Journal of Software Engineering and Applications, 2014, 7, 162-176
Published Online March 2014 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2014.73018

Scientific Research

A Quality Assurance Model for Airborne Safety-Critical Software

Nadia Bhuiyan[1], Habib A. ElSabbagh[2]

[1]Departmentof Mechanical and Industrial Engineering, Concordia University, Montreal, Canada
[2]Xmile Inc., Montreal, Canada
Email: nadia.bhuiyan@concordia.ca

Received 8 December 2013; revised 5 January 2014; accepted 13 January 2014

Copyright © 2014 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

The tragic nature of safety-critical software failure's consequences makes high quality and extreme reliability requirements in such types of software of paramount importance. Far too many accidents have been caused by software failure error or where such failure/error was part of the problem. Safety-critical software is widely applied in diverse areas, ranging from medical equipment to airborne systems. Currently, the trend in the use of safety-critical software in the aerospace industry is mostly concentrated on avionic systems. While standards for certification and development of safety-critical software have been developed by authorities and the industry, very little research has been done to address safety-critical software quality. In this paper, we study safety-critical software embedded in airborne systems. We propose a lifecycle specially modeled for the development of safety-critical software in compliance with the DO-178B standard and a software quality assurance (SQA) model based on a set of four acceptance criteria that builds quality into safety-critical software throughout its development.

Keywords

Safety-Critical Software; Software Quality Assurance; Airborne Systems

1. Introduction

Embedded systems are special-purpose computer systems, which are encapsulated or mounted into the device they are built to control. Embedded systems play a critical role in human beings' daily life. They are present in a wide range of applications from personal systems to large industrial ones. In modern embedded systems, there is always a software element. Furthermore, that software is typically used to control a larger system of which it is only one part. A safety critical system is a system where a malfunction can result in loss of life, injury or illness,

How to cite this paper: Bhuiyan, N. and ElSabbagh, H.A. (2014) A Quality Assurance Model for Airborne Safety-Critical Software. *Journal of Software Engineering and Applications*, **7**, 162-176. http://dx.doi.org/10.4236/jsea.2014.73018

or serious environmental damage. The tragic nature of safety-critical software failure's consequences makes high quality and extreme reliability essential requirements.

While standards for certification and development of safety-critical software have been developed by authorities and the industry, very little research has been done addressing safety-critical software quality. The research in this paper looks into a number of questions. Is safety-critical software quality controllable and manageable like any other software? How should safety-critical software quality be built and managed? After studying safety-critical software quality and demonstrating how it differs from other software and products, we briefly discuss a lifecycle model we developed for safety-critical software development activities. We then propose a new software quality assurance model specially designed for the development of such software products. The proposed model advocates building quality into the product right from the moment the development project is initiated. The objective is to produce a high quality, reliable, and certifiable safety critical software on time, within budget, and through a development lifecycle free of iterations that are common in software development. Our work mainly targets safety critical software projects developed for aerospace applications. Furthermore, the main interest is "make-to-order" projects which have more specific requirements and operational functionalities in comparison to projects for commercial off the shelf products.

Methodology

Literature on previous work done in the field of airborne safety-critical software was meticulously investigated and studied. In addition, software quality standards and guidelines were reviewed and the standard DO-178B, which is the de facto standard in the aerospace industry, is at the center of interest. The software quality assurance proposed in this paper is directly related to and complementary to the DO-178B. Our research is also based on the experience and lessons learned while working on software quality assurance and certification issues on a collaborative research project entitled Dynamic Test Bed (DTB) for Flight Management System (FMS). The project is a research and development effort based in an aerospace company: it deals with the development of a real-time Dynamic Test Bed (DTB) which serves as a comprehensive design/test tool for the Flight Management System (FMS) product line at the company. The DTB is used to simulate the FMS with a real aircraft environment and conditions. The main objective is to reduce the number of flight tests by getting flight test credit by Transport Canada, prepare and run the entire engineering and Transport Canada flight-test, and reproduce customer problems with the same conditions under the DTB. While a failure of the DTB will not result in any human tragedy or injury, the software components of the DTB are considered to be safety-critical since they simulate real-time flight scenarios used to test FMS's. A failure in a flight scenario might not detect an error in an FMS which once installed on an airplane might fail and result in terrible consequences.

The paper is organized as follows. Section 2 provides a review of existing research done on safety critical software quality literature. Section 3 explores software quality and how quality control, methods, and tools can be applied to safety-critical projects. Software quality assurance and its benefits are then discussed and how it can benefit safety critical software projects. Section 4 presents a lifecycle model for the particular case of safety-critical software development projects. In Section 5, the software quality assurance model proposed in this paper is presented. Finally, we conclude in Section 6 and provide an outlook on future work.

2. Literature Review

While software quality has been studied thoroughly, little research has been done on safety critical software quality and quality assurance [1]. The early studies were interested in whether independently developing variants of the same software could contribute to the increase of safety. These studies dealt with the quality of safety-critical software as being synonymous with reliability; if the software is reliable then it is of good quality and hence safe. The basic idea has been to first produce independently two variants of software to solve the same task then demonstrate that each single variant has achieved certain reliability, and finally conclude the probability of common failure by multiplying the probability of failure of the first software by the probability of failure of the second.

Whether certification makes a difference in the quality of the final product or whether other factors, such as Human Factors, are at work was questioned in [2]. They argued that due to the tragic impact of an error in their codes, safety-critical software developers would meticulously check their work and review it with their peers.

However, a quality assurance model would still be required to insure quality requirements.

The quality of software development tools which are defined by DO-178B as "computer programs used to help develop, test, analyze, produce or modify another program or its documentation […] tools whose output is part of airborne software and thus can introduce errors into that airborne software" was studied [3]. They experimentally demonstrated that the proper assessment process could provide insight into a tool's quality and provide a basis on which to decide on how well the tool meets qualification criteria.

