of style. Style found in source code represent aesthetics of human readability and comprehension based on a writer's or an engineer's taste and judgment.

The expressiveness of features of a programming language is found in its specification. Languages with few and simple features were less complex to use; however, the complexity of applications and their hosting computer environment have historically been drivers for the use of languages with more features. Such languages may satisfy one application domain but not another because the complexity of its features are found hard to understand nor safe nor predictable. This is often the case with hard real-time embedded systems. Current solutions in the use of programming languages such as C, C++ and others consist of a series of constraints that narrow the use of the features of a programming language. Figure 1 is an illustration with three funnels that distill down to the baseline-programming environment for the project's software developer.

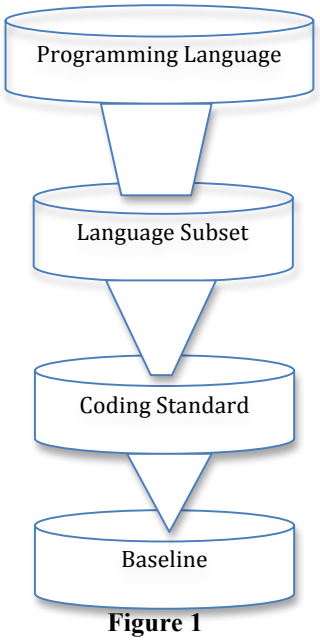


**Figure 1**

The first step is to start with the specification for the programming language. Depending on the needs of the domain a language subset may be defined. In practice this is done with significant investment on the part of a software vendor or the open source community. This subset specification is used to implement a compiler that only supports source code that is compliant with the narrowed specification. With this subset there may be a need for the creation of a coding standard. In practice the

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

compiler vendor's customer such as a flight software project usually does this. It may be asked why have the need for a coding standard. If the language or its subset were sufficiently matched to the application domain then, in principle, the coding standard would not be needed. In practice, however, it is needed because programmers must have project specific conventions that allow them to share one others' code and for the need for flexibility. As the life cycle of a project goes through its phases, experience may uncover additional rules to add to the standard based on lessons learned. The places where the project's software stakeholders can exert on their use of the compiler are in the rules of the coding standard. If it is not feasible to sufficiently cover software robustness and safety through the application of the coding standard then a remedy has to be sought with the vendor.

## II. Background

### A. Purpose and Scope

This paper covers the use of a coding standard in a NASA flight software project and provides a balance of introduction and detail of its specific rules; it is not meant to serve as a treatise. It also provides discussion of a tool used to automatically check source code compliance according to the standard.

The standard used by the flight software project serves as the definitive writing style for onboard source code. However, the standard encompasses more than just writing style. Within the scope of the flight critical project discussed below, design and testing are part of it. The rules covering design and testing, although only a small percentage of the total number, have ramifications to program and project level stakeholders across multiple organizations and disciplines within NASA and other government agencies and contractors with responsibility for high integrity systems.

The paper assumes that the background of the audience is similar to these stakeholders and includes individuals from disciplines of system engineering and integration (SE&I), and flight software managers and software developers. When working with onboard flight software, these stakeholders are tied together through the application of the coding standard. This is best seen in the context of software V&V lifecycle. The V&V lifecycle discussed in a section below, which is given in terms of a V-model, serves as a basis for discussion of the role and impact of coding standards.

One point of this paper is to make a significant case of how critical just one coding rule can impact the operational integrity of a mission. A historical case study is provided in terms of how stakeholders' requirements collectively led to a deviation in coding practice with unintended consequences in flight and ultimately to a mishap.

V&V process changes can modify the project's coding standard. This paper discusses how the impact of deviations is managed within the life cycle of a high integrity flight software project.

The paper discusses how design and testing rules may result in source code that is not compliant to the standard and how resolution of compliance deviations may need the consensus of stakeholders' to choose what trade-offs are appropriate.

