
rerbit
Hello World:
Software In Space.

Content

Introduction

Software enjoys a strange reputation in the space industry. On one hand — let's call it the hero side — software tends to be the sought-after lifeline when it comes to solving problems on a troubled flying satellite, given that there is basically nothing else you can do after launch other than trying to fiddle with the on-board software and see if you can bring things back to normal. On the other hand — the villain side — software is considered the evil of all evils by outsiders. In any troubled space mission, the fingers naturally tend to point to software as the probable cause of failure (usually without tangible evidence) — any conversation next to a coffee machine between two non-software engineers (say, a mechanical and a thermal engineer) would either blame software, or radiation.

What does the evidence say?

In Fault-Tolerant Attitude Control of Spacecraft (Qinglei Hu, Bing Xiao, Bo Li, Youmin Zhang, Elsevier), the authors collected and analyzed spacecraft data from databases such as the Satellite Encyclopedia (TSE), Satellite News Digest, Mission and Spacecraft Library, Airclaims Space Trak, Space Systems Engineering Database (SSED), and the Mission Failure Analysis for NASA AMES Research Center. In terms of percentage of failures per subsystem, it shows that ACS (Attitude Control Subsystem, or the system in charge of controlling the orientation of the spacecraft in space) accounts for 32%, followed by Power subsystem. This should not come as a surprise[1], since the ACS subsystem is the most complex subsystem on a spacecraft, therefore it runs against the odds on surprises. Now when it comes to the different types of spacecraft faults, they can be further classified into four types of failures: mechanical failure, electrical failure, software failure, and other unknown failures. The mechanical failure mainly refers to the mechanical structure deformation caused by temperature changes, external force, friction, and pressure. The electrical failure is induced by power overload, short circuits, and abnormal battery power generation. The software failure mainly consists of incorrect computer instructions and onboard software abnormalities. As you can see in Figure 1, software faults account for only 6% of the total, whereas electrical faults account for 45% of them. And in Figure 2, you can see the types of faults for ACS only.

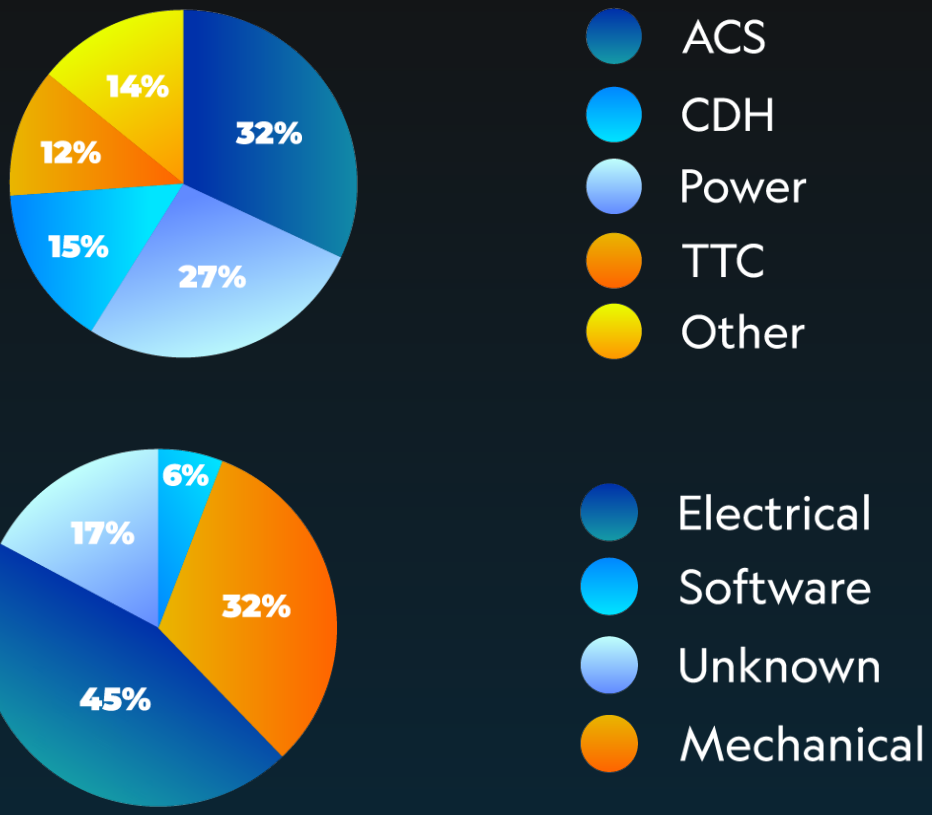

Fig. 1 - Subsystem faults and types of faults (source: *Fault-Tolerant Attitude Control of Spacecraft* (Qinglei Hu, Bing Xiao, Bo Li, Youmin Zhang)

Fig. 2 - Types of faults in ACS

According to the aforementioned source, there is no evidence that software faults are the main causes of mission losses. Or, put more bluntly, evidence indicates software is seldom the problem. There have been, though, some software failures which have gone down in history as very resonant, such as the Ariane 5 failure[2], Mars Climate Orbiter[3], and others[4].

Even if software were the problem, there is no chance a spacecraft can be launched without any software running on it. Satellites are becoming more and more software intensive and nothing indicates this trend will stop anytime soon. Then, let's discuss what's so special about making and running software for space missions.

What does it take to run software on a spacecraft?

Well, for sure you need a microprocessor, and some memory. These can be discrete — as in, individual — devices, or they can be part of the same integrated circuit. In any case, what we call software is basically a set of instructions which are fetched from a memory and interpreted by a microprocessor and then executed. Such instructions are typically mathematical operations of different kinds and data movements from one place to another. This doesn't really differ from the software which runs on your laptop. The main difference is that the computing resources spacecraft have tend to be more specialized compared to the resources consumer electronics as laptops have. For example, spacecraft may use microprocessor architectures which consider reliability matters by adding voting capabilities in their internal registers. Because energetic particles can interact with the electronic devices, we use such as memories or internal registers of microprocessors, then it means that a particle may alter the content of a memory cell or a status register. One single bit being changed by radiation can cause serious consequences to the execution of software. Hence, adding voting capabilities ensures that, upon the upset of a memory location or internal content of the processor, still the value finally used will require that 3 identical registers will show the same reading. This comes with the penalty of a larger area of semiconductor used for implementing the voting scheme. Because bits can be upset at any time while in orbit, several techniques to mitigate this issue have been devised.

For