2011 Fifth International Conference on Secure Software Integration and Reliability Improvement

Safe Software: Does it Cost More to Develop?

W. Eric Wong, Andrea Demel, Vidroha Debroy
Department of Computer Science
University of Texas at Dallas, Richardson, Texas
{ewong, axd073000, vxd024000}@utdallas.edu

Michael F. Siok
Lockheed Martin Aeronautics Company
Fort Worth, Texas
mike.f.siok@lmco.com

Abstract

The importance of system safety has intensified in recent years given the ever-growing use of safety-critical systems in avionics, medicine, nuclear energy, and other fields. However, despite the abundance of standards which exist to provide guidance for the development of safe software for safety-critical systems, there is no consensus on how to achieve safety assurance in a cost-effective fashion. This paper reviews five software safety standards: the FAA System Safety Handbook, the US DoD MIL-STD-882D, the UK MoD DEF-STAN 00-56, NASA-STD 8719.13b and the RTCA DO-178B; and evaluates each in terms of cost effectiveness. It provides an overview of several safety-critical projects; ones that have incurred significant cost overruns as well as ones that have produced safety-critical software in a reasonably cost-effective manner. By virtue of discussing such projects we posit that it is possible to develop software, despite significant safety assurance requirements, without necessarily sacrificing cost. Specifically, projects can realize savings by using mature processes and appropriate tools to assist in development of safety-critical software.

Keywords: safety-critical software, safety standard, software safety, system safety, cost effectiveness

1. Introduction

The past several decades have seen a rapid increase in the use of software in safety-critical systems - systems used in the avionics, medical, nuclear, transportation, and military industries [13]. Software has vastly extended the capabilities and flexibility of these systems, but it has come at some cost - added complexity and difficulty in managing its development. Safe software is that software that does not contribute to operational system hazards [31].

In [48], it was shown that software has played an increasingly significant role in accidents and mishaps which have resulted in the loss of life, property, and money. For example, the National Aeronautics and Space Association (NASA) Mars Climate Orbiter (MCO), launched December 11, 1998, was assumed lost while in operation around Mars. The spacecraft was designed to enter an orbit of approximately 140-150 kilometers above the surface of Mars; it may have reached an altitude as low as 57 kilometers. While the exact cause of MCO's demise may never be verified, the MCO accident investigation board reasoned that the orbiter may have been lost due to a programming error in software; instead of metric units (newtons), imperial units (pound-seconds) were incorrectly used in the navigation module. Thus, the computer incorrectly estimated the MCO's thruster forces causing the spacecraft to veer off its intended path and assume an incorrect trajectory into the Martian atmosphere. The spacecraft was assumed destroyed by atmospheric stresses at low altitude. This programming error and subsequent loss of the aircraft cost NASA approximately $85 million in spacecraft development and mission operations costs [48].

Another example of a significant system loss is that of the European Space Agency's (ESA) Ariane 5 heavy lift launch system which was designed as an expendable payload-transfer system for low Earth orbit transfers. Shortly after liftoff June 4, 1996, the first Ariane 5 deviated from its intended flight path and exploded, causing the destruction of both the rocket and its cargo. Similarly to the MCO, the problem was traced to a relatively simple software error. In the case of the Ariane 5, the software attempted to convert a 64-bit floating point number to a 16-bit signed integer. However, the conversion failed because the value being converted was larger than 32,767 and thus outside the range that could be represented by a 16-bit signed integer; this caused an overflow error that was not handled. As a result, both the active and backup computers shut down and navigation control of the rocket was lost. The rocket self-destructed, per design. This loss cost the European Space Agency $500 million in rocket and cargo losses as well as unrealized development costs estimated at $7 billion [48]. The program has since performed successfully with relatively few other issues [4].

While these programming errors are relatively easy to understand, their realization in developed systems have caused extreme losses for their organizations. In these cases, software was identified as a contributor to the system realizing a specific hazard that ultimately resulted in the occurrence of an accident. Many other software errors in these and other industries have also led to catastrophic and significant losses [48]. A lack of software safety assurance in some instances may contribute to the raising of system hazards. Software can contribute to system hazards through inappropriate actions, either automated or operator-assisted, while operating a system. Software can also contribute to hazards by misleading system operators through its output or behavior (i.e., providing Hazardous Misleading Information (HMI)). Flaws or limitations in the software design are often extremely costly both in the actions required to fix the problems in the code and in the potential consequences of its aberrant behavior if the problem makes it to the field. As a result, industries with a

978-0-7695-4453-3/11 $25.00 © 2011 IEEE
DOI 10.1109/SSIRI.2011.28

high reliance on safety-critical systems have increased their focus on the design and development of safe software through an effective software safety assurance practice involving the use of engineering and managerial processes.

The effective management of safety programs for software is far from trivial. Some software safety assurance activities can significantly extend development time and cost of a project. Industry thus faces the challenge of how to reliably develop safe software at an affordable price; it is a value proposition. The decision of how much work is necessary often comes down to the need for a level of safety (i.e., the likely consequences of certain hazard realizations) versus the cost of the safety activities needed to mitigate the selected hazards. Different projects have different risk profiles; these risk profiles help determine the resources needed for assuring the safety of the system.

Due to the complexity and range of assurance activities available to apply to the development of safety-critical systems, there is a general need for industry guidance on what software assurance activities are expected. Several standards exist to provide guidance on system and software safety assurance. In this paper, five standards are discussed: the United States (US) Federal Administration Aviation (FAA) System Safety Handbook [14], the Department of Defense (DoD) MIL-STD-882D [11], the United Kingdom (UK) Ministry of Defense (MOD) DEF-STAN 00-56 [9], NASA STD 8719.13B [37], and the Radio Technical Commission for Aeronautics (RTCA) DO-178B [41]. These standards provide requirements and recommendations for ensuring system safety often in the context of a specific industry or type of system.

