Software Safety: A User’s Practical Perspective

William R. Dunn; University of Southern Colorado at NASA Ames Research Center; Moffett Field

Lloyd D. Corliss; NASA Ames Research Center; Moffett Field

Key Words: Software safety, Software reliability, Software error, Real-time software, Reliability analysis, Safety.

Abstract

This paper addresses the theme of the 1990 Reliability and Maintainability Symposium—Product Assurance Progress Report—by examining recent software safety assurance practice on several real-time, safety-critical operational projects at NASA Ames Research Center. Software safety definitions and concepts are discussed. It is shown that, to be safe, software must, for all practical purposes, be error-free. Case histories cover software developments on two digital flight control systems and two ground facility systems. For each case history, the overall system and software organization and function are described and the software safety issues and their resolution are presented. The effectiveness of safety assurance methods discussed. Methods include conventional life cycle practices, verification and validation testing, software safety analysis, and formal design methods for realizing safe software. Three conclusions are drawn: 1) at present, a practical technology for assuring that software is safe does not exist, 2) it is unlikely that a set of general-purpose analytical techniques can be developed for proving that software is safe, and 3) successful software safety assurance practice will have to take into account the detailed design processes employed in the software development and show that the software will execute correctly under all possible conditions.

Introduction

Software safety has long been a priority at NASA. However, NASA now faces a new, major challenge in the practice of software safety assurance. We are now being swamped with low-cost commercial computer products for use not only in our traditional airborne applications but also in hazardous ground-based applications, many of which were formerly handled by analog technology. Also becoming available at low cost are many software products such as real-time operating systems, compilers, and control application languages. This sudden flood of computer-based products into the Ames environment has increased the number of software-critical applications, the complexity of the systems, and the number of applications that use conventional data processing practices and hence were not built against safety requirements. Unfortunately, in spite of the dramatic increase in workload, software assurance resources have remained static.

In this paper, we discuss recent software safety assurance philosophy and practice at Ames Research Center. The four major Ames systems discussed here include projects covering both aerospace and ground facility applications, and all employ safety-critical software.

Software Safety—Basic Definitions

Software safety assurance practice, viewed as a formal discipline, is a relatively new activity. As a result, definitions, concepts, and interpretations vary widely across the literature (Ref. 1). The following define basic terms and concepts as they are used in this paper.

Our primary software safety concern is the probability that a software error or "bug" might surface in real-time control and cause a catastrophic event. For purposes here, "software errors" are those made in translating software requirements, that is, errors associated

with software design and coding but not with flaws or omissions in software requirements. More generally, we do not address system and human errors even though they, not software errors, represent the majority of problems that show up in real-life systems.

"Probability" is treated in the strict quantitative sense. For example, in experimental manned flight work at Ames, a maximum target failure rate of \(10^{-6}\) catastrophic failures per hour is often budgeted for the entire flight control system. For the ground facilities discussed in the next section, a maximum of 0.25 catastrophic control failures over the 40-year life of the system is specified, corresponding to a mean failure rate also on the order of \(10^{-6}\) catastrophic failures per hour. These figures correspond to all elements of the system, not just software, and constitute an upper bound on the allowable catastrophic failure rate of control software.

The term "catastrophic" covers not only death and/or major property loss or damage but any event resulting in personnel injury and/or significant property damage as defined in Ref. 2. Finally, "software safety technology" embraces both classical methods and newer ideas in the literature.

Software Safety Case Histories

The following case histories cover four real-time control systems currently or recently under development at Ames and focus on how the software safety issues were resolved.

Case History #1—X-Wing Flight Control System

The X-Wing concept marries the efficiencies of a helicopter and a fixed wing aircraft through the use of a four-bladed wing/rotor that can be rotated or stopped in flight (Fig. 1a). Control requirements for the X-Wing rotor are complex. Starting and stopping the rotor requires use of a clutch, brake, and index/locking subsystem connecting the main rotor shaft to the aircraft gas turbine power...
To provide both fixed and rotary wing aerodynamic lift with the same body, symmetrical rotor blades are used with "circulation control," a process of controlling lift with blowing (Refs. 3-5).