It was argued that certain software concerns that directly affect safety are not within the scope of the DO-178B or not included at all [4]. They recommended the use of DO-178B as an assurance standard along with additional guidance material.

In 1992 the RTCA introduced DO-178B as a guideline for safety-critical development and certification. While it emphasizes transition criteria from a development stage to the next it does not exclude the use of agile methods. It was demonstrated how agile practices can improve a software process for the development of avionics software in [5]. ISIS is a navigation system developed by German company Andrena Objects Agfor process and software quality management in an agile software development environment [6]. The tool is based on several metrics such as: number of bugs, deviation from approximated time usage, and compiler warning. While the authors argued that ISIS captures the quality of a piece of software at a given moment accurately and follows its development process sensitively they have assumed they can draw conclusions regarding the whole quality of the software by only analyzing parts of it.

Another approach to safety-critical software quality is trying to apply to software systems safety analysis techniques successfully used in non-computer-based systems. Fault Tree Analysis (FTA), Failure Mode and Effect Analysis (FMEA), and Hazard Operability Analysis (HAZOP) are examples of those techniques. FTA is a graphical representation of the major faults or critical failures associated with a product, the causes for the faults, and potential countermeasures. The tool helps identify areas of concern and corrective actions. FMEA is a quality planning tool that examines potential product or process failures, evaluates risk priorities, and helps determine remedial actions to avoid identified problems. FMEA mainly investigates the following nine areas: Mode of Failure, Cause of Failure, Effect of Failure, Frequency of Occurrence, Degree of Severity, Chance of Detection, Risk Priority, Design Action, and Design Validation. HAZOP is based on a theory that assumes that risk events are caused by deviations from the design or operating intentions. It is a systematic brainstorming technique for identifying hazards. It often uses a team of people with expertise covering the design of process or product and its application.

In many cases, these techniques cannot be directly applied to software due to special characteristics of software-controlled applications like discrete nature of processes, complexity, domination of design faults, or real-time constraints.

In order to overcome this problem, some tried adapting those techniques to software and some looked into extending the techniques to software safety analysis. For instance, FTA was adapted to safety-critical software [7] while a method was developed to adapt HAZOP with respect to computer systems [8]. A "method of informal safety analysis of software-based systems using FTA and FMEA" was introduced in [9]; this work has been applied in the nuclear industry. On the other hand, theoretically "extending the FMEA method to make it suitable to analyze object-oriented software designs" was studied in [10].

A "safety improvement process" which "must assure that the software safety analysis is closely integrated with the system safety analysis and the software safety is explicitly verified" was described in [11]. They emphasized the analysis first and verification next.

In a report for the British Computer Society, it was suggested to focus the assessment of safety-critical software on the development tools used, which "normally have direct influence on the safety system, such as compilers, but also design tools that generate code for safety related target systems" [12]. A number of criteria were proposed for development tools assessment.

Studying the quality of safety-critical software used in nuclear reactors control systems, a new direction for the use of formal (*i.e*. mathematical) methods was suggested [13]. A set of formal methods is a methodological framework for system specification, production and verification. The authors proposed using formal methods for establishing regulatory requirements which "could help to eliminate various understanding of problems or ambiguity of informal definitions, to allow rigorous assessment of satisfaction to requirements and finally to increase the safety level of a system".

3. Software Quality

Software quality was defined as "the composite of all attributes which describe the degree of excellence of the computer system" [14]. This definition was further developed in [15] by stating that software quality is "the degree to which a software product possesses a specified set of attributes necessary to fulfill a stated purpose". Software Quality Control is defined in [14] as "the assessment of procedural and product compliance. Independently finding these deficiencies assures compliance of the product with stated requirements". According to [15], "Software Quality Control is the set of verification activities which at any point in the software development sequence involves assessing whether the current products being produced are technically consistent and compliant with the specification of the previous phase."

Quality Control methods rely on quantitative inputs obtained from measurements and tests. Measurements are relatively easily made in physical products (hardware) since they mostly consist of measurable metrics such as length, weight, time, temperature, or logical metric (true or false). However, software products cannot be measured using those tangible metrics. This raises two important problems: how to measure software and how to apply Quality Control methods to safety-critical software products?

In order to answer those two questions, the following sections first look into work done on software measurement and identifies metrics that are used in the case of safety-critical software. Then, the feasibility of testing safety-critical software for Quality Control is analyzed.

3.1. Software Quality Measurement

The first attempt to create a method for software quality measurement appears to be the work in [16] where quality factors were considered to be based on elementary code features that are assets to software quality. They defined attributes and metrics for software code. Rubey and Hartwick defined seven high-level attributes, and the presence of each attribute in the software is ranked on a scale of 0 to 100 which gives a numerical expression of the conformance of the software with the attributes. According to the authors, this can be used as a metric to measure and control software quality. The importance of characterizing and dealing with attributes in terms of software quality was also recognized in [17]. He identified seven non-overlapping attributes: robustness, cost, human factors, maintainability/modifiability, portability, clarity, and performance. As the trend in software use became widespread, more people emphasized software quality and the importance of establishing good programming practices.

The most known methodology for software inspection can be found in [18]. It consists of seven steps: planning, overview, preparation, inspection, process improvement, re-work, and follow-up activities. The goals are to detect and remove errors in the work products. Fagan maintained that "there is a strong economic case for identifying defects as early as possible, as the cost of correction increases the later the defect is discovered".

3.2. Safety-Critical Software Quality Measurement

The interest in safety-critical software products is to have an output conforming to all design requirements. A failure in producing such an output may lead to tragic consequences on human life. Therefore the target is to have zero failure.