In this paper, we discuss the use of these standards with respect to software and we discuss their use and limitations in the context of several large-scale safety-critical projects. The remainder of this paper is organized as follows. In Section 2, we present the background of each of these five safety standards. Section 3 provides a discussion of projects that have faced relatively high costs in the development of their safety-critical software systems, followed by Section 4 which discusses projects that have successfully managed safety assurance processes in an apparent cost-effective manner. Finally, we present our conclusions on the cost of software safety in Section 5.

2. Software Safety Standards

Each of the five standards discussed in this paper provides guidance for safety assurance activities. However, their approaches to safety engineering vary widely. We discuss their background and differing approaches to, in particular, software safety in the sections that follow.

FAA System Safety Handbook

The FAA System Safety Handbook provides recommendations for implementing safety risk management activities for entities involved with the FAA. The handbook provides examples and specific guidelines for what is expected in a comprehensive system safety program. It is not a standard. This handbook is designed to aid the development and improvement of system safety programs in organizations and projects. To this end, the FAA System Safety Handbook provides guidance and recommendations for processes to improve the safety assurance of a product being developed for use in civil aviation systems. The handbook provides examples of documentation that should be developed and includes guidelines as to what activities should be included in a comprehensive system safety program.

The FAA System Safety Handbook discusses the importance of the safety benefit versus cost tradeoff. The handbook acknowledges that the safety effort is directly related to the cost of the safety program and provides a notional illustration of this relationship as shown in Figure 1. The handbook indicates that the level of safety effort needed for the project per the hazard analyses conducted takes into account the estimated cost of potential accidents as well as the estimated costs of developing safety programs. (See the discussion on DO-178B regarding safety levels.) Similarly, the costs of developing safety programs involve personnel with the required skill set, available resources, and development time. As shown in Figure 1, when the safety effort increases, so too does the total cost of development once the total program cost reaches its minima. Clearly, the cost of accidents and the cost of developing safety programs have an inverse relationship; they sum to the total cost of the project which also includes the costs for the design, development, and operational phases of the system. The label 'X' in the figure indicates the ideal safety effort versus cost balance in which the total program cost including the safety effort is minimized.

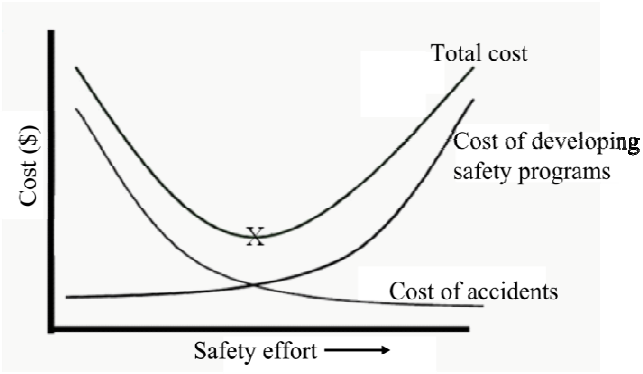

Figure 1: Safety effort versus cost analysis [14]

Taking into account these cost concerns, the FAA System Safety Handbook notes that in some cases a smaller system safety program may be tailored to reduce costs if the safety assurance requirements for the system support it. To this end, the Handbook recommends a minimum set of tasks for small system safety programs which includes preparing a *preliminary hazard list*, conducting a *preliminary hazard analysis*, and assigning a *risk assessment code* [14].

The Handbook also recommends design and verification parameters for each level of hazard risk in the system. Assessment of risk is made by combining the severity of consequence with the probability of occurrence of each hazard in a matrix and determining the acceptance criteria for the risk based on its assessed criticality. According to the Handbook, high-risk hazards are unacceptable and tracking in the FAA Hazard Tracking System is required until the risk is reduced and accepted. For medium-risk hazards, the hazard is acceptable with review by the appropriate management authority and tracking in the FAA Hazard Tracking System is required until the risk is accepted. Low-risk hazards are acceptable without review; the Handbook does not require these low risk hazards be tracked. [14]

The FAA Handbook provides a Comparative Safety Assessment process for measuring and determining safety benefit versus cost. This assessment is provided as an aid for decision-making when choosing an appropriate safety design from specified design alternatives. The handbook requires the assessment be used to examine the costs and safety risks associated with each design alternative under consideration. It requires that hazards associated with each design alternative proposed be listed and written so that decision-makers can clearly distinguish the relative safety merits of each alternative. These alternatives are then rank-ordered by how they best satisfy the safety requirements of the project. In this way, the most appropriate alternative may be selected.

Unfortunately, data is not available on the amount of cost use of the FAA System Safety Handbook contributes to an aviation project. However, its discussion of the cost versus safety benefit indicates that the FAA acknowledges such tradeoffs exist and that these trade-offs should be taken into account when applying a system safety protocol to the project.

MIL-STD-882D

MIL-STD-882D is a system safety standard produced for the DoD under the authority and oversight of the system safety committee of the Government Electronic and Information Technology Association (GEIA). This standard describes the requirements for system safety procedures and practices applicable to the development, test, production, use, and disposal of DoD systems, subsystems, equipment, and facilities. The standard is recommended for use by all departments and agencies within the DoD. [11]

MIL-STD-882D provides a full life-cycle approach to system safety in which hardware and software safety tasks are addressed in combination as a system. The standard is a goal-based, performance-oriented standard which aims to ensure that zero mishaps occur in DoD systems. To this end, the standard focuses on the identification, analysis, and mitigation of mishap risks. Additionally, the standard requires that a system safety approach be defined and thoroughly documented by the system developer and it requires thorough traceability of requirements throughout the development process. Furthermore, the standard provides extensive guidance on the use of mishap severity levels and requirements analysis.