### B. EFT1 Flight Software

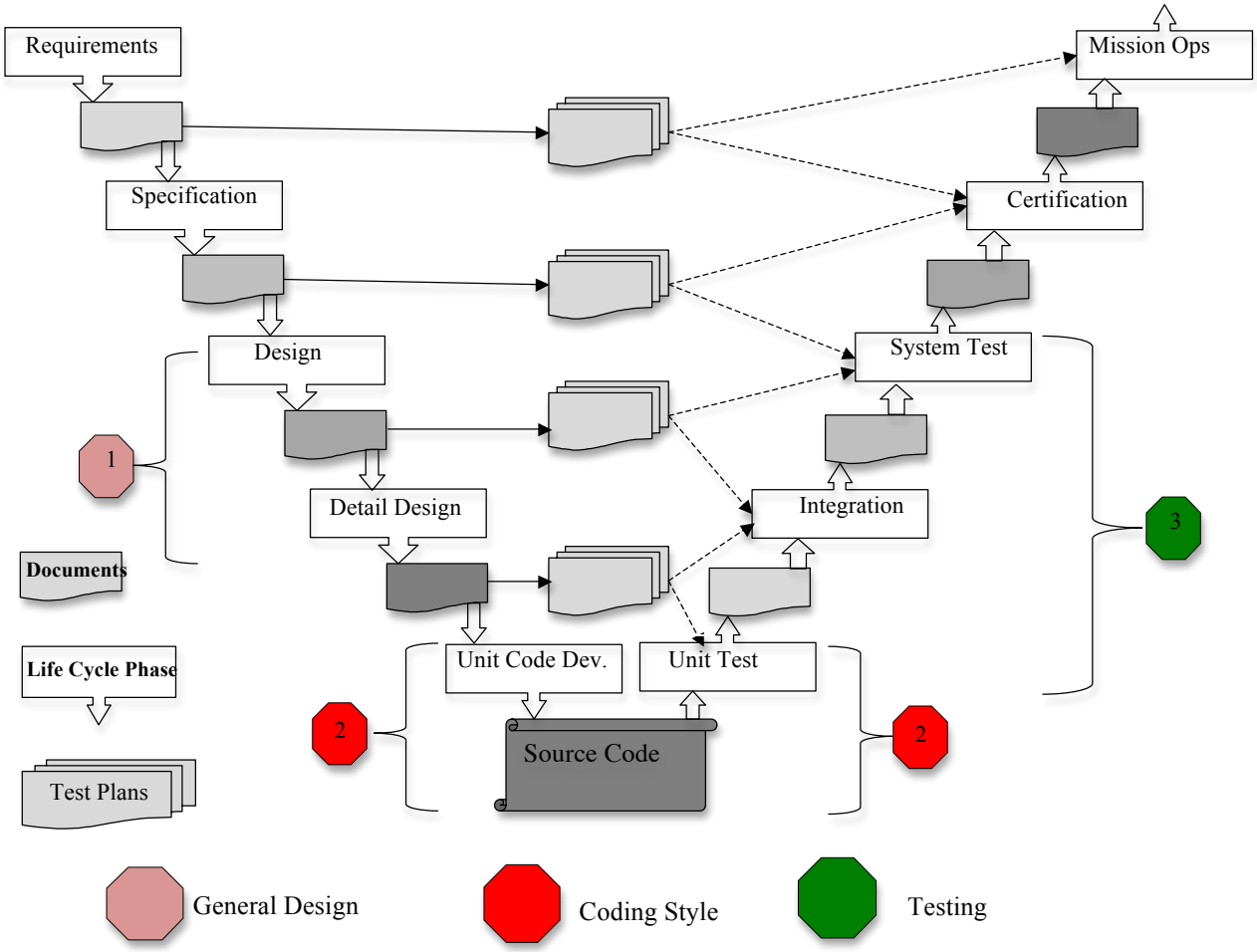
NASA's Exploration Test Flight-1 (EFT1)[5] is the first of a series of test flights that will be an in-house step in returning NASA to human space flight with the launch of the crewed Orion Multi-Purpose Crew Vehicle (MPCV) in the 2020s. The scope of EFT-1's mission will validate Orion's basic physical capabilities for orbital flight and a re-entry trajectory similar to those coming from beyond low earth orbit. The EFT1 avionics and onboard flight software consist of primarily core software that provides command and data handling functions and basic applications support for data recording and mission management. Its role is to be the "glue" for hardware testing in flight. The current flight software load image is approximately 500K source lines of code (SLOC) and is estimated to be around 50 percent of what eventually will be the footprint for a crewed MPCV.

Currently, the flight software is being developed as NASA 7150.2 class B software, which is mission critical. The ARINC 653 standard[6] is used as the overarching avionics architecture. Use of COTS software of DO-178B pedigree provides the real-time operating system, which is compliant to this standard. Additional software is provided under contract that extend the necessary functionality to support applications for space flight that have had prior flight certification for FAA air frames such as the Boeing 777. This software is being leveraged for use in NASA human space flight systems. Proprietary intellectual property and ITAR restrictions cover much of the flight software's life cycle and its content. This paper has been written to cover this material in a way suitable for public use.

### C. The Verification and Validation Model

EFT1 flight software is developed under the auspices of the Orion program's and flight software project's system life cycle. This cycle draws on a number of NASA project management and engineering guidelines[13] adopted and integrated with the prime contractor's own guidelines as well as the subcontractors'.

To understand the role of coding standards within the V&V software life cycle end-to-end, it helps to use a reference model. The diagram illustrated below is based on a "V" development lifecycle[7] for software, and referred here as the V-model.




**V-Model Figure 2**

This diagram emphasizes the testing of software, a common philosophy for embedded safety critical systems. This model is not one that is traced directly to processes and procedures following project documents; such models are meant to serve for notational purposes of philosophy. How or what model is followed in practice is driven by contract stipulations. Regardless of which V&V life cycle model is used the project has to meet contract deliverables in specific form and on schedule. These do the most to force a project to follow for example, a waterfall or a spiral or hybrid approach.