Failure can be measured by a failure rate (θ) which is the ratio of the number of defective units in a population to the total number of the population. In software, a defective unit is a defective output. The defect rate is expressed mathematically as follows:

$$\theta = \frac{D}{N} \quad (1)$$

where θ is the defect rate

D is the total number of defects in the population

N is the size of the population

In probability, statistics, and quality control studies, the defect rate (θ) has always been related with the binomial distribution. The binomial distribution is used when there are exactly two mutually exclusive outcomes of a trial, either a success or a failure. The term "trial" refers to each time a physical item is tested to check if it is defective or not. When the product tested is software, "trial" refers to each time an input is given to the soft-

ware program/function in order to verify the output. The formula of the binomial distribution is given by:

$$P(x)=b(x)=\binom{n}{x}\theta^{x}(1-\theta)^{n-x} \tag{2}$$

Equation (2) gives the probability of getting x failures in a sample of n units taken from a population having a defective Analysis of Safety-Critical Software Testing

S, using Equation (3) the probability of failure $P(S')$ of safety-critical software products can be computed as follows:

$$P(S')=P(x>0)=1-P(S)=1-P(x=0)=\binom{n}{0}\theta_S^0(1-\theta)^{n-0}=1-(1-\theta_S)^n$$

$$P(S')=1-(1-\theta_S)^n \tag{3}$$

where: θ_s is the failure rate of the whole product

n is the number of inputs tested

A software product (S) is a set of several subroutines and functions (s). Based on Equation (3), the probability of failure of a certain subroutine $s_i\left(P(s_i')\right)$ can be expressed by the following equation:

$$P(s_i')=1-\left(1-\theta_{s_i}\right)^{n_{s_i}} \tag{4}$$

where: θ_{s_i} is the failure rate of the subroutine

n_{s_i} is the number of inputs tested

The failure of a safety-critical software product (S') occurs when at least one of its composing subroutines or functions fails (s'). Therefore, $P(S')$ is true if any $P(s_i')$ is true for $i=1\cdots K$.

A general formula for $P(S')$ in terms of s_i can be derived when s_i's are independent from each other:

$$P(S')=1-\left(1-\theta_{s_1}\right)^{d_1 n_{s_1}}\left(1-\theta_{s_2}\right)^{d_2 n_{s_2}}\cdots\left(1-\theta_{s_k}\right)^{d_k n_{s_k}}$$
$$P(S')=1-\sum_{i=1}^{k}\left(1-\theta_{s_i}\right)^{d_i n_{s_i}} \tag{5}$$

The formula can now be used to derive all parameters required for the analysis of the feasibility of testing safety-critical software.

4. Analysis of Safety-Critical Software Testing

Testing is fundamental in product quality control. Most quality control methods are based on data and statistics collected from testing the products and/or their parts. In order to answer the question of how to apply quality control to safety-critical software, it is important to check whether safety-critical software can be tested or not.

In the following an analysis first looks at how long it will take to test safety-critical software products by means of two approaches: Time Functions and Growth Models. Then, the analysis investigates the assumptions of subroutines independencies and failure replacement and their effect on testing time if any.

4.1. Testing Time Using Time Functions

The customary way of checking the performance of a software product is through testing. The interest is to find defects and report them to be fixed. Therefore, the testing time is the same as the expected run time to failure (R_t). Safety-critical software products "are designed to attain a probability of failure $\left(P(S')\right)$ on the order of 10^{-7} to 10^{-9} for 1 to 10 hour missions" [19]. A $P(S')$ within this small range will lead to an extremely long testing time as illustrated in the following:

The Expected Run Time to Failure (R_t) is the estimated length of time the software will run before f failure occur in l loop or iteration.

$$R_t=M_t\frac{f}{l} \tag{6}$$

where M_t is the mean time to failure (*i.e.* the average time that elapses before finding the first failure). Given

Equation (7) of $P(S')$ in terms of M_t:

$$P(S') = 1 - e^{-\frac{t}{M_t}} \tag{7}$$

then R_t can be derived as a function of $P(S')$ where:

$$R_t = \frac{t}{-\ln(1 - P(S'))} \frac{f}{l} \tag{8}$$

If $P(S')$ is 10^{-9} for a 10 hours mission ($t = 10$) then the testing time to find the first failure ($f = 1$) in 1 loop ($l = 1$) will be almost equal to 1,141,550 years! It will take 1,000,000 loops (l) to decrease the testing time to about 1.15 years!

4.2. Testing Time Using Growth Models

"In growth models the cause of failure is determined; the program is repaired and is subject to new inputs. These models enable [the estimation of the] probability of failure of the final program" [19]. Growth models were experimented with in order to determine an expected testing time for six different software programs [20]. They derived a log-linear model from which estimation can be made for the time needed to reach a certain probability of failure $(P(S'))$. The result was even longer than what has been computed when $P(S') = 10 - 9$. "Clearly, growth techniques are useless in the face of such ultrahigh reliability requirements. It is easy to see that, even in the unlikely event that the system had achieved such a reliability, we could not assure ourselves of that achievement on an acceptable time" [21].

4.3. Independency

So far, only the case where all subroutines and functions (s_i) composing the end software product (S) being independent from each other has been considered. All formulas used took advantage of the independency property of an intersection of two probabilities ($P(s$'1($P(s_1' \cap s_2') = P(s_1') \times P(s_2')$). This property simplified calculations. In software, it is common to have interdependent subroutines and functions. When the independency does not hold, the computations performed still hold since $P(s_i')$ is also required to be extremely small.

4.4. Failure Replacement

When using the binomial distribution, an assumption is made that each time a failure is found it is fixed (or replaced) and the sampling continues into the next failure. When sampling is done without replacements, the geometric distribution should be used. However, as demonstrated in [19] the "expected [testing] time with or without replacement is almost the same".

It has been shown that performing tests on safety-critical software products is infeasible given the ultrahigh reliability requirements for such products. In addition, a conclusion can be made that Quality Control techniques in their mathematical and statistical forms cannot be applied to safety-critical software products. Therefore, it is necessary that credible procedures be developed to ensure that quality and reliability are built into safety-critical software products from the first stage of their lifecycle. These procedures are better known as Quality Assurance.