However, this system safety standard is also relatively brief (i.e., compared to the other safety standards reviewed). As it is a standard designed to address overall system safety rather than software safety in particular, it also lacks information in some software-specific areas such as partitioning the software into safety-critical and non-safety components so that potential faults in non-safety-critical software sections do not affect the operation of software in the safety-critical partition.

MIL-STD-882D discusses the issue of safety benefit versus cost. It stresses the importance of considering total life cycle cost in any decision and ensuring that neither inadequate nor overly restrictive safety requirements be used. However, this standard provides little detail on how this cost versus safety benefit tradeoff is to be specifically implemented and analyzed. It acknowledges the existence of the tradeoff, but it does not devote many resources to describing its nature nor does it provide specific tools or processes for navigating the details of this issue.

Although MIL-STD-882D is not required for project certification by any governing body, it is still recommended for use in military aviation and US DoD projects. However, data is not available on the specific costs associated with using this standard.

DEF-STAN 00-56

DEF-STAN-00-56 is a safety standard that describes the requirements for safety management of UK MoD defense-related systems. This standard specifies safety management procedures, analysis techniques, and safety verification techniques intended to aid in ensuring system safety. These procedures and practices are applicable to all MoD authorities and any projects for which they may be responsible. The standard has been produced for the MoD Defence Materiel Standardization Committee by the Safety Standards Review Committee and is solely intended for the MoD. [9]

This standard is split into two parts. Part 1 provides the specific requirements mandatory for any project following the standard. Part 2 is not mandatory but provides more detailed guidance and recommendations on how to fulfill the obligations described in Part 1. The standard's safety requirements in Part 1 center on the use of a safety case to show that an acceptable level of safety has been reached. The safety case consists of an argument, created by the contractor, for how system safety will be achieved in the specific case, followed by supporting evidence such as hazard analyses, hazard logs, testing logs, and other documentation intended to argue that a thorough safety process was followed and that the risk of a hazard occurring relative to this safety case is as low as reasonably practicable. The requirements for the design, production, and delivery of the safety case are discussed extensively in Part 1 of the standard and supporting guidelines, such as recommendations for hazard analysis, are

provided in Part 2. Part 2 of the standard also includes recommendations specifically pertaining to software, such as the use of detailed software safety requirements, quantitative risk-based testing, and interface verification and validation. [9]

Overall, DEF-STAN 00-56 is a goal-based standard that requires the creation of and proof of adherence to a contractor-created safety case. Rather than describing process-based requirements and techniques for safety assurance, the standard provides general safety requirements for the final product but does not direct how they are to be met. The contractor must propose and justify their chosen method of compliance. This puts the burden of proof on the contractor but also allows them flexibility in tailoring their approach to safety to fit the needs of their specific project.

NASA-STD 8719.13B

NASA-STD 8719.13B is a software safety standard that describes the requirements for technical procedures and practices for all NASA programs. This standard was produced by the NASA Office of Safety and Mission Assurance; a companion guidebook provides guidance on implementing the software safety program.

NASA-STD 8719.13B provides a systematic approach to software safety within the context of the safety of the total system. The standard considers a total life-cycle approach to software safety including the generation of requirements, design, coding, test, and operation of the software. The standard extensively addresses several aspects of software planning and development, including hazard analysis, traceability, testing, and validation. Requirements for software safety personnel qualifications and training are also thoroughly discussed. Unlike many other standards, NASA-STD 8719.13B thoroughly discusses the use of Commercial Off-The-Shelf (COTS) software tools and the evaluation, support, and approval of these tools in safety-critical systems and their development. [37]

This standard does not directly or explicitly discuss the issue of the safety benefit versus the cost of a safety program. However, it should be noted that the standard is intended only for use in NASA systems; NASA systems often require a high level of safety assurance. Although the standard does make a few concessions to cost concerns, such as allowing the size of the documentation and the traceability system chosen to reflect the size and criticality of the project, overall it does not directly address cost concerns or allow for much process tailoring.

It is difficult to determine the cost of software development using this standard. However, several high profile NASA projects, for which the use of this standard is mandatory, have had significant losses related to software and/or software-related problems.

DO-178B Guidelines

DO-178B is a software safety guidelines document jointly prepared by Special Committee 167 of the RTCA and the European Organization for Civil Aviation Equipment (EOCAE). This document offers guidance for the aviation community in producing reliable software that complies with airworthiness requirements. These recommendations reflect the avionics software community's consensus regarding best practices for development of safety-critical software. As a result, this document has strongly influenced much of software development in the civil aviation industry and some steps have been taken to use this guidance in military contexts as well [25]. The FAA recognizes this document as the primary means (but not the only means) of showing compliance to US law with respect to avionics computer software used in US civil aircraft airworthiness certification activities [42].

DO-178B is an evidence-based approach supporting certification for avionics systems and it requires a well-documented and executed system safety process based on engineering best practices. While DO-178B provides guidance on many safety issues, such as the use of integrity levels and testing, it also lacks information on other issues such as complexity management. Industry complaints have described insufficient discussion of configuration management, insufficient guidance for requirements definition and analysis, and inadequate and ambiguous guidance for partitioning, tool qualification, and COTS software. Furthermore, some have questioned the effectiveness of some activities discussed in the standard, including preparing documentation, tracing requirements to code, establishing independence, and demonstrating structural coverage [20]. In response to many of these issues over the years, the FAA has issued various Certification Authorities Software Team (CAST) position papers providing additional guidance and expectations for some of these key topic areas [15]. Overall, however, the objective-based approach of DO-178B makes it both rigorous and flexible for individual projects.

DO-178B introduces five different levels of safety criticality, ranging from Level A (most critical) to Level E (least critical). As the criticality level of the system increases, so too do the number of requirements for design, reviews, implementation, and verification/validation activities. As a result, cost often increases as well. Each increasing level of criticality of the standard could add significant costs to the development program.