A broader view gained from experience finds that multiple software development processes have been used in different organizations on the way down the left side of the V-model. For example, some departmental level organization units have used Scrum[14], an agile software development approach, with special to Scrum methodology tool support. What activity and steps used down or up the V-model by the organizations' software practitioners are required to eventually link up with program level phases that set major project level activities and milestones. These linkages occur through a series of incremental software builds and software test plans that culminate in a final step of V&V sufficient to pass, for example, EFT1's Flight Test Readiness Review (FTRR).

Note that the traffic-stop like diagrams above in the V-Model will be found in following sections of the paper: 1) refer to General Design; 2) refers to Coding Standard; and 3) refers to Testing. Please also note the legend of diagrams on the left: **Documents**, **Life Cycle Phase**, and **Test Plans**. Also note that the shading of the **Documents**

diagrams refers to the potential increase of documentation update and maintenance. Going down on the left side the darker the shade of **Documents** represents substantial increases in investment effort that relies on those above. Changes at the **Source Code Component** for whatever reason may require resolution and updates to be made to higher level documents and, in turn, bring in those stakeholders responsible for these and their approval. Likewise on the right of the V-Model the shades of gray start out light and go darker as testing at high levels of integration represent more substantial investment and potentially more stakeholder participation.

## III. Motivation

Coding standards are important because mission operations of a high integrity system can be compromised. One omission of the application of a guideline in one line of source code may contribute to unintended consequences. Such an omission may place unintended behavior of software to be part of the chain of causality linking unintended system level interdependencies between hardware and software.

Systems integrity with mission and safety critical requirements for aviation and human space flight domains are intended to operate with no more than 1 in $10^{-9}$ failure occurring within operational life-times of airframe or human rated spacecraft. Embedded software in avionics subsystems needs to be built with enough confidence that margins of correct behavior are sufficient to contribute to overall system integrity[2]. Software coding standards and guidelines[2, 3] are leveraged in practice to attain desired levels of software's *robustness*, which is integral to system integrity. However, the design of the system may lessen its exposure to failing parts through use of fault containment mechanisms. In these circumstances hardware or software modules of sufficient redundancy are not required to perform to the same level of integrity as the system as a whole. However, it is not possible to practically demonstrate by testing the requirement of very long MTTF for a high integrity system even for redundant modules.

High integrity systems' certification is based on reliability models for hardware. Random processes of failure in physical phenomena found in hardware reliability models are not currently feasible to assign to software. Verification and validation (V&V) techniques used to attain software certification rely on testing measures of adequacy and coverage. In place of the measure of confidence given by a reliability model for hardware certification, software gains sufficient confidence through the application of standards such as DO-178B (and its recent update to DO-178C) or NPR7150.2 for certification. This difference in V&V approaches follows the fundamental dichotomy of a requirement being assigned as to hardware or as to software. A potential systems engineering oversight is to view avionics components as hardware with only random failures, which already have been covered by prior certification margins derived from physical phenomena, to the exclusion of embedded software. Such a systems engineering level oversight does not take in account of the possibility of a failure not caused by random event but by unintended behavior in the software. A case in point of this kind of oversight occurred with the Ariane 501 flight mishap.

The Ariane accident report[4] points to software as at the end of the chain of causality leading to mishap. On the maiden flight of Ariane 5 a 16-bit variable was given value assignments from a 64-bit variable larger than those used in previous software images in operational flights of the Ariane 4; these had not overflowed in previous missions. However, during flight 501 of the Ariane 5, upon detection of overflow, the hardware entered the Operand Error exception handler which, by design, resulted in a shutdown of the Inertial Reference System (SRI); the identical backup SRI also shutdown soon afterwards from the same overflow condition. The result with these avionics units off-line the On-Board Computer (OBC), which was Ariane 5's flight controller, reacted to SRI diagnostic data on the data-bus. The OBC continued to use this diagnostic data as if it were inertial data in calculations to command a flight trajectory and the result was to veer the vehicle to destruction.

Current coding standards practices identify such programming numerical overflow conditions. Ariane's software developers were well aware of the omission of checking for overflow, but other stakeholder's requirements overruled. One was to protect CPU usage margins; any nonessential computing overhead was scrubbed. The calculations of the flight trajectory of Ariane 4 were found to never cause overflow in specific lines of code. Therefore in these cases, it was credible to omit overflow checking. The reuse of Ariane 4's avionics components in Ariane 5 went through V&V as hardware without sufficient software testing in integration. End-to-end simulations lacked coverage of the new trajectory and the potential for unintended behavior on the part of the software. The lessons learned from Ariane 5 have been incorporated since then and therefore, it is less likely for a similar oversight to occur now because of requirements management tools that automate much of the requirements traceability matrix (RTM) found in large mission critical projects. This assumes that requirements such as found in a coding standard are included in the project's end-to-end life cycle RTM.