4.5. Software Quality Assurance

The American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE) define Software Quality Assurance (SQA) as "a planned and systematic pattern of all actions necessary to provide adequate confidence that the software product conforms to established technical requirements" [22]. This definition stresses that actions taken should be "planned and systematic" in order to achieve quality with "confidence". The philosophy championed is "Prevention of non-conformities from the start rather than Detection at the end" [23]. This prompts the need for a procedure assuring that a product conforms to functional and design performance criteria during all levels of the development process; this procedure is to prevent non-conformities and detect the root cause of possible errors while making sure they will not recur. Accordingly, "software quality assurance lays considerable stress on getting the design right prior to coding" [24].

Over the years, the software industry witnessed a high percentage of projects going over budget and/or over schedule. In 1995, US companies spent an estimated $59 billion in software project cost overruns and another $81 billion on cancelled software projects [25]. In another survey of 72 information system development projects in 23 major US corporations, the average effort overrun was 36% and the average schedule overrun was 22% [26]. Given this fact and that software development teams tend to sacrifice quality in order to meet budget and schedule, the SQA definition given by ANSI and IEEE to include budget and schedule concerns was rewritten [27]. Accordingly, SQA is "a systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines" [27].

SQA activities are usually the responsibility of one person, an SQA team, or a whole department depending on the organization and the project. Typically those activities include [28]:

1) SQA Planning: prepare a software quality assurance plan which interprets quality program requirements and assigns tasks, schedules and organizational responsibilities;

2) Software testing surveillance: report software problems, analysis of error causes and assurance of corrective action;

3) Software verification: verify the correctness of the software;

4) Software validation: validate that software complies with the requirements;

5) Software development process: audit the development process;

6) Quality promotion: promoting quality and its importance in the organization;

7) Certification: prepare the software product to pass any needed certification requirements;

8) Records keeping: keep design and software problem reports, test cases, test data, logs verifying quality assurance reviews and other actions.

The SQA team is independent from the software development team; people building the software cannot be the same as those checking its quality.

4.6. SQA Tools

Software Quality Assurance is extremely important in the world of safety-critical software. The success of such a product and the success of the organization producing it highly depend on the high quality of the product and hence on the SQA implemented. Therefore, tools are required to monitor the SQA plan and ensure that it is leading to its goals. Moreover, these tools can be used by the organization to compare its SQA processes between different projects, monitor its SQA improvements from a project to another. It is essential for software organizations to learn from mistakes made in order to prevent them in the future, SQA tools can help with that too.

The following provides an overview of different SQA tools that can be applied in safety-critical software production environment. Those tools can be grouped into two categories: Error Management Tools, and Managerial Tools.

5. Standards and Models

Since the early 1970s, several institutes, associations, and governmental organizations tackled the importance of software quality by developing standards and quality assurance methods. Those organizations include: the US Department of Defence (DOD), the US Federal Aviation Administration (FAA), the Institute of Electronic and Electrical Engineering (IEEE), and the North Atlantic Treaty Organization (NATO). The first major standard, entitled "Software Quality Program Requirements", was released by the US Army in 1975 and; it is better known by its reference name: MIL-S-52779. Every standard and guideline developed since then was influenced by this landmark standard. In general, the two most common and widely used standards and models are the Capability Maturity Model (CMM) and SPICE. CMM and SPICE are currently the two widely used standards and guideline models for software process assessment and improvement. Their successful implementation assists organizations in increasing the maturity and capability of their software development processes. CMM and SPICE were developed for all types of software production organizations and their integrity has been proven. However, they are too general and wide-ranging, they tolerate small share of defects and errors even at the most advanced levels. This nature of CMM and SPICE makes them weak and ineffective for implementation on safe-

ty-critical software production due to the characteristics and extremely high quality and reliability requirements of such products. Therefore, several standards dedicated specially to safety-critical software production have been published. Those standards are reviewed in the following.

Standards for Safety-Critical Software

A number of differing standards for safety-critical software already exist or are currently under development. Most of those standards are designed specifically to a particular industrial sector or even to a type of application within a sector. Mainly, three sectors dominate most of the standards' studies and developments: aircraft industry (particularly airborne software systems), nuclear energy (particularly reactors control software), and railway transport (particularly signalization systems).

In general, all standards designed for safety-critical software share the following two main tasks: 1) Establishing regulatory requirements for systems and software; *i.e.*, requirements (criteria, norms, rules) important for safety; 2) Assessing the system and software. The purpose of this assessment is to be convinced that the system and software answer established regulatory requirements. In the field of aircraft related and airborne software development, DO-178B and its updated revisions remain the landmark standard used. DO-178B establishes software considerations for developers, installers, and users, when aircraft equipment design is implemented using microcomputer techniques. According to RTCA, the purpose of DO-178B is to provide guidelines for the development of airborne systems' software that "performs its intended function with a level of confidence in safety that complies with airworthiness requirements". DO-178B is primarily concerned with the development processes. As a result, certification in DO-178B requires delivery of multiple supporting documents and records. The quantity of items needed for DO-178B certification, and the amount of information that they must contain, is determined by the level of certification being sought. The DO-178B specification does not contain anything magical; it enforces good software development practices and system design processes.

Whilst DO-178B and its clarification notices detail all actions and requirements necessary throughout all stages of the development lifecycle in order to assure the integrity of the software, they do not offer any assessment method to verify that these requirements and actions were met and performed at their corresponding stage before the transition to the following stage in the lifecycle. Hence the objective of the software quality assurance model proposed in this paper.

6. Proposed Safety-Critical Software Development Lifecycle

The lifecycle model proposed by in [29] is based on the experience acquired during the work on the CRIAQ project and on the software development activities described by DO-178B. Aircraft and avionics safety-critical software projects go through different stages that can be grouped into three categories: pre-development, development, and post-development. As illustrated in **Figure 1**, the proposed lifecycle models ten safety-critical software development stages in a flow under those three categories. Those ten stages were either taken directly from DO-178B or inspired from its description of a safety-critical software development process. Therefore, in the description of the stages of the lifecycle, DO-178B is referenced whenever a stage is taken from it.