Although there may be added expense for using DO-178B with respect to each level of safety criticality, these costs are balanced by the many benefits of using the approach, including greater requirements clarity, decreased coding iterations ("churn"), fewer bugs in critical code sections, and greater consistency of the software. These requirements and specific recommendations for safety in software have made DO-178B the '*de facto*' standard in airborne software for commercial, and in some cases, military avionics projects. It

is, however, widely believed in industry that compliance with DO-178B objectives can be expensive.

Software development using the DO-178B guidelines is often thought to contribute to the high costs of commercial aviation systems. The FAA, which requires showing compliance to DO-178B (or equivalent) requirements for civil aviation certification, has received many industry complaints regarding the time and expense involved with certification for high-criticality systems. As a result, the FAA created the Streamlining Software Aspects of Certification (SSAC) program in which industry and technical experts collaborated to examine whether the cost and time associated with certifying aircraft could be reduced [20].

In the SSAC program, anonymous opinions and feedback were solicited from over 400 industry representatives representing more than 70 companies regarding the cost, effectiveness, and efficiency of DO-178B [21]. Several major areas of concern were highlighted. First, issues related to inconsistency in the FAA's certification offices and program offices were an oft-cited concern. These issues included problems with varying and unclear documentation requirements, a lack of clear definition of when documents are due for certification, and unreasonable overly-conservative demands for compliance [20,21]. Second, it was noted that the lack of collaboration between companies contributed to increasing costs for software development and certification as there is no large industry-wide group that gathers data or researches new topics in these areas [20]. Finally, one of the greatest cost drivers cited was "poor requirements" [20].

In general, it is difficult to determine the exact costs introduced by using DO-178B. However, one source notes that many companies spend between 75% to 150% more for developing safety-critical software to meet the safety guidelines specified by DO-178B than for developing non-safety-critical software applications [23]. To cite another example, it is stated that developing software to show compliance to Level A of DO-178B raises the cost by a factor of five over non-critical software [2]. There is also a clear industry consensus that DO-178B is often very costly. These high costs often occur for many reasons. For example, the level of criticality chosen may be too high for the project and there may be a lack of understanding of how to effectively and efficiently manage the rigorous documentation and testing processes. Additionally, a poor application of software engineering best practices, such as proper requirements definition and management, can also play a significant role in cost issues.

3. Projects Incurring More Cost for Safety-Critical Software

Several high-profile projects have encountered budget overruns and schedule delays while developing safety-critical software. Although each of these projects claimed to follow one or more software safety standards, they faced significant challenges in their attempts to comply with their safety requirements and to comply in a cost-effective manner.

For example, the first McDonnell Douglas (now Boeing) C-17 military aircraft was delivered to flight test a year late and over budget in 1991. Development of the C-17 required use of MIL-STD-882D. McDonnell Douglas faced both design and production problems with the C-17 aircraft development; the development team chose not to use a computer-aided design approach to design the aircraft and manufacturing system. While the initial design approach of the aircraft was to use off-the-shelf components and software, the aircraft mission requirements ended up driving new ways for these off-the-shelf technologies to be used (e.g., complex avionics system to reduce crew size, new wing design for range and payload requirements). A change from a mechanical flight control system to an electronic fly-by-wire system was decided late in development. Integration of the on-board software-intensive avionics system was challenging [39]. The use of a clear, rigorous testing program is key to development of a safe system. It is suggested that the uncertainty of flight test program requirements is one of three key errors that caused the C-17's cost and schedule problems; technical risk in software and avionics integration and structural deficiencies in the wings were the other two [39].

One of the most important aspects of safety-critical software development is the effective determination and communication of requirements early in the design phase. In the early phases of the project, the C-17 project team failed to specify a single programming language to the various subcontractors who were developing different parts of the software [39]. Today companies can successfully use different programming languages to develop software for the same computing system. However, McDonnell Douglas faced significant cost and schedule delays in their software integration for the C-17. Over 3,500 subcontractors were identified on the team of which 33 were considered critical; management and integration of the subcontracted software proved extremely problematic for them [26].

Some aircraft development projects required to comply with DO-178B have also encountered significant cost and schedule issues. The Airbus SAS A380 super-jumbo jet was over 2 years behind schedule as of 2006 [33] and even after production start, is unable to reach profitability in 2010 due to cost overruns and the previous delays [45]. Building this jet aircraft cost over 50% more than originally planned due to software glitches in testing [45] and incompatible software used to design the aircraft at different factories [33]. The A380 team has been criticized for making the software too complex [16]. The widespread software problems plaguing this project would indicate that Airbus software developers may have experienced some difficulty in using best software development practices.

The Boeing 787 Dreamliner has also faced issues with the development and certification of their safety-critical software showing compliance to DO-178B. The 787 project has

experienced seven separate delays and is currently expected to enter production in the first quarter of 2011 over 2 years behind its original schedule. Initially, Boeing faced delays due to flight-control code and other crucial software provided by GE Aviation [17,18,27]. More recently, the software for the braking system did not meet traceability requirements specified in DO-178B and parts of the software had to be rewritten [44].

The Airbus A400M transport aircraft has also faced significant delays and expense. The aircraft is currently several years behind schedule and more than 7 billion Euros (about 9.5B USD) over budget [10,38]. The A400M is undergoing civil aviation certification which includes showing compliance with DO-178B [36]. For its engine, the Airbus A400M team planned to use an engine and engine controller that were qualified to military standards, not civil aviation standards. In order to obtain the civil aviation certification, the engine manufacturer was required to revamp the engine controller software and documentation to conform to the civil standards [43]. Airbus indicated that the engine controller software redesign is blamed for some of the 4-year delays experience by the program [43].