## IV. Programming Language—C++

The role and impact of coding standards cross cuts several topics. Coding standards may be viewed as necessary extensions of the programming language in which to implement requirements in robust software, which has additional properties, such as maintainability, portability, etc. On the other hand, the software development model and processes and procedures, which are independent of a programming language, are covered, often times, also included within a standard. This independence is not straightforward because the concepts embodied in a programming language link up with philosophy and its methodology of the life cycle.

### A. C++ Language Specification

The C++ language has been used in a wide variety of applications. The user community for the language is large and active. Implementations of the language's specification and new applications for it are also active; a recent paper discusses ramifications of updates in features; in robust use of code constructs; and in compiler code generation[15]. For the remainder of the paper the reader may find it helpful to have some familiarity with C++ or similar language.

The goal and use of C++ in high integrity systems is to minimize, among other matters, *ambiguity* in declaration and code usage, and behaviors that are *unspecified* or *undefined*[10]. A software developer writing robust C++ code for high integrity applications needs to be cognizant that the Standard C++ specification does not require a diagnostic or warning to be issued by a conforming compiler in certain valid uses of C++ code. Such instances of code use or structure may have semantic *ambiguity* and may result in either *unspecified* or *undefined* behavior. *Unspecified behavior* is at the discretion of the compiler writer, but must provide intended behavior. *Undefined* behavior is defined by the C++ language specification as an omission of explicit definition or that a program uses an erroneous construct or data. In the latter case, the result of this behavior may have unintended consequences; however, an erroneous construct or data instance should accompany with a compiler diagnostic, which is part of the language specification.

Note that many C++ coding standards and texts on C++ coding practices provide guidance about these possible pitfalls for software developers.

## V. Language Subset—Embedded C++ Language

### A. EC++ Language Specification

The Orion Flight Software project's starting point for an avionics-computing platform was a real-time operating system (RTOS) and subset of C++, called Embedded C++ (EC++), which are supported by Green Hills Systems (GHS). The specification for EC++[8] is used with GHS RTOS products by the project. The EC++ subset is said to be a strict subset. In principle, any code written in it will compile with any standard C++ compiler, although compilers native to EC++ are necessary for results to be of industrial strength. The features of C++ that are left out of EC++ are:

1) Multiple Inheritance
2) Virtual base classes
3) Run-time type information (typeid)
4) C++ style of type casting( static_cast, dynamic_cast, reinterpret_cast, and const_cast)
5) Mutable type qualifier
6) Namespaces
7) Exceptions
8) Templates

One of flight software project's initial tasks was to create a coding standard suitable for this subset. The approach taken was to adopt an existing C++ coding standard already in use in high integrity avionics systems.

## VI. Coding Standard

### A. The Joint Strike Fighter (JSF) AV Coding Standard

The de facto coding standard for C++ used in high integrity embedded systems is the Joint Strike Fighter (JSF) C++ coding standard[9]. The JSF coding standard does not specify which subsets of C++ are compliant with it. It is up to the user of the subset to scope their use of the standard for their target. The JSF coding standard has 229 rules, where the enumeration of the rules varies slightly depending on interpretation and version.

The standard is organized into three major headings: *General Design*; *C++ Coding Standards*; and *Testing*.

*1. General Design*

Aside of a brief discussion of cohesion and coupling design, this topic covers qualities of programming design in terms of *ilities:* reliability, portability, maintainability, testability, reusability, extensibility, and readability. Three coding rules provide metrics relevant to these *ilities*. One of the rules is: **AV Rule 1** *Any one function (or method) will contain no more than 200 logical source lines of code*. The others are discussed further in a subsequent section.

*2. C++ Coding Standards*

Other of the standard's more generic topics is covered under this heading instead of the previous one. This is emphasized here for the sake of clarification. They are as follows:

a) Rules. The rules are assigned with the designations of *should*, *will* and *shall* where *shall* is mandatory. Some of the rules include an exception clause that may be used depending on circumstances and which designation has been assigned. These may incur certain handling in processes and procedures down and up at different levels of the V-model.

b) Style. This subheading as well as other closely related subheadings, which cover environment, file management and others, is meant to aid human readability and comprehension of the source code. Out of 229 rules, style has 223 of them.

*3. Testing*

Three rules are given for testing. These support verification and validation in test adequacy and test coverage. One of the rules is discussed further in a subsequent section is about structural coverage. The others closely support it and deal with how the code is structured for sake of inspection by either manual or automatic checking.

Note: All C++ language features are allowed with the JSF standard except for Exceptions: use of C++ *catch* and *throw* constructs are strictly not allowed. The use of all other Standard conformance C++ features is subject to coding rule requirements.

## VII. Baseline

### A. The Orion Coding Standard

The Orion coding standard is based on a subset of the JSF's rules with specific modifications and additions. These were necessary for software robustness in safety critical applications as they are not alone sufficient to satisfy flight or mission worthiness certification. The JSF C++ standard references a JSF system safety documents, SEAL 1/2/3, which provides additional safety stipulations; NASA has processes to apply its own equivalent safety stipulations as necessary. This was done through formal project processes and procedures involving peer reviews, which included a number of stakeholders across the Agency, some of which were from Safety and Mission Assurance (SMA) and Software Quality Assurance (SQA).

