The Complexity Challenge in Modern Avionics Software

Lui Sha, lrs@cs.uiuc.edu
August 14, 2006

Abstract: Large and complex avionics software has emerged as a new source of safety hazards in practice. It is impractical to exhaustively test large and complex avionics software. Nor is it practical to formally verify them because the complexity of verifying temporal logic specification is exponential. This position paper argues for a complexity control approach at the architecture level and integrates it with formal methods to verify the reduced complexity critical core and the reduced complexity interactions with the rest of the system.

1.0 Introduction

FAA database indicates that commercial avionics have achieved a very good accident rate when measured in accidents per flight hour. This rate has been fairly stable over the past 10 years. However there are two complimentary trends that might require significant changes. There is a huge increase in the number of airplane hours flown. Many of the previous electro-mechanical systems are being incorporated into software, and the existing functions are being asked to perform more functions. This exponential increase in the amount of software is taxing the existing ability to provide dependability assurances.

Safety critical software is required to be certified by DO-178B. Historically, this certification process has been highly effective from the viewpoint that there is yet a fatal accident to be attributed to software failures. However, as the complexity of modern avionics software increases[1], the effectiveness of DO-178B has been challenged. In manned flights, the actions taken by the software, e.g., the auto-pilot, are supervised by the pilot. As software takes on more responsibility, the supervision becomes more difficult, and pilots are less able to compensate for unexpected software behavior. For example, as reported by Wall Street Journal in May 30, 2006,

> "As a Malaysia Airlines jetliner cruised from Perth, Australia, to Kuala Lumpur, Malaysia, one evening last August, it suddenly took on a mind of its own and zoomed 3,000 feet upward. The captain disconnected the autopilot and pointed the Boeing 777's nose down to avoid stalling, but was jerked into a steep dive. He throttled back sharply on both engines, trying to slow the plane. Instead, the jet raced into another climb. The crew eventually regained control and manually flew their 177 passengers safely back to Australia. Investigators quickly discovered the reason for the plane's roller-coaster ride 38,000 feet above the Indian Ocean. A defective software program had provided incorrect data about the aircraft's speed and acceleration, confusing flight computers. The computers had also failed, at first, to respond to the pilot's commands."

Such incidents, while still rare, are clearly safety hazards. The FAA's emergency airworthiness directive (AD 2005-18-51) regarding this safety incident, notes, "*These anomalies could result in high pilot workload, deviation from the intended flight path, and possible loss of control of the airplane*."

As another example, during the development of the air Traffic Alert/Collision Avoidance System (TCAS) software, "*version 6.00 was the original software for TCAS II. When using this software, some very interesting problems occurred. False conflict alerts were being triggered by transponders on ships and bridges. Additionally, parallel final approach courses less than 5000 feet apart were causing false alerts. It has even been reported that a pilot's own aircraft can cause a false alarm. In this situation the pilot found himself trying to outmaneuver himself*"[2] A study of the airworthiness directives (ADs) and service difficulty reports (SDRs) for large aircraft (1984-1994) revealed 33 avionics ADs, 13 of which were software related.[3]

Software failures have emerged as a new source of safety hazards. We must take such warnings seriously before the ever increasing complexity of avionics software causes fatalities.

---

[1] The latest jets have over 5 million lines of code, compared to about 1 million lines of code in older aircraft. "Incidents Prompt New Scrutiny Of Airplane Software Glitches", May 30, 2006, the Wall Street Journal.
[2] http://www.allstar.fiu.edu/aero/TCAS.htm
[3] Avionics Software Occurrence Rate, http://doi.ieeecomputersociety.org/10.1109/ISSRE.1996.558695

2.0 The Complexity Challenge

The increasing incident rate in avionics software is the manifestation of building modern avionics systems with a complexity higher than what can be handled by existing technological infrastructure. In a March 2004 GAO report to congress, it reports that "*in the last 40 years, functionality provided by software for aircraft, for example, has increased from about 10 percent in the early 1960s for the F-4 to 80 percent for theF/A-22*", and that "*DOD's software-intensive weapon system acquisitions remain plagued by cost overruns, schedule delays, and failure to meet performance goals*."[4] It also reports that in 2003 DoD spent 8 billion (40%) of its software budget in fixing bugs.

Technically, the rapid advancement of processor and networking speed has led to extensive sharing of processor and networks by multiple applications which had been previously federated. All Computations and data streams with different criticality and time constants now share the processors, and network links, the software drivers, the OS, and quite often the middleware that mediates the traffics. The hardware safety barriers in mission systems have been rapidly replaced by a combination of hardware and software based isolation mechanisms. The detection and elimination of errors in software based isolation architecture has been a challenge, especially when middleware is involved. Furthermore, the uniformity of modern processors and networks encourages the practice of dynamic reconfiguration to reduce hardware redundancy.

Under the old federated system architecture, redundant hardware is needed for each dedicated functional box whenever a high degree of reliability is required. This is wasteful in terms of hardware spares but makes fault and failure isolation simple. We now only need a few spares because processors and network links are interchangeable. Complex and powerful middleware also makes physical location transparent to the applications. However, dynamic reconfiguration increases system complexity. And the detection and elimination of errors in the isolation architecture becomes even more daunting when dynamic reconfiguration algorithms are involved. Errors in the reconfiguring actions have become a source of spreading an isolated failure to a system wide failure.