To safely execute all of the control functions, the X-Wing relies solely on a quadruply-redundant, digital, fly-by-wire control system, at the heart of which are four flight control computers (FCC) (Fig. 1b). The four identical channels use three software voting planes and provide a two fail/operate capability.

Software in this centralized system controls all of the X-Wing subsystems and includes a significant amount of code for redundancy management.

The primary software safety concern with the X-Wing flight control system was that an undetected software error might surface during flight, causing loss of control resulting in loss or damage of the aircraft and/or death or injury to the crew. In the original X-Wing design, the contractor maintained that hardware redundancy alone would adequately meet a $10^{-6}$ errors/hr reliability requirement for safe flight, based on the idea that ground verification and validation (V&V) testing (Ref. 6) would prove the software correct and therefore safe for flight. The underlying rationale was that program execution would be deterministic and predictive so all software paths could be exercised and the significant bugs purged (Refs. 7 and 8).

Proponents of this thinking could cite the successful V&V track record of F-8, F-16, and (contractor's) F-20 digital fly-by-wire software developments. In the minds of others, however, this thinking conflicted with the longstanding belief that testing can reveal bugs in software but not prove their absence. Hence, as a part of our assurance activities, we examined both positions.

We looked first at experience on other flight systems. This study (Ref. 9) showed that a software package of X-Wing's size (approximately 120K lines of assembly source code) could be expected to enter initial flight testing with a residual catastrophic error rate of $10^{-2}$ to $10^{-4}$ (catastrophic) errors/hr. This figure is far in excess of the allowable $10^{-6}$ errors/hr budget for the entire control system. As a result, we implemented backup control software (BUCS) in each FCC channel similar to that discussed in Ref. 10. BUCS could be invoked at will by the pilot (via hardware reset) and would provide the necessary primitive functions to land the aircraft safely. (The reliability argument was made that joint probability of loss of both primary and backup would be far below that established for safe flight (Ref. 9).

Subsequently, we also examined the viability of the contractor's original assertion that use of finite code execution paths coupled with thorough testing could yield a safe program. Using customized computer routines we attempted to recreate the real-time data flow and control flow. Our analysis effort bogged down since the software designers had employed an efficient but exceedingly complex priority-tasking executive, distributed across the four processors, which resulted in an actual program flow that was unpredictable from frame to frame. This executive caused an excessive data latency which resulted in a compressor control instability which was not discovered until closed-loop testing.

Case History #2 - VSRA Flight Control System

The NASA Vertical Short-takeoff-landing Research Airplane (VSRA) is a modified Harrier fighter used at Ames in advanced guidance and control research (Fig. 2a). One of the principal modifications is an advanced flight control research computer system which interfaces the pilot controls to the aircraft servos. The computer system hosts experimental control algorithms designed to assist pilots in safely landing on a sharply pitching surface, such as those found on a small ship in high-sea states. The VSRA computer system is superimposed on the standard aircraft control system, as shown in Fig. 2b. Dual, parallel computers are used; this choice grew out of earlier simulation work (Ref. 11) which showed that control system failures could be tolerated during the critical, terminal landing phase provided they were automatically detected and announced to the pilot, giving him a chance to abort the landing. It was determined that dual sensors and computers would guarantee that random hardware faults would be fully detectable. By contrast, it was determined that a simplex computer architecture with self-test was inadequate, and a triplex architecture providing fail-operational performance like X-Wing would be an overkill for this experimental aircraft.