Although there were differences in rules of the standards between Orion's and JSF's, which were substantial, the similarities between the two allowed for comparison of the output of a commercial static checker, which has had extensive industrial use with the JSF standard. This has provided additional confidence to the flight software project's software verification testing.

### B. Tool Support for Orion C++ Coding Standards

Software tools such as static checkers that automate many of today's rules of coding standards aid the verification and validation processes of source code compliance to it. To do without a tool would require manual inspection that would be a large use of personnel resources and would be prone to human error. However, any nontrivial coding standard such as JSF's or Orion's still required considerable human inspection to check compliance. The reason for this is that static checkers are not in the position to check an instance of source code the same way that a compiler does. This is best explained here in the use of the Liverpool Data Research Associates (LDRA) static checking product[12], which is under contract for use in the Orion V&V software tool chain.

According to LDRA's automatic checking of the JSF's rules, coverage of these fall into the following categories:

- 130 implemented
- 8 partially implemented
- 89 not implementable
- 2 not implemented but may be feasible

What it means for a rule to be implemented is it is possible to write a pass/fail test or a default metric test that is executed by the tool against the source code. If a piece of source code passes or is within nominal parameters then it is deemed as verified per that portion of the coding standard. The remaining not implemented 99 rules, including

those only partially implemented, require manual verification. According to NASA's terminology, the remaining forms of verification comprise of manual analysis, inspection or demonstration.

A number of reasons that drive why the rules are not implemented are as follows:

1) The rules do not apply to C++ coding but to matters such as project management level directives. For example, AV rules 4, 5, and 6, which is shown here—**AV 6** *Each deviation from a **"shall"** rule **shall** be documented in the file that contains the deviation). Deviations from this rule **shall not** be allow, AV Rule 5 Notwithstanding.*

2) The rules are more descriptive than prescriptive. The rules are specific to C++ coding but are insufficient to be testable. For example, **AV rule 15**: *Provision **shall** be made for run-time checking (defensive programming).* This rule provides descriptive guidelines about coding techniques and checks that depend on software practitioner's expertise. Another example is **AV 218:** *Compiler warning levels **will** be set in compliance with project policies.* This is handled by inspection and/or demonstration, which may technically be covered by a test script and is subject to software project management.

3) The remaining rules specific to C++ coding are currently impractical to implement because of affordability and/or theoretically intractable issues. Here are a couple of examples. **AV Rule 2** *There **shall not** be any self-modifying code.* Devising sufficient coverage of instances of such code is problematic. There may be ways to test some cases for this rule, but it is questionable how effective the detection would be and at what cost. The current cost effective and practical approach to detect possibilities of such code is through inspection through peer code walk-throughs. As mentioned earlier, a community of software professionals sharing one another's code is essential today as well as it was in the past and is critical in gaining confidence of software robustness. For this code to pass muster formal peer or technical code reviews are the means that the code satisfies *all* the rules adequately, including those without an automated test.

The second example illustrates aspects of the C++ language specification. **AV Rule 71.1** *The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure. Rationale: Compilers are not required to diagnose this ambiguity.* According to LDRA this rule is one that is not currently checkable, but may be feasible. It may be more of a matter of practical affordability, but then it may be part of cases of code that are "legal" or permissible to compile, but whose behavior at execution has unintended consequences.

There are a number of other static checking tools for C++ that specialize in support of *safe* use of C++ in high integrity systems. This paper has limited itself to one tool and one set of coding rules for the sake of scope, brevity and the fact that the Orion tool chain was base-lined for LDRA.

### C. Baseline Processes and Procedures

It is worth noting that software certification per DO-178C or NPR7150.2 do not favor any specific coding standard (or specific programming language). However, the incorporation of a coding standard in the V-model has ramifications on its processes and procedures. As a project progresses down the left side of the V-model it invests in *tailoring* the use of the tool with the ideal that it is sufficient for multiple V&V cycles through the V-model, which occurs with a spiral model. Once the configuration of the tool is set it means its reports are fixed. In practice, when there is a deviation or discrepancy found in the tool's reports then the code is updated appropriately to remedy the discrepancy, but when it is found otherwise larger considerations are invoked. For example, when high-level specification tools such as those based on UML automatically generate C++ source code, the output may not conform to the coding standard. The choice of whether to have the UML tool vendor to modify their product or to modify the coding standard's rules is up to the project's stakeholders.

An alternative to *tailoring* the coding standard is to issue a *waiver*, which is like a one time exception to a rule. The impact on cost and schedule of requesting a *waiver* requires additional peer reviews and likely the approval of a software control board. Requesting a *tailoring* after the initial baseline is greater that what it takes to seek a *waiver* and is not to be taken lightly because it involves more levels of review and approval. It should not just be taken lightly because of the impacts on personnel costs and project schedule, but because of the implications to deviating from the coding standard. Rules of style that cover aesthetics such formatting, naming and other conventions based on human judgment may be viewed as relatively superficial, and rightly so, but, an extreme view of coding standards is that for high integrity systems they are virtually part of the programming language specification itself.