During Pre-Development, the software project is launched and prepared for development through defining requirements and planning. Development involves all activities taken to build the software product in conformance to all predefined requirements. In Post-Development stages, the software is prepared to be released for use through integration, validation, and certification.

7. SQA Modelling Approach

As discussed earlier, in commercial avionics systems, DO-178B and its European equivalent ED-12B are the main documents used as the means to ensure compliance with regulatory, certification, and safety requirements. DO-178B describes the objectives of each stage of the software lifecycle, the stage activities, and the evidence of compliance required. However, DO-178B does not set any norms or guidelines on when to consider a stage complete and the software ready to move to the next development stage.

In order to achieve the quality challenge, it is of high importance to ensure that each stage in the lifecycle is completed error free and with all requirements met accurately. Therefore, a software quality assurance model is developed in this chapter. It consists of a proposed set of Quality Acceptance Criteria derived by the author. These

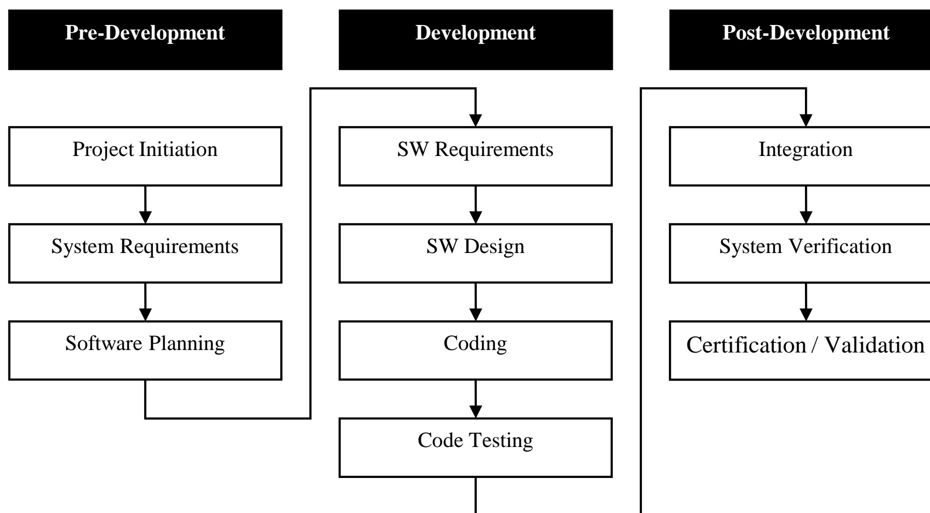

Figure 1. Proposed lifecycle model for safety-critical software.

criteria complete DO-178B by being indicators and rules according to which quality assessment is carried out after each development stage and the final conclusions about software conformity the requirements are done. The proposed set includes four criteria: Completeness, Documentation, Intelligibility, and Independence.

7.1. Quality Acceptance Criteria

A set of four Quality Acceptance Criteria forming the foundation of the SQA model are proposed in this thesis. The criteria were derived by the author from readings and analysis of certification and validation requirements set by regulatory bodies as well as from the experience drawn from the CRIAQ project. The criteria are proposed as an assessment tool to ensure that each stage of the development lifecycle of safety-critical software is consistent with DO-178B requirements. This will prevent iterations within the lifecycle and build quality into the software from as early as the first stage, hence favouring a certifiable product developed within budget, timeline, and in compliance to all requirements.

A successful and certifiable safety-critical software project is a project with all activities, tasks, and objectives fully complete and in conformance with planned requirements and norms. Moreover, all regulatory bodies require accurate documentation of the project development and outputs that are clear and traceable. They also emphasize the independence of personnel conducting tests, verifications, and assessments throughout the development. Therefore, the proposed set of acceptance criteria is made of the following four criteria that embrace all the characteristics of a successful and certifiable safety-critical software development project: Completeness, Documentation, Intelligibility, and Independence.

7.1.1. Completeness

The first quality acceptance criterion is Completeness. Besides errors and bugs, loops in the software lifecycle are also due to unfinished activities, unmet or forgotten requirements, and unachieved stages' specifications and objectives. The purpose of the Completeness criterion is to ensure that a development stage fulfills all its activities while accomplishing all goals and requirements. The aim is assuring that the next stage in the lifecycle can be launched with zero risk of having to go back to an earlier stage. The Completeness criterion requires that the development stage being verified for quality acceptance satisfies the following set of norms of completeness:

1) Specifications

All specifications developed at that stage match accurately and completely all the specifications required for the software project in question.

2) Functional Requirements

All system and software functional requirements are present in the development progress at that stage. If a certain functional requirement cannot be worked on at the current stage, its exclusion should be justified.

3) General Requirements

All general requirements such as standards and safety issues are an integral part of the development stage in question.

4) Objectives and Goals

All objectives and goals of the current stage have been completely met and justified.

7.1.2. Documentation

Documents are an integral part of software development and of the software product (e.g. user manuals). No software can be certified and/or validated without the presentation of sets of documents required by regulatory bodies. For instance, DO-178B requires one or more document to be prepared at each development stage. Documents are also necessary for activities and information flow between stages; coding requires design documents, and design is based on software requirements documents. Therefore, it is important to make sure that all necessary documentation is completed on time; hence, the importance of the Documentation criterion.

The Documentation criterion is satisfied if all necessary and required documents from the development stage in question are completed according to standards, norms, and rules required for the project (e.g. DO-178B documents). All documents in question are to be finalized then audited and accepted by the Quality Assurance member(s) of the Developer's team. It is important to note that Completeness and Documentation are two interrelated criteria that go in parallel with each other.

7.1.3. Intelligibility

The Intelligibility criterion is made of two norms, Clearness and Traceability, which both should be satisfied in order to achieve Intelligibility.