As indicated, the computer system augments the manual flight control system and hence, unlike the X-Wing, the aircraft can be flown satisfactorily without the computer. However, a primary software safety issue does exist in that a software error might surface during ship landing and go undetected, leading to an undesirable control response and possible collision with the vessel.
Like X-Wing, VSRA developers started off by designing software which could be made safe using deterministic paths in the flight control algorithms, and which could then be proven safe through hardware-in-the-loop V&V testing. The potential for realizing a safe, assured software design came when a structured design methodology (Ref. 12) and tools for translating VSRA software requirements into well-delineated, codable units, were adopted. However, several problems resulted. First, the input data generated from dual inertial navigation units was asynchronous with the flight computer sampling rate, meaning that exact data flow would be unpredictable frame-to-frame. Second, a decision to use a high-level compiler in developing the code and to employ a real-time operating system kernel for executing input/output and control algorithms created a software safety problem; there was no known practical way to guarantee that the compiled and installed machine-level code would be free from the required one-in-million chance of having a data sensitive, catastrophic bug. Finally, to meet real-time computational throughput, program structure would likely have to be altered, leading to reduced predictability of control function.

Like X-Wing, the answers to these software safety problems lay in the use of redundant software. A simple algorithm was devised (Ref. 11) to monitor aircraft trajectory during landing, thus detecting any incorrect trajectory. This simple software will be coded in assembly language and, to provide protection against random hardware failures, located in each of the two flight computers. This software monitor will then be executed on each hardware-generated frame pulse guaranteeing that this software will run irrespective of failures in primary software.

**Case History #3 – High-Pressure Air Distribution Control System**

A high-pressure air distribution system at Ames supplies 3000 psi of air to several high-energy wind tunnels located throughout the research center. The air originates in a remote, high-pressure “tank farm,” isolated for safety, and is distributed in an underground piping network around the center. The function of the High Pressure Air Distribution (HIPAD) Control System (Fig. 3) is to route air from the source to one or more user facilities and to provide real-time monitoring of pressures throughout the network.

The control system has three major elements: 1) a Master Station Computer system (centrally located in a main compressor building) consisting of a pair of computers each utilizing DEC MicroPDP 11/73 microcomputers interfaced to a conventional disk drive, CRT terminal, and printer peripheral units, 2) ten remote, simplex Distributed Control Units (DCUs) which interface to valve actuators and to pressure, temperature, flow, and valve position sensors, and 3) a dual Fiber-optic Communications Network linking the Master Station and DCUs.

Operators at the Master Station site command pneumatic valve positions via the keyboard or with a CRT light pen. The commands are transmitted via the fiber optics to the DCUs, which in turn electrically actuate the pneumatic valves. In a reverse flow, the DCUs at remote sites collect and transmit pressure, temperature, flow, and valve position data to the Master Station for processing. Pressure sensor data can be displayed on CRTs or called up in tabular form. Pressure sensor values are monitored, and when they exceed a set maximum an alarm is activated at the Master Station. A primary safety concern was that operator error, hardware failure, or a software bug might accidentally open a system valve while the user had his corresponding valve open.

Functional software requirements for this system were simple. Fortran code written against these requirements was correspondingly simple enough that rudimentary walkthroughs and demonstration testing would have sufficed to prove that this application software was safe. Unfortunately, the application code was buried in a large, undocumented custom operating system which the controls vendor had developed to easily meet any and all potential user applications. The resulting program, totalling some million lines of code, was unanalyzable.

Hence a software safety issue was still present. It was assumed that software could inadvertently open any system valve at any time, so each system-user interface was equipped with a relay logic interlock which would physically prevent the opening of the control system valve into the open user valve. However, the user valve would have to be closed before receiving air service. This fix accommodates not only software errors, but human errors and random hardware failures as well.