Although the paper focuses on one coding standard and a static checker, much of the previous discussion applies equally to other coding standards and vendors' tools. A question may be asked about the multiple uses of static checkers in a project. There can be only one coding standard and so whatever tool is used it must be configurable to cover the standard. Multiple use of static checkers require some additional configuration management and resolution in what results are officially used as part of verification credit in the V-model. At the very bottom of the V and on the left side, software practitioners may informally use any tools that help them prepare

for development test. But when verification of the code moves up the right side of the V-model through unit test to integrated testing and so on the project's software tool chain's reports are part of the basis of official verification credit. If more than one static checker product is used then results have to be correlated and resolved per additional processes and procedures. Multiple static checkers have been used informally within the Orion project, but not part of the baseline used for credit. Access to the use of the baseline static checker to all developers and to required third party testing organizations, as soon as possible, whose participation is mandated by high integrity standards, would have significant impact in reducing software project testing costs as all parties would be, so to speak, on the same page.

### D. Coding Standard Rules Trade-offs

Two coding rules have required additional consideration in interpretation in use and application of results. The first one to be discussed is Cyclomatic Complexity and the second is Modified Condition/ Decision Coverage (MC/DC). The first one falls under the heading of General Design and the other is under Testing. These two rules are out of six total under these headings, which augment the C++ language specification through additional constraints found in the standard.

*1. Cyclomatic Complexity*

***AV Rule 3*** *All functions shall have a cyclomatic complexity number of 20 or less*. This rule is software metrics to aid programmers to more readily read and comprehend the control flow in a software module and to reduce the number of tests. The number of 20 has been derived over time from studies of code samples from different programming languages. Subsequently, it has been adopted somewhat arbitrarily as the default applied to C++ in high integrity applications because little, if any, in the open literature on C++ was available in the formulation of Orion's own standard.

In practice in the development of some of the on-board mission planning and event software, there were several software components found to have greater than the index of 20. For one component, in particular, it was found to be "naturally complex". The design had to account for all practical combinations of a number of data type comparisons for a number of users/definers for a number of various mission events. Upon attempting a better way to express the necessary event logic so that it would satisfy the cyclomatic complexity index, it was found that breaking up the module would add little value to improving the design and more importantly, reducing the number of tests necessary. It would just spread the tests around to multiple components. A major refactoring of the module would have to be undertaken to meet the rule and to benefit in reducing the number of tests. The trade-off was to seek a *waiver* in the near term and, in the long term, to re-consider whether a refinement to this coding standard could be reformulated to fit such special application domains and eventually find its way into the standard via *tailoring*.

*2. Modified Condition/ Decision Coverage (MC/DC)*

**AV Rule 220** *Structural coverage algorithms shall be applied against flattened classes*. This rule requires that test coverage of the code take in account all instances of functions (class methods) possible in an inheritance hierarchy. Although MC/DC is not explicitly stated as part of the JSF Coding Standard it is included as guidance in Orion's V&V baseline and will be required or equivalent for subsequent human rated flights. There has been some confusion about the meaning of MC/DC[11]. The distinction between structural coverage analysis and structural coverage testing helps towards clarification and a recent update DO-178C, which provides additional guidance. However, caution is advised in relying on a vendor's product that supports MC/DC because different coverage analysis products use different instrumentation schemes as well as other factors.

In practice in the domain of math utilities, for example, in processing the Direction Cosine matrix for multiple eigenvalues with near singular or ill-conditioned inputs it was found that the function logic was not compliant to MC/DC. Upon review of the code, the trade-off between changing the code to be compliant versus seeking a waiver was problematic. The results of the tool's output appeared to be indeterminate as to whether code was not compliant or that it hit its own limitations of coverage. At this time, refactoring the code had risks in schedule and in effort as it was not straightforward to refactor the code and know that the result would be MC/DC compliant and also meet CPU and memory footprint constraints.

Other than clarification of terminology of what a function is within the context of C++, these rules are independent of the language specification. In the case of cyclomatic complexity the complexity found in the code may reflect the complexity of the requirements and the system level design. Whether this is mirrored between the two to some extent as "natively" intact or whether the software is refactored requires a tradeoff to be made based on stakeholders' discretion and engineering judgment. In the case of MC/DC the rationale for such structural analysis is to have clarity in sound logic in conditions and decisions formulated at the systems level as they are traced down to the source code. In this case, questions of compliance to MC/DC may be resolved through changes in the source text

or to seek clarification of the logic coming down from higher up specifications in system architecture and requirements.

### E. Challenges to Coding Standards Robustness

Two rules to be discussed cover some of the fundamental aspects of software robustness. These face some of the greatest challenges that are found in the use of C++ and other similar languages, and in their implementation and run-time environment support.