Navia Aviation of Norway developed a GPS-based Instrument Landing System for the civil aviation domain and planned to show compliance with DO-178B, Level B. Navia noted that their first attempt at certification failed; they continued, however, to improve their software development practices in order to show compliance [30]. During the inspection for certification, the approval rate of their documentation was only 25%. Navia attributed this low acceptance percentage to their underestimation of the rigorous nature of the approval process and the fact that the internal examinations prior to formal inspection were insufficient [30]. Navia recognized there was a problem, performed an analysis, and subsequently made changes to their inspection process; significant improvements in the accept rate of documents was observed. Finally, Navia stressed the importance of stability in the organizational environment (i.e., at the time of the project, Navia was reorganizing and resizing their workforce) and noted that change control was a difficult and expensive aspect of the process that was not previously considered [30].

Several FAA modernization programs have also faced cost and schedule problems such as the Wide Area Augmentation System (WAAS), Standard Terminal Automation Replacement System (STARS), and Airport Movement Area Safety System (AMASS) programs [22]. In all of these cases, software problems had significantly contributed to cost and schedule overruns. "*Software development – the most critical component of key FAA modernization programs – has been the Achilles' heel of FAA's efforts to deliver programs on time and within budget*," according to Gerald Dillingham of the United States General Accounting Office [46].

Overall, the observations related to these projects suggest that development of safety-critical software is far from a trivial issue and can often be prohibitively expensive. Because several of the projects previously discussed are still in progress, as well as facing significant issues, complete reports and detailed descriptions of the methods and processes used for producing their software are not generally available. However, it is clear that in many, if not most of these cases, more careful planning and more rigor in the software practices could have alleviated some of the problems and issues faced by these project teams. For example, the Airbus A400M, Boeing C-17, and Boeing 787 discussions indicated that if specific safety requirements are not determined and understood from the beginning, problems will inevitably follow. Traceability and careful documentation, as well as complete and thorough determination of these requirements among all software teams and subcontractors, must be provided from the beginning if development of safe software is to be achieved cost-effectively. And finally, lack of rigorous configuration and change control has been shown to play a key role in contributing to cost overruns for many projects.

4. Projects Incurring Less Cost to Produce Safety-Critical Software

On the positive side of this issue, evidence exists that developing software following rigorous software safety practices, such as embodied by DO-178B, does not have to be as expensive as previously expected. The use of appropriate development processes and mature methods designed to decrease development effort can enable projects to achieve certification in a much more cost-effective manner. In addition to using software engineering best practices, it has been shown that by using COTS software tools under the right conditions, organizations showing compliance to DO-178B can decrease project development times by as much as 25% and reduce the associated costs by as much as 70-80% [19]. Commercial tools specific to DO-178B and the aerospace industry have become available which automate some required safety processes. For example, *VAPS,* developed by Engenuity, enables automated display code generation, the *Integrity®-178B Real Time Operating System (RTOS)* from Green Hills® Software provides a DO-178B certification package with their software, *VectorCast* by Vector Software helps automate software unit testing and code coverage analysis, and *Reqtify™* from Greensoft facilitates requirements traceability and change analysis [24].

One such example of automation and methods helping with software development is the Lockheed Martin C-130J Hercules II aircraft development [2], which entered production in early 2001. Beginning in September 1992, the C-130J was a project that completely updated the very successful Lockheed Martin C-130 Hercules transport aircraft. Most of the changes involved updating avionics systems and software, including the advanced systems monitoring and navigation as well as an extensive integrated diagnostics system [8]. While the C-130 Hercules aircraft were largely mechanical systems, the C-130J Hercules II is a software-intensive system which uses mission computer software to improve its overall mission performance [8].

The C-130J was developed to Level A (the highest assurance level) of DO-178B for some parts of the software [8]. The project team reported that their development process for the safety-critical software was conducted for half the cost of non safety-critical code [1,2]. Additionally, Lockheed Martin reported that their testing process for the C-130J was conducted for less than a fifth of normal industry costs [2, 3]. Lockheed Martin saw additional benefits through their development approach: code quality improved by a factor of 10 over industry norms and productivity improved by a factor of 4 [3]. The C-130J approach to development bears examination to determine the various decisions that helped Lockheed Martin achieve such striking gains in both cost and safety. The factors that helped them achieve these lower cost and increased safety benefits included: (a) their choice of programming language, (b) a focus on software reuse, (c) a rigorous, requirement-based automated testing process, and (d) static code analysis. Development was driven by verification emphasizing "Correctness by Construction," and formality was introduced early in the specification phase. The programming language chosen for the project was SPARK, a subset of Ada with properties specifically designed for embedded and safety-critical applications. SPARK is unambiguous, free from implementation dependencies, all rule violations are detectable, and it is formally defined and tool supported [8]. The behavior of a program written in SPARK is entirely defined by and predictable by its source code.

One aspect of the language that made it most useful to Lockheed Martin was the SPARK Examiner, a development tool which emphasized static analysis as a way of ensuring correctness and reliability in the code. Static analysis performs a syntactic check of the code and ensures that coding errors recognized by the tool have been minimized. The SPARK Examiner automates a portion of the code review process and many problems are caught early during coding, allowing the developer to focus on more broad and higher-level issues with the code such as whether the code meets its specification.

Static analysis is most cost-effective when applied during software code development rather than in retrospect during the integration and testing process. The testing phase of a project is frequently a bottleneck and can be expensive for many projects [3], while static analysis can be conducted early during the coding and unit test phases. Errors can be caught at the developer's desk automatically. It has been argued that DO-178B has an undue emphasis on testing activities and places too little importance on analysis and review [3]. SPARK allowed Lockheed Martin to perform analysis on source code before the test phase of the project was entered, greatly decreasing the amount of time and resources spent on verification.