1) Clearness

All documents and all released material are to be clear and understandable to the client and third party experts who did not take part in the development process but are auditing the process. Consequently, Clearness allows faster and more efficient reviews, evaluations, and assessments.

2) Traceability

Each stage should be traceable (linked) to its previous stages. Traceability should also be present between system requirements and software requirements, between low-level requirements and high-level requirements, and between the source code and the low-level requirements. Traceability keeps track of the flow and evolution of all development activities that facilitate the detection of development mistakes and sources of any error, bug, or defect.

7.1.4. Independence

The Independence criterion is related to all verification and audit processes. These processes should be carried out by an expert or group of experts who is/are independent from the developer's team. Verification, audit, and testing cannot be conducted by the same person or group who developed the software. Independence is relative; the verification experts can be a part of the same organization developing the software as they can be from a different organization administratively and/or financially independent. The more the software is safety-critical, the more independence is required between the developer and the verification expert(s).

7.2. The Use of Quality Acceptance Criteria

The proposed set of Quality Acceptance Criteria is designed to be applied after each development stage. However, due to the characteristics of each criterion, not all of them are applied to all of the development stages. **Table 1** summarizes which criteria to apply at each development stage of the proposed lifecycle; a (+) sign means that the criterion has to be applied where a (−) sign means that it does not. A development stage cannot be started before the preceding stage has been found to satisfy all of the applicable criteria in the set.

Development Lifecycle with Quality Acceptance Criteria

Applying the set of Quality Acceptance Criteria results in a change in the development lifecycle proposed in Section 4. As it has been said earlier, each development stage should be checked for applicable criteria before the following stage is started. In case of an error, the developer should restart the stage in question to fix all found problems. **Figure 2** illustrates the development flow when the set of Quality Acceptance Criteria is applied.

Table 1. Acceptance criteria by stage.

	Completeness	Documentation	Intelligibility	Independence
Project Initiation	−	+	+	−
Syst. Requirements	+	+	+	−
SW Planning	+	+	+	−
SW Requirements	+	+	+	−
SW Design	+	+	+	−
Coding	+	+	+	−
Code Testing	+	+	+	+
Integration	+	+	+	+
System Verification	+	+	+	+
Cert./Valid.	−	+	+	(−)[1]

[1]Independence is not required for that stage since no actual certification processes are taking place.

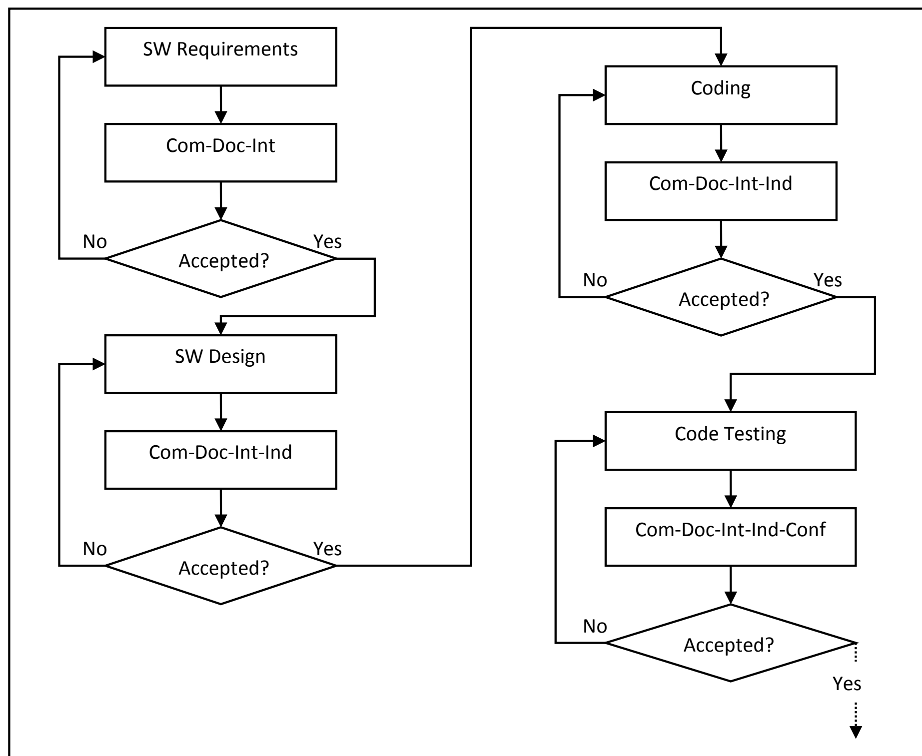

Figure 2. Quality Acceptance Criteria applied to the development lifecycle.

Carrying out an assessment of the set of Quality Acceptance Criteria as stated in the previous paragraph will ensure to the developer that the development of the software is on the right track and that it will be completed right from the first development loop. However, this assessment after each stage has its drawbacks as well. It introduces an extra process right after each development stage which can be time and resource consuming; safety-critical software development project tend to go over schedule. In addition, it introduces a new loop to each stage as illustrated in **Figure 2**; this can be seen as breaking the Plan-Build-Release loop into four smaller loops.

In order to overcome these problems, the Quality Acceptance Criteria assessment should be applied concurrently with the development stage. While developers are working on the software development, a team of Quality Assurance works on evaluating the progress in the current stage with respect to the set of Quality Acceptance

Criteria. Figure 3 illustrates the proposed development flow when concurrent assessment is applied.

The advantage of this concurrent approach is time saving and the avoidance of having regressive assessment loops within each stage.

Table 2 summarizes the interaction between the lifecycle and the SQA acceptance criteria from the initiation of a safety-critical software project until the end. For each development stage the table shows which criteria apply, what output documents are required according to DO-178B and/or according to the SQA model designed in this paper.

8. Discussion

When applied to safety-critical software development projects, the proposed SQA model ensures that the end-product is of high-quality by keeping track of all activities throughout the development and making sure that nothing is missed. It assists the developers in building quality into their product right from the initiation of the project.