*1. Exception Handling*

**AV Rule 208** *C++ exception shall not be used (i.e. throw, catch and try shall not be used.)* This rule excludes the use of C++ features for handling arithmetic errors such as overflow; divide by zero, and array out of bounds and other indications. Although state-of-the-art C++ compiler technology has significantly improved support for these features, it is taking time for the high integrity embedded systems community to adopt these. The reasons are because exception-handling implementation in hard deadline real-time systems are found to be complex, and also, programmers' use of C++ exception handling features in applications have been historically deemed to have design complexity. It may be agreed that without use of these features testing less complex designs may be easier since they are free of C++ exception features. However, this alternative is not ideal in complex avionics applications. For the EFT1 flight software project, the GHS RTOS implements the ARINC 653 features that support application exceptions and error handling. These are covered under the ARINC 653 Health Monitoring *application error* type. This hardware replaces completely the use of C++ exception, but this hardware solution has some limitations.

Currently, the hardware is the backbone for the system's integrity as features in the RTOS's board support package cover all software anomalies. Unintended consequences in software are, by implementation, are covered by the hardware, but recovery time may impact the availability and hence the integrity of the system. The time to field exceptions from start to return back to recovery and the rate at which the system can sustain these are critical performance parameters for high integrity systems, which may result in diminished real-time and throughput capability and flexibility. In addition, the detail of information about the type of anomalous behavior, which is passed to the hardware as an event of *application error*, may often be mapped into ready-made fixed categories and, in such cases when software diagnostic information does not fit, some of it is discarded.

*2. Defensive Programming*

**AV Rule 15** *Provision shall be made for run-time checking (defensive programming).* This rule could be under the heading of *General Design* as well although it is found under *C++ Coding Standards Environment*, which covers C++ language specification features. Mitigation of failure modes of hardware faults and software errors through design are owned by this rule. The rule requires that safety hazard analysis be applied in the development of the software where a component or function is identified to be in the possible (and credible) chain of causality leading to loss of mission or crew. Hazard analysis may require programmers to design and implement specific additional code to existing components or functions that defensively mitigates their possible behavior in unusual or hazardous circumstances.

On the other hand, more than just covering design, this rule covers the explicit use of C++ features that fall under the rubric of *defensive programming* in general. Some explicit C++ coding risks to mitigate are: arithmetic errors, pointer arithmetic usage, array memory usage, range errors and other "hazardous" uses of the language's features, which in nonhazardous circumstances would not nominally be necessary without the specific context of the mission.

The challenge here is to provide additional sub-rules to this one that can be more prescriptive and therefore, may be automatically verified by a static checker tool. This would help cognizant subject matter experts whose background covers system level safety and hazard analysis and, at the same time, covers the safe and robust properties specific to defensive programming.

## VI Conclusion

The use of today's coding standards such as JSF AV or Orion's may have prevented the Ariane 501 flight mishap had the following rule been enforced: *The rule **AV Rule 203** Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented).* Much in the way of progress has occurred in the past sixteen years in software technology and its infrastructure of tool chain support for the V-model's verification processes and procedures.

The actual story of a software flaw as the root cause of Ariane and other similar accidents traceable to software are often far greater than a flaw in the software. In the case of Ariane the decision making process was founded on the prevailing "cultural" view of project engineering that random events in avionics modules can't be predicted or

can't be credibly mitigated. Application of this view led to rationalization that the possibility of encountering a numerical overflow was not credible or random. This view was reinforced in practice because the Inertial Reference System (SRI) was redundant. Simulation testing of these prior to use in Ariane 5 did not account for launch operations variations and changes to flight trajectory in complete systems-wide coverage. If the root cause had been a hardware fastener the effect would have been not different. For either a hardware or software flaw these may occur at the following points in the life cycle: 1) flaw in the design; 2) flaw in manufacture; 3) flaw in the application of the part as placed into the system in test; or 4) flaw in use as unintended in an environment which causes the part to be used outside of specification.

Many software "bugs" may be traced down through the V-model to a cascade of derived finer or detailed requirements or design specifications that contain the parent's inherent flaw that do not match the mission profile or are used inadvertently as unintended. Of the three main headings in the JSF coding standard, *General Design* and *Testing* most address 1) flaw in the design and 3) flaw in use as unintended. The third heading, *Coding Standards,* covers 2) manufacture where actual implementation of code rests on the substrata of language and environment specification. The flaw as encountered in the Ariane 5 was not one of manufacture or a cause traced below within the implementation of the substrata or in the language specification itself.

This observation refers back to the earlier mention of dichotomy of requirements allocation between hardware versus software. The point added here is that coding standards can do best to clearly impact software robustness in high integrity systems by focusing, perhaps exclusively, to manufacture. This means that the primary, if not, the sole role of the coding standard would be treated as if it were a part of the compiler. The certification of software for a project in a high integrity system rests on a certified compiler product and on its own in-house use of its coding standards. It is conceivable that a lapse in coding standards could be tantamount to a conceivable defect in the compiler. The root cause of Ariane 5 flight 501 may be viewed from today's perspective as not following the coding standard. If the rule had been actually part of the compiler then the source code would have never had compiled and therefore it would never have been a possibility for stakeholders' discretion come into play. Whether such a rule or others found in the coding standard have the same impact that a compiler does when it finds the code is not compilable is a question of what kind of servant is software suppose to be? The answer depends whether baseline software should be a pliant servant or not one that guards robustness and integrity.