Overall, the C-130J project met its safety objectives and decreased its development costs through its "Correctness by Construction" and testing and verification processes. Objectives and requirements were specified and verified early in the development process and test cases were always based on requirements and were traced. Automated tools were used as much as possible to generate test cases and re-run these tests [8]. Software reuse was at the center of the C-130J development process. Through use of template-based design, Lockheed Martin reported productivity gains, improved reliability, and reduced testing overhead [8]. Source code, test scripts, and documentation were often reused among devices which allowed different teams to share data. In part due to the template-based design, source code could be reused for different device interfaces with only minor modifications resulting in vastly decreased development cost and time. Traceability and documentation reuse was also heavily emphasized throughout the project. In general, reuse significantly contributed to a lowered program cost [8].

Lockheed Martin observed efficiency gains through the use of common software development tools. These tools, which were configuration managed, allowed the company to save in terms of purchase expense and personnel training [8]. Lockheed Martin was able to allocate personnel among multiple teams due to their common knowledge and the common tools that were used throughout the projects. Automated tools for data collection assisted with the certification process. The C-130J team found certification activities challenging but reported that the introduction of automated data collection made it much easier to meet certification requirements and it saved additional development costs [8].

Following the successful development of the C-130J project, Lockheed Martin used the same development principles in a smaller project for the C-27J aircraft, which required Level B DO-178B software assurance. It was another successful use of the "Correctness by Construction" development method. During the C-27J project, the process used to develop the software for the C-130J project was reused and some code was re-used as well. The company reported an additional 4-fold productivity improvement over the C-130J project which gave a total of 16-fold productivity improvement [40].

Like the Lockheed Martin C130J and C27J, the UK MoD Ship Helicopter Operating Limits Information System (SHOLIS) project also achieved success in lowering cost and efforts in their testing process through the use of SPARK for code and static analysis. This project was created for the MoD to be used on the UK Royal Navy and Royal Fleet Auxiliary vessels by Ultra Electronics PMES with Praxis Critical Systems as a subcontractor responsible for all application software. This system was developed to comply with MoD DEF-STAN 00-55 and Interim DEF-STAN 00-56.

The development process for the SHOLIS used SPARK in the software design specification and code. The SHOLIS project used the Z notation for construction of the required formal specification and design with formal arguments linking the specification to the design to the code [29]. The SHOLIS team kept track of the number of faults found at different stages in the development process; the percentages are shown in Table 1 below. In this case, a fault is defined as an error in the system development that, if undetected, could lead to a fault in the final delivered system.

Table 1. Faults Found, Effort Spent During Project Phases [29]

Project phase	Faults found (%)	Effort (%)
Specification	3.25	5
Z proof	16	2.5
High-level design	1.5	2
Detailed design, code & informal test	26.25	17
Unit test	15.75	25
Integration test	1.25	1
Code proof	5.25	4.5
System validation test	21.5	9.5
Acceptance test	1.25	1.5
Other*	8	32

**Staff familiarization (1%), project management and planning (20%), safety management and engineering (7%) and IV&V non testing activities (4%)*

A comparison of the percentage of faults found to the percentage of development effort expended shows which phases were the most cost and effort efficient. It is clear that the detailed design, code, and informal test, as well as the unit test and system validation test phases were highly effective at finding a significant number of faults, but these activities also took a relatively large amount of development effort. In contrast, the SHOLIS team noted that the Z proof was effective at finding a significant number of faults with relatively little effort early in the development process. The major fault types found by the Z proofs were: incorrect functionality specified, contradictory operations, lack of mode/history information modeled, missing cases, and incorrectly loose specifications. SPARK proof work was accomplished by the SPARK examiner, simplifier and proof checker [29]. In general, it appears that the formal method of fault-finding employed by the SHOLIS project was a cost-effective and effort-efficient process.

An international survey conducted between November 2007 and December 2008 on the use of formal methods in safety-critical industrial projects found that on the whole, formal methods had a beneficial effect on time, cost, and quality. Three times as many respondents reported a reduction in time, rather than an increase, and five times as many respondents reported reduced, rather than increased, costs. 92% of all respondents believed that formal methods had a beneficial effect on the quality of their projects [49]. Based on these results, it seems that formal methods are finding their place as an effective and perhaps cost-efficient form of assurance in safety-critical software systems.

In addition to the formal methods and software development practices described above, carefully considered application of COTS tools has also helped several large projects reduce their costs in developing safety-critical software. One such example is the Eurocopter EC135 project. Eurocopter is a large producer of both civil and military helicopters with over 11,000 aircraft sold around the world. The EC135 and EC155 civil helicopters, with extensive autopilot systems, were both developed to show compliance to DO-178B Level A. Eurocopter used the SCADE specification and code generation tool from Esterel Technologies [7] in order to reduce development time, certification time, and costs. As a result, 90% of the code was generated by the SCADE tool and development time was reduced to 50% of the time needed to manually code an equivalent system. The certification was successfully completed [6,7].

Airbus also used the SCADE automatic code generation tool to develop the software for the Flight Control Secondary Computer used by its A340/500 aircraft. The amount of automatically generated code on the A340 was as high as 70% of the total code [35]. Moreover, there were no errors found in the SCADE generated code. Airbus reported a reduction in modification cycle time by a factor of 3 to 4 compared to manual coding through use of code generation. As a result, time-to-market was reduced for the A340 and the project successfully showed compliance to DO-178B Level A [6,7].

Success at employing DO-178B has also been observed through the use of other development and testing tools. For example, MDS Technology, based in South Korea, was able to develop their NEOS real-time operating system to comply with Level A of DO-178B in a cost-effective way by using the commercial VectorCAST testing tool. MDS Technology reported an 83% reduction in testing time and were able to complete their project within a short period despite the rigorous DO-178B guidelines [47]. Ulta Electronics Datel was able to show DO-178B Level B compliance for a significant upgrade to its previous avionics display computer. The upgrade involved moving from a proprietary hardware system to a COTS-based solution, using the Wind River VxWorks DO-178B safety-critical subset operating system, and re-engineering pre-existing software and device drivers to meet the higher software assurance requirements required to show compliance to DO-178B. Datel used the VAPS code generation tool [28] and the LDRA tool suite to successfully reduce the amount of effort for the requirements specification, design, and test phases of their avionics display development when compared to the previous development effort. As a result, they reported 25% faster time to market and 87% lower development costs [28].