By assessing each development stage for four acceptance criteria before allowing the next stage to start, the proposed SQA model does not tolerate any error/defect to be transmitted or inherited from a development stage to a subsequent one. This strictness is the main difference between the proposed model and other models (such as CMM and SPICE) which are leaner on defects and errors.

From the managerial side, the advantage of the proposed SQA model is that it starts by requiring all activities and documents to be well defined at the beginning and then follows rigorously the progress of each activity and document until its completion while ensuring that none is left undone. The SQA model also stresses on the traceability and clearness of all the work performed throughout the development. Moreover it emphasizes on independency of verifications and assessments when they are required.

For software in aerospace applications DO-178B is the standard used for developing a certifiable product. DO-178B is at the heart of the framework of the proposed SQA model which when applied assists developers to perform an activity or output a document at the right time as required by DO-178B. In addition, the proposed model complements DO-178B by offering a method to assess all development lifecycle stages with respect to their corresponding DO-178B activities, actions, and requirements.

The proposed work in this paper was based on the development of a real-life project. This development followed only DO-178B guidelines. DO-178B does not offer any means to verify that all of its requirements are being met, and several activities of the CRIAQ project fell short. This inspired the proposed SQA model. For instance one of the software components developed had to be redesigned and rebuilt several times because of requirements that have not been met and because of requirements that just have been forgotten. In the proposed model, all requirements should be clearly defined and each stage is assessed on whether all of those requirements are in the development or not.

From the documentation side, some documents were still incomplete as the project progressed through time. This was mainly due to the absence of a definition and description of all documents required and of a time limit for their completeness. In the proposed SQA model, all documents should be clearly described at the beginning of each stage and given an importance level. According to the documentation criterion acceptance rules this importance level sets a certain time limit for each document; thus, at any time in the development, activities cannot be undertaken unless all documents due at that time are complete. In addition, several software components and documents were left without any configuration what fails to meet traceability requirements which are essential in DO-178B and the proposed model.

At the time of completion of this research, none of the developed components were verified for operational functions and none of the tolerances were properly defined. If the proposed SQA model was applied to the CRIAQ

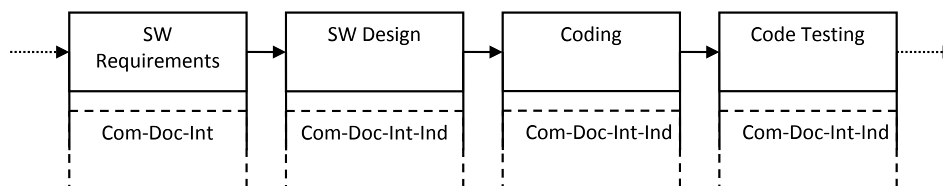

Figure 3. Concurrent assessment for Quality Acceptance Criteria.

Table 2. Lifecycle and SQA.

Stage	Acceptance Criteria				Document Output	
	Completeness	Documentation	Intelligibility	Independence	DO-178B	Present work
Project Initiation		√	√			Project Contract
System Requirements	√	√	√			Definition of requirements
SW Planning	√	√	√		SW development plan SW verification plan SCM plan SQA plan Plan for SW aspects of certification	
SW requirements	√	√	√			SW requirements document (High-level)
SW Design	√	√	√		Design Description	SW requirements document (Low-level)
Coding	√	√	√		Source code	
Code Testing	√	√	√	√		Test results and recommendations
Integration	√	√	√	√		Integration Record
System Verification	√	√	√	√	Verification results document	
Certification/ Validation		√	√			Checklist of documents required for certification

project all those components would have been verified by now since the model does not allow the progress of the development without a successful verification of all work done.

9. Conclusions

Software applications in which failure may result in possible catastrophic consequences on human life are classified as safety-critical. These applications are widely used in a variety of fields and systems such as airborne systems, nuclear reactors' control, medical monitoring and diagnostic equipment, and electronic switching of passenger railways. Unfortunately the world has seen several accidents and tragedies caused by failures of safety-critical software.

This paper looks into safety-critical software embedded in systems in the aerospace field. Those systems can either be airborne or ground-based systems (*i.e*. aircraft simulators). The following contributions have been made: 1) a demonstration that methods used for quality control and management of software and products in general cannot be applied effectively to safety-critical software products, and 2) a proposed of an SQA model that builds quality into safety-critical software throughout its development.

Quality control techniques as used in general rely on analyzing data and statistics collected from testing samples or all products produced. In safety-critical software the tolerance on defects is extremely small; only products with a defective rate smaller than 10^{-8} can be accepted which makes testing safety-critical software for errors infeasible due to the time it would take to detect an error. Hence quality should be built into safety-critical software and managed throughout the development.

This paper also proposes an SQA model designed for safety-critical software embedded in systems in the aerospace field. The model was designed by the author based on the DO-178B standard and experience and lessons learned while working as an SQA engineer on a project to develop a software components for an aircraft simulator that will be used as a Test-Bed for Flight Management Systems. While the proposed model can be implemented to all types of safety-critical software projects, it was mainly designed for "make-to-order" development projects which in general are very specific and clear.

While developers are working on their activities following the proposed lifecycle, an SQA team is responsible of implementing concurrently the proposed SQA model during all stages. The SQA team should be independent from the developers; however, excellent communication between both of them is the key for success.

The work proposed in this paper does not replace DO-178B; it however complements it by offering methods ensuring its implementation appropriately. Several developers in the industry have complained that DO-178B is not clear enough and does not include any method to verify that it is being accurately applied during the development. The lifecycle proposed in this thesis offers a clear flow of all the activities required during development, while the proposed SQA model checks each development stage for its compliance with DO-178B and all other requirements set by the developers.

Safety-critical software is an emerging field of research. We proposed a new model and approach to safety-critical software quality assurance. Like all new contributions in a field that is on a continuous technological advancement, it opens new doors for future research, expansion, reviews, and adjustments. The following presents some ideas for future work:

1) Develop metrics to measure during a development project the implementation of the proposed SQA model and the "response" of the project stages with the four criteria.

2) Develop a method that can be implemented in parallel with the proposed SQA model aimed at managing changes that might occur during the development process.

3) Further examine the verification of safety-critical software and develop a set of protocols for the verification processes.

Finally, the proposed work should be reviewed when a new version or notice of DO-178B is released or when a new standard is adopted. It should be also reviewed when new research results are published. It is recommended to implement the proposed work during the second phase of the CRIAQ project, which involves developments of further software components of the DTB. This implementation allows the investigation of possible changes in the lifecycle and in the SQA model.

References

[1] Kornecki, A., Zalewski, J., Ehrenberger, W., Saglietti, F. and Górski J. (2003) Safety of Computer Control Systems: Challenges and Results in Software Development. *Annual Reviews in Control*, **27**, 23-37. http://dx.doi.org/10.1016/S1367-5788(03)00004-X

[2] Jackson, D., Thomas, M. and Millet, L. (2007) Software for Dependable Systems, Sufficient Evidence. Academic Press,

Washington DC.

[3] Kornecki, A.J. and Zalewski, J. (2005) Experimental Evaluation of Software Development Tools for Safety-Critical Real-Time Systems. *Innovations System Software Engineering*, **1**, 176-188. http://dx.doi.org/10.1007/s11334-005-0013-1

[4] Sakugawa, B., Cury, E. and Yano, E. (2005) Airborne Software Concerns in Civil Aviation Certification. *Dependable Computing*, **3747**, 52-60. http://dx.doi.org/10.1007/11572329_7

[5] Wils, A., Van Baelen, S., Holvoet, T. and De Vlaminck, K. (2006) Agility in the Avionic Software World. *Extreme Programming and Agile Processes in Software Engineering*, **4044**, 123-132.

[6] Rauch, N., Kuhn, E. and Friedrich, H. (2008) Index-Based Process and Software Quality Control in Agile Development Projects. http://goo.gl/RxNXJ

[7] Leveson, N.G., Cha, S.S. and Shimeall, T.J. (1991) Safety Verification of Ada Programs Using Software Fault Trees. *IEEE Software*, **8**, 48-59. http://dx.doi.org/10.1109/52.300036

[8] Redmill, F., Chudleigh, M. and Catmur, J. (1999) System Safety: HAZOP and Software HAZOP. John Wiley & Sons, New York.

[9] Maier, T. (1995) FMEA and FTA to Support Safety Design of Embedded Software in Safety-Critical Systems. *Proceedings of the ENCRESS Conference on Safety and Reliability of Software Based Systems*, Bruges, 12-15 September 1995.

[10] Cichocki, T. and Górski J. (2000) Failure Mode and Effect Analysis for Safety Critical Systems with Software Components. *Computer Safety, Reliability and Security*, *Lecture Notes in Computer Science*, **1943**, 382-394.

[11] Elliott, L., Mojdehrakhsh, R., Tsai, W.T. and Kirani, S. (1994) Retrofitting Software Safety in an Implantable Medical Device. *IEEE Software*, **11**, 41-50. http://dx.doi.org/10.1109/52.300036

[12] Wichmann, B. (1999) Guidance for the Adoption of Tools for Use in Safety Related Software Development. Draft Report, British Computer Society, London.

[13] Bowen, J.P., Vilkomir, S.A. and Kapoor, K. (2003) Tolerance of Control-Flow Testing Criteria. *Annual International Computer Software and Applications Conference* (*COMPSAC* 2003), Dallas, November 2003, 182-187.

[14] Fisher, M.J. and Cooper, J.D. (1979) Software Quality Management, Petrocelli Books Inc., Princeton.

[15] Reifer, D.J. (1985) State of the Art in Software Quality Management. Reifer Consultants, Torrance.

[16] Rubey, R.J. and Hartwick, R.D. (1968) Quantitative Measurement of Program Quality. *ACM National Conference*, Las Vegas, 27-29 August 1968, 671-677.

[17] Wulf, W.A. (1973) Programming Methodology. *Proceedings of a Symposium on the High Cost of Software*, Stanford Research Institute, Menlo Park.

[18] Fagan, M. (1976) Design and Code Inspections to Reduce Errors in Software Development. *IBM Systems Journal*, **15**, 182-211.

[19] Butler, R.W. and Finelli, G.B. (1993) Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, **19**, 3-12. http://dx.doi.org/10.1109/32.210303

[20] Nagel, P.M. and Skrivan, J.A. (1982) Software Reliability: Repetitive Run Experimentation and Modeling. NASA Contractor Rep. 165836.

[21] Littlewood, B. (1989) Predicting Software Reliability. Philosophical Transactions of the Royal Society, London, 513-526.

[22] ANSI/IEEE (1981) IEEE Standard for Software Quality Assurance Plans. ANSI/IEEE Std 730-1981.

[23] Kumar, C.A. (1994) Excellence in Software Quality. India Infotech Standards, New Delhi.

[24] Manns, T. and Coleman, M. (1998) Software Quality Assurance. Macmillan Education, London.

[25] Johnson, J. (1995) Chaos: The Dollar Drain of IT Project Failures. *Application Development Trends*, **2**, 41-47.

[26] Genuchten, M. (1991) Why Is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Transactions on Software Engineering*, **17**, 582-590. http://dx.doi.org/10.1109/32.210303

[27] Galin, D. (2004) Software Quality Assurance, From Theory to Implementation. Pearson-Addison Wesley, New York.

[28] O'Regan, G. (2002) A Practical Approach to Software Quality. Springer-Verlag, New York.

[29] ElSabbagh, H. (2006) A Quality Assurance Model for Airborne Safety-Critical Software. M.Sc. Thesis, Department of Mechanical and Industrial Engineering, Concordia University, Montreal.