While more efficient in terms of weight and power than federated systems, the new avionics architecture now must rely on complex hardware and software designs to achieve the isolation and protection from faults and failures of components that share common processing and communication resources. In addition, the trend in human factors is to reduce information overload by determining the relevant information needed to be displayed at any specific phase of flight. While this frees up pilot bandwidth by eliminated the need to mentally fuse information from many displays to obtain a model of situational awareness, it greatly increases the complexity of the applications and of the system design as well.

Indeed, modern integrated avionics are facing formidable dependability challenge that cannot be taken lightly, because the three pillars of dependability are: redundancy, isolation, hazard and fault tree analysis. We now have reduced redundancy, hard to verify isolation, and much more complex interactions for hazard and fault tree analysis.

3.0 Using Simplicity to Control Complexity

The challenge of the complexity of modern avionics must be taken seriously. For the most part, avionics software is real time software whose behaviors can be modeled as timed system states. They can be specified and analyzed by temporal logics. Unfortunately, temporal logics that can reason about time constrained behaviors such as MTL has exponential complexity and more expressive ones such as RTL was shown to be undecidable. That is, unless we can keep avionics software sufficiently simple, it is either impossible or impractical to verify formally. Nor can we exhaustively test them.

The complexity of modern avionics system makes it impossible to establish that the avionics software as a whole is 100% error free.

Fortunately, all features are not equal. Nor do we have to follow the current integrated avionics architecture practice that allows for extremely complex resource sharing and interaction. First, we note that a very small

[4] http://www.gao.gov/new.items/d04393.pdf

number of features are safety critical, a modest number of features are mission critical, many are just useful, and some have questionable values. Quite often, only a small subset of software needs to be highly reliable while many only need to have an acceptable degree of availability with respect to application needs.

This opens the door for a new architectural approach that uses simplicity to control complexity. Under this approach, we will 1) establish a protected simple and reliable core; 2) simplify the complexity of resource sharing; 3) allow only simple interactions between the core and the rest of the system; and 4) formally specify and verify the core and the interactions. This integrated architectural design and formal method approach would allow us to ensure the critical services while at the same time allowing us to safely exploit the service of non-safety critical complex software components with residual defects. This idea of using simplicity to control complexity can be illustrated by the following example.

Suppose that we can verify Bubble Sort program but not some ComplexSort program. One solution is to use the slower Bubble Sort as the watchdog. That is, we first sort the data items using ComplexSort and then pass the sorted items to Bubble Sort. If ComplexSort works correctly, Bubble Sort will output the sorted items in a single pass. Hence, the computational complexity is still $O(n \log(n))$ if ComplexSort is correct. If ComplexSort sorts the items in an incorrect order, Bubble Sort will correct it and thus guarantee the critical requirement of sorting. Under this arrangement, we not only guarantee sorting correctness but also have higher performance than using Bubble Sort alone, as long as ComplexSort works most of the time.

To ensure this arrangement can handle any fault in ComplexSort, we will make additional architectural arrangements: 1) memory safety: isolate ComplexSort from Bubble Sort so that Bubble Sort program cannot be corrupted; 2) errors space control: encapsulate the input data by a verifiable "Permute to that ComplexSoft can rearrange the data as needed but cannot modify any of them; 3) temporal protection: put in a timer to guard against infinitive loop in ComplexSort. Once we formally verify the architecture and the Bubble sort, we can make very power arguments: our system will guarantee the correctness of sorting and a performance no worse than $O(n^2)$, for all known and unknown complex sorting programs. Furthermore, should these complex program work, we can enjoy the performance gains.

The moral of this story is that we can exploit the features and performance of complex software, even if we cannot verify them, provided that we can guarantee the critical requirements with simple software and we are unable to ensure the safe interactions between them. This is the spirit of *using simplicity to control complexity* in Simplex architecture[5].

4.0 Summary

The ever increasing complexity in modern avionics software is a serious challenge for the safety and dependability of modern avionics systems. The wisdom of "Keep it simple" is self-evident. We know that simplicity leads to reliability, so why is keeping systems simple so difficult? One reason involves the pursuit of features and performance. Gaining higher performance and functionality requires that we push the technology envelope and stretch the limits of our understanding. Avoiding complex software components are not practical in most applications including modern avionics.

Since useful-but-unessential features cause most of the complexity, what is needed is an approach that allows us to safely exploit the features provided by complex components. This approach should combine complexity control architecture designs with formal methods, so that

- the architecture design
    - Ensures the simplicity of the critical service;
    - Ensures the simplicity of resource sharing;
    - Ensures the simplicity of the interactions between critical and non-critical components;
    - Provides mechanisms to exploit the service of complex components.
- the formal method
    - Ensures the correctness of critical service;
    - Ensures the safety of the interactions between critical and non-critical components.

[5] Readers who are interested in how Simplex architecture can be used in feedback control is referred to http://www-rtsl.cs.uiuc.edu/~lrs/Simplicity.pdf