Many of these successful projects used COTS tools in order to achieve some of their cost savings. Starting in the second half of the 1990s, NASA too began to study the use of COTS tools in order to save on development costs in their Software Engineering Laboratory (SEL). The process improvements that resulted from this change caused the SEL to decrease costs by 10% and shorten schedules by 5-20% [5]. However, despite these obvious benefits, the use of COTS in safety-critical applications is a relatively new phenomenon and most safety standards offer little guidance on the development of safety assurance for COTS software [32]. Despite the potential benefits offered by COTS products, the usage and integration of these products into safety-critical systems should be approached carefully with a thorough understanding of the risks involved. For example, given the long shelf-life of many safety-critical systems, concerns about vendor tie-in and future availability are an issue to consider. If a COTS system is used in a project, the project may have to rely on the COTS vendor for support, modifications, and future availability [12].

Furthermore, the integration effort required may add considerable overhead and increase development costs [1, 34]. Due to the increasing trend towards the use of COTS software in safety-critical systems, safety standards committees should likely consider including further practical guidance on the proper use of these tools.

These successful examples indicate that it is possible to develop software with significant safety assurance requirements without sacrificing project cost effectiveness. Specifically, projects can realize cost savings by using mature processes and appropriate tools to assist in their development and certification.

5. Conclusions

The increasing flexibility, extensibility, and automation capabilities of software have caused a growing number of industries to depend on it in the development and operation of safety-critical systems. Due to the nature of these systems, safety is paramount and the development of safe software is therefore a high priority. However, designing for safety comes with some costs. Despite the abundance of software safety standards which exist to provide guidance for the development of safe software in safety-critical systems, there is no consensus on how to achieve safety in a cost-effective fashion.

In this paper we have reviewed five software safety standards: the FAA System Safety Handbook, the US DoD MIL-STD-882D, the UK MoD DEF-STAN 00-56, NASA STD 8719.13b, and the RTCA DO-178B. We have reviewed each of these documents in terms of their cost effectiveness and the tradeoff between cost and safety. Furthermore, we have examined several projects which have encountered significant cost and schedule issues in their development of safety-critical software. It is clear from these examples that high costs in safety-critical software development are often incurred due to insufficient communication, poor requirements, and poor or ineffective planning. This indicates that although insufficient guidance in some software safety standards may have played a role in cost and schedule problems faced, numerous technical, managerial, and company environmental factors all can contribute to project difficulties. No single factor can be identified as 'the' contributing cause; however, in most of the projects studied, a lack of rigor using the company documented software engineering processes and practices seemed to be a recurring theme.

We have reviewed several large projects which have managed to cost-effectively develop safety-critical software which adhered to relevant safety standards. These projects demonstrated that it is clearly possible to achieve software safety objectives without sacrificing cost and schedule and that this can be achieved through a reliance on effective planning and the application of software engineering practices throughout the development lifecycle. Additionally, techniques are making their way into general practice that provide for further potential cost reduction, including use of formal methods and COTS tools. The risks and benefits of these techniques are an important point for further study and the rapid adoption of these methods in many industry contexts means that similar focus should be placed on ensuring that effective guidance is provided in software safety standards, many of which do not address the certification of tools or COTS software. The movement towards results-oriented rather than process-based standards does provide some flexibility for industry projects to use these methods, but a lack of thorough guidance on their proper use, as well as their advantages and disadvantages could pose cost, schedule, and safety issues for future product developments. In general, however, software safety standards provide useful and effective guidance but should not necessarily be blindly followed. Each organization must carefully plan and tailor their safety approach to the particular needs of the project focusing on safety engineering best practices as evidenced in their company documented engineering practices in order to cost effectively deliver safe software.

Acknowledgements

This work is supported by the National Science Foundation (NSF CCF-0851848). The authors would also like to thank the Industrial Advisory Board members of the Net Centric Industry/University Collaborative Research Center for providing additional funding for the research reported here.

References

1. Abts, C., Boehm, B. W., and Clark, E. B. (2000). *COCOTS: A COTS Software Integration Lifecycle Cost Model – Model Overview and Preliminary Data Collection Findings.* USC Center for Software Engineering.
2. Amey, P. (2002) *Correctness by construction: better can also be cheaper,* CrossTalk Journal, March 2002.
3. Amey, P. and A. Hilton. (2001) *Practical experiences of safety- and security-critical technologies,* Ada User Journal, March 2001.
4. ArianeSpace. (2011). *Ariane 5 Milestones.* http://www.arianespace.com/launch-services-ariane5/milestones.asp
5. Basili, V. R., McGarry, F. E., Pajerski, R., & Zelkowitz, M. V. *Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory*. (2002). In: Proceedings of the 24th International Conference on Software Engineering.
6. Camus, Jean-Louis. Esterel Technologies, (2002). *Efficient Development of Avionics Software with DO-178B Safety Objectives.*
7. Camus, Jean-Louis, and Bernard Dion. Esterel Technologies, (2002). *Efficient Development of Airborne Software with SCADE Suite*.
8. Conn, R. and S. Traub and S. Chung. (2001) *Avionics modernization and the C-130J software factory,* CrossTalk Journal, September 2009.
9. Defence Standard 00-56. *Safety Management Requirements for Defence Systems.* Issue 4, 1 June 2007.
10. Defense Industry Daily. (Nov. 2010). *Airbus' A400M Aerial Transport: Delays and Development.* http://www.defenseindustrydaily.com/A400M-Delays-Creating-Contract-Controversies-05080
11. Department of Defense. (2000). *MIL-STD-882D Standard Practice for System Safety.*
12. Dewar, Robert B. K. (Nov. 2000). *COTS software in critical systems: The case for Freely Licensed Open Source Software.* Military Embedded Systems.
13. DOD Software Tech News (2011), Tech Views – Challenges Dominate Our Future, Ellen Walker.
14. Federal Aviation Administration. (2000). *FAA System Safety Handbook.*
15. Federal Aviation Administration. (2009). *Cast Position Papers.* http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/