Currently, the coding standard, as it is organized and used, covers the whole V-model. This monolithic document serves nearly all levels. An overarching recommendation is to separate the sections on *General Design* and *Testing* into their own documents and allow them to be expanded. Some of the rules found in the section on *C++ Coding Standards* may get extended treatment if they were moved to either of the other two. For example, defensive programming is a *shall* requirement, but has no actionable rules as it is not prescriptive; it should belong under *General Design*.

A recommendation to the section, *C++ Coding Standards*, is to include the use of C++ exception handling. The embedded software community as a whole and compiler technologists, together, are making rapid progress in use of the expressiveness of the C++ language to be suitable for high integrity systems. Currently, high integrity systems will, when encountering anomalous behavior, readily go to reset or will simply fail the box. Module resets impact the availability of a system and sensitivity to transitory behavior may result in module shut down. Both result in diminished mission capabilities or higher overhead in keeping redundant spares. The use of exception handling for diagnostics for anomalous behavior at the source of detection cuts out the round trip time for recovery of many anomalies now handled at higher and remote levels of the system. Finally, programmer defined exception handling routines not used currently by design may now be incorporated to add flexibility in recovery for functional robustness instead of resorting to reset or shutdown.

## VII. Summary

Coding standards originally started when source code conventions aided human readability and comprehension. Since then its role has substantially grown. For high integrity systems in general and for NASA's EFT1 project in specific the role for a coding standard, such as the JSF AV C++ guideline, is indispensible for attaining flight software certification. This paper discussed a life cycle of verification and validation in terms of the V-model, which, for much of the safety-critical computer systems community, is a guiding methodology.

The impact of the coding standard on the V-model is found throughout. It impacts the activities at the levels of design, down through to coding and upwards through the phases of testing. Its strength is in its comprehensive nature of tying together general design, language specifications usage, in the case C++, and testing.

Static checker tools do much of the compliance checking of source code to a standard. The product LDRA was discussed in terms of its coverage of the JSF AV standard. Approximately forty three percent of these were not

checkable. These fell into categories where some were software project guidance directives; some were of prescriptive guidance in C++ programming; and the remaining rules were impractical to implement because of apparent limitations in affordability and/or found to be intractable in theory. These rules' compliance can only be done by manual review where expertise in C++ language specifications is required.

Trade-offs in the use of rules covering cyclomatic complexity and modified condition/decision coverage (MC/DC) were discussed in terms of the V-model to tailor a rule or to waiver an exception to a rule. A challenge to robustness native to the C++ language in application source code are found because exception handling features native to the programming language had been disallowed, but moved to the hardware layer at a loss of capability and performance. The other challenge was that the defensive programming rule was limited to a narrative guidance without any actionable checks.

## Acknowledgments

The following colleagues helped significantly with observations and suggestions from their reviews: Rick Alena, Joe Coughlan, Steve Jacklin, Chris Knight, and Peter Robinson.

## References

[1]RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," December 1, 1992, URL: www.rtca.org

[2]IEEE 829, Standard for Software and System Test Documentation, URL: http://standards.ieee.org/findstds/standard/829-2008.html

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

[4] Lions, J.L., Ariane 5 Flight 501 Failure Report, http://www.di.unito.it/~damiani/ariane5rep.html

[5] Orion Exploration Flight Test-1, http://www.nasa.gov/pdf/663703main_flighttest1_fs_051812.pdf

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

[7] Storey, N., *Safety-Critical Computer Systems*, Addison Wesley, 1996, pp. 314.

[8] Embedded C++ Homepage Official Website, http://www.caravan.net/ec2plus/

[9] Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program, Document Number 2RDU00001 Rev C, December 2005, www.stroustrup.com/JSF-AV-rules.pdf

[10] ISO/IEC 14882-2011(E) Information technology—Programming languages—C++, 3rd edition, 2011-09-01

[11] Hayhurst,K.J., Veerhusen, D.S., Chilenski, J.,J., Rierson,L.k., NASA/TM-2001-210876, A Practical Tutorial on Modified Condition/Decision Coverage, May 2001, http://www.sti.nasa.gov

[12] LDRA Software Technology, http://www.ldra.com/

[13] NASA Systems Engineering Handbook NASA-SP-2007-6105 Rev1, URL: http://www.acq.osd.mil/se/docs/NASA-SP-2007-6105-Rev-1-Final-31Dec2007.pdf

[14] URL: www.scrum.org

[15] Stroustrup, B., "Software Development for Infrastructure," Computer", IEEE Computer Society Press, Vol. 45, Jan, pp. 47, 58