**Case History #4 – Wind Tunnel Control System**

Ames is currently modernizing two major wind tunnel facilities to enhance productivity and improve safety. This includes installing computer controls for operating the tunnels and collecting and processing experimental data. The basic control architecture to be used is a distributed system as shown in Fig. 4. A centralized work station allows operators to program coordinated tunnel functions which are collectively carried out by three distributed computation units that perform three basic functions: 1) tunnel condition control, 2) model control, and 3) auxiliary systems control. The tunnel condition controller regulates pressure, temperature, and wind velocity in the tunnel by controlling the compressor speed, the tunnel air source pressure, and the air cooling subsystem. The model controller establishes the exact position and attitude of the scale test model under investigation. Auxiliary controls control various tunnel functions including compressor lubrication and...
hydraulic power subsystems for model actuation. A major software safety issue lies with the model controller, in which software errors might lead to unstable model control, leading to model destruction and/or damage to the tunnel. Worst case failure effects could include debris penetration of protective screens into the tunnel fans or damage to the tunnel wall.

Unlike our previous applications, control software in this system will use application packages designed specifically for control engineers. With these utilities, engineers do not program; they implement control laws and logic algorithms by drawing analog-like control schematics and relay ladder logic diagrams with a graphics input device. Translation of these algorithms is completely automatic. For a one-of-a-kind installation like the tunnel control, an application-oriented language is effective because one goes directly from requirements to object code without the cost, schedule, and reliability penalties of conventional software design. The safety problems, however, are the same as those for the high-pressure air distribution system. That is, although a high level of confidence for the integrity of control laws and logic might be possible, potential hazards in the full-up compiled code are not known. A potential, cost-effective workaround is to use an independent microcontroller exclusively dedicated to monitoring the operational envelope of the model support system. In the event of any operator mistake, hardware fault, or software error leading to unsafe model motion, the system would detect the onset of errant motion and disconnect the associated control.

Discussion

In our work, software safety is synonymous with software ultra-reliability. A $10^{-6}$ errors/hr reliability requirement on software is expected in any commercial, industrial, or medical application where property and life are at risk. Given this requirement, we consider any software system that can produce dangerous effects, and which has any amount of complexity, to be unsafe. As the case histories illustrate, we have found no way to prove that a given software package is safe, so we use some kind of system workaround or fix.
Much of our work in software development and many recent ideas in the literature are aimed at software safety. We now look at these in the context of our case histories, noting that many of our conclusions have also been reached in Ref. 13.

Software Life Cycle Assurance

Conventional life cycle assurance standards (e.g., DoD-Std-2167A (Ref. 13) and NASA’s SMAP (Ref. 14)) set up effective, systematic procedures for minimizing software errors, uncovering and removing them during system development. Code inspections and testing uncover errors; elaborate techniques correct software without introducing new errors. In this process, large numbers of errors are initially encountered. With continued testing and removal of bugs, software becomes "stable" (i.e., many hours of testing go by before a bug is encountered). As testing and bug removal progresses, software becomes more reliable. Given weeks or months of error-free testing, the subjective feeling is that the software is, finally, safe. The impression is illusory. For example, if a software package undergoes a solid month of error-free testing since the last serious bug, it will have demonstrated a reliability no better than \(10^{-3}\) errors/hr, not \(10^{-3}\) errors/hr.

Software Safety Analysis

Safely analysis methods, recently of great interest (Ref. 16), appear to have the same Achilles’ heel as testing—they can reveal error presence but not prove error absence. We believe, therefore, that analysis methods (like testing) can make software safer but not necessarily safe. To assure that a given software package is safe, we need to prove that it is error-free. With some thirty years of computer science behind us, we still do not know how to do this for software of the complexity of our case histories. For that matter we do not know whether it can be done at all. We suspect that if analytical methods have a future role in the safety assurance of software, they will appear as a part of a software design process which inherently produces safe software.

Safe Software Design

As the case histories illustrate, we start, in software design, hoping to realize bug-free software but are ultimately defeated. We find three factors at work here.

Firstly, no matter how simple the control problem, software quickly becomes extremely complex because the systems are real-time applications. We start with a clean, well-structured program and find that it will not meet throughput maximums. To improve speed, we partition modules, rearrange control flow, and perhaps go to multiprocessors, so that the resulting program becomes an intricate mess whose behavior is unpredictable. According to Stankovic (Ref. 17) we are not alone with this problem.

Secondly, we are no longer working close to the machine level where software can be inspected and its behavior predicted. To gain practical access to modern CPUs we are now involved with compilers, operating systems, and other translators that make the functioning code invisible. Tools and techniques, many developed by NASA, abound for analyzing source code and object code, but to our knowledge, there is no general-purpose tool available for proving correctness of an arbitrary real-time program.

Finally, the major problem is one of schedules and available resources. We will always be behind schedule and be the last ones to finish work. If existing computer science holds the keys to getting safe software, we never get the chance in the real life environment to find them.

Conclusions

Software safety goes way beyond software reliability for, to be safe, software must be essentially perfect. As the case histories illustrate, our present, practical software safety assurance practice is primitive. We assume that all critical software is inherently unsafe, so we require an independent, simple, predictable element which will protect the system and its environment if and when software goes bad. We invariably seek a system solution which is unique to the problem; it is frequently not a software solution. At present there is no practical technology to assure that software is safe.

Contemporary thinking and practice associated with "software safety" appear primarily to address software reliability improvement. To make software safe as defined in this paper, we will have to see reliability improvement methods which, in a reliability context, will give an unsafe \(10^{-3}\) errors/hr software package a \(10^{-7}\) errors/hr failure rate. It is unrealistic to expect that such methods could evolve in the near future, if indeed ever. We do not believe that life cycle practices, testing, and analysis techniques will improve to the point that an arbitrary software package can be made safe solely through their repeated application.

It does seem reasonable that softw. software can be achieved through design. However, this belief centers around a specialized class of software problem—the realization of deterministic real-time control system requirements. This kind of problem sets up readily in finite state machine representation where—and here is one possible key to software safety assurance—global algorithmic behavior is fully visible and predictable. Getting this representation safely translated in the practical world of real-time computer products is, unfortunately, an unsolved problem (Ref. 16). To be effective, software development safety assurance practice will have to take into account the actual design processes employed in the software development and show that the resulting product will execute in accordance with the requirements under all possible conditions.

When all is said and done, it appears that in a software-intensive system in which safety is a major issue, a suitable backup to the software is still required. Current technologies do not yet assure software safety.

Acknowledgement

This paper was supported in part by NASA Grants NCC 2-276 and NCC 2-669.

References


Biographies

William R. Dunn
University of Southern Colorado at NASA Ames Research Center MS 237-7
Moffett Field, CA 94035 USA

W. R. Dunn is a Faculty Research Associate at the University of Southern Colorado who is on a research Cooperative Agreement at NASA Ames Research Center. B.S. Physics, University of California at Berkeley; M.S.E.E. and Ph. D., University of Santa Clara. He is currently active in advanced research in ultrareliable systems design at NASA Ames where he also serves as an independent design consultant and safety review board member on advanced safety-critical computer control applications. Past experience includes Director of R&D, Century Digital Systems, Assistant Professor, University of Santa Clara; and Research Engineer, Lockheed Aircraft Corporation. Dr. Dunn has published 27 technical papers and is co-author of the short text, Designing with Microprocessors. He is a member of the Tau Beta Pi and Eta Kappa Nu engineering honorary societies.

Lloyd D. Corliss
National Aeronautics & Space Administration
Ames Research Center MS 237-7
Moffett Field, CA 94035 USA

L. D. Corliss is a NASA Flight Controls Engineer presently on the staff of the Ames Military Technology Office. B.S.E.E., 1963, M.S.E.E., 1966, Michigan State University; Control Systems; Graduate Studies Aeronautics, Stanford University. Mr. Corliss was formerly with U.S. Army Research and Technology Labs at Moffett Field, California as program engineer on the development of several VTOL and helicopter digital flight control system designs and as project engineer for numerous flight test and simulation studies of VTOL and helicopter handling qualities. Mr. Corliss has authored over 20 technical papers and reports in the area of controls, simulation, and handling qualities. He is a professional engineer in the state of California and is a member of the American Helicopter Society.