16. Flightglobal, (2010). A380 In-service report: Technical issues. http://www.flightglobal.com/page/A380-In-Service-Report/Airbus-A380-In-Service-Technical-issues/
17. Gates, Dominic. The Seattle Times, (2007). 787 flight delay blamed on unfinished structures, software.
18. Greising and Johnsson. Chicago Tribune, (2007). Behind Boeing's 787 delays.
19. Hawthornethwaite, Mark, and Luc Marcil. VME and Critical Systems, (2007). *Paving the road to DO-178B compliance with COTS tools.*
20. Hayhurst, K., Holloway, C., Dorsey, C., Knight, J., Leveson, G., McCormick, G., and Yang, J. (1998). *Streamlining Software Aspects of Certification: Technical Team Report on the First Industry Workshop.* National Aeronautics and Space Administration, Langley Research Center. NASA/TM-1998-207648
21. Hayhurst, K., Holloway, C., Dorsey, C., Knight, J., Leveson, G., and McCormick, G. (1999). *Streamlining Software Aspects of Certification: Report on the SSAC Survey.* National Aeronautics and Space Administration, Langley Research Center. NASA/TM-1999-209519.
22. Hayhurst, K., and Holloway, C. (2001). *Challenges in Software Aspects of Aerospace Systems.* Proceedings of the 26th Annual IEEE NASA Software Engineering Workshop.
23. HighRely Inc. (2005) *DO-178B and DO-254: Big Bang or Evolution?*
24. HighRely Inc. (2005). *DO-178B Costs versus Benefits*
25. Hilderman, V. (2009). *DO-178B and DO-254: A unified aerospace-field theory?* Military Embedded Systems Magazine.
26. Hopkins, J.R. and De Keyrel, C.R. Master's Thesis, US Air Force. *An Analysis of the Root Causes of Delays and Deficiencies in the Development of Embedded Software for Air Force Weapons Systems.* December 1993.
27. Inca Group War and Peace, (2007). Boeing 787, the new US flying coffin.
28. James, Parkinson, and Roberts. Wind River Systems, (2009). *Case Study: Ultra Datel Safety-Critical Avionics Upgrade Using COTS*
29. King, S., Hammond, J., Chapman, R., Pryor, A. *Is Proof More Cost Effective Than Testing?* (August 2000). IEEE Transactions on Software Engineering, vol 26, no. 8.
30. Kvinnesland, Kenneth. Navia Aviation, (2002). *Implementation of metrics in development of highly safety-critical SW.*
31. Leveson, N. *Safeware: System Safety and Computers.* Addison-Wesley, September 1995.
32. Lindsay, P. and G. Smith. Technical Report No. 00-17. *Safety Assurance of Commercial-Off-The-Shelf Software.* (May 2000). Software Verification Research Center, University of Queensland, Australia.
33. Matlack, Carol. Business Week, (2006). Airbus: First, Blame the Software
34. Maxey, Burke. *COTS Integration in Safety Critical Systems Using RTCA/DO-178B Guidelines.* H. Erdogmus and T. Weng (Eds.): ICCBSS 2003, LNCS 2580, pp. 134-142.
35. Menendez, Jose K. *Building Software Factories in the Aerospace Industry.* MS Thesis, Massachusetts Institute of Technology, February 1997.
36. Mikro Elektronik, (2010). In-flight equipment for Airbus A400M.
37. National Aeronautics and Space Administration. NASA-STD 8719.13. *Software Safety.* Revision B with Change 1, 8 July 2004.
38. People's Daily, (2010). France, Britain rule out sharing aircraft carriers.
39. Pike, John. GlobalSecurity.org, (2005). C-17 Globemaster III - History. http://www.globalsecurity.org/military/systems/aircraft/c-17-history.htm
40. Praxis Critical Systems. *SPARK: A successful contribution to the Lockheed C130J Hercules II*
41. Radio Technical Commission for Aeronautic, Inc. (Dec. 1, 1992). *RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification*.
42. Radio Technical Commission for Aeronautic, Inc. (1993). *FAA Advisory Circular 20-115B.*
43. Reuters, (2009). Airbus A400M software revamp almost ready.
44. Rigby and Hepher. Reuters, (2008). Brake software latest threat to Boeing 787.
45. Rothman, Kammel, and Harris. Bloomberg, (2010). Airbus A380 Order Dearth Risks Double-Decker-Dud Fate (Update1).
46. United States General Accounting Office. *National Airspace System: Problems Plaguing the Wide Area Augmentation System and the FAA's Actions to Address Them.* Statement of Gerald L. Dillingham, Associate Director, Transportation Issues, Resources, Community, and Economic Development Division before the Subcommittee on Aviation, Committee on Transportation and Infrastructure, House of Representatives GAO/T-RCED-00-229, June 29 2000.
47. Vector Software, (2010). *MDS Technology relies on VectorCAST for DO-178B Level A certification testing.*
48. Wong, E.W., Debroy, V., Surampudi, A., Kim, H. Department of Computer Science, University of Texas at Dallas. With Mike F. Siok, Lockheed Martin Aeronautics Company. *Recent Catastrophic Accidents: Investigating How Software Was Responsible.* Fourth IEEE Conference on Secure Software Integration and Reliability Improvement, 2010.
49. Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. *Formal Methods: Practice and Experience.* (October 2009). ACM Computing Surveys, Vol. 41, No. 4, pp. 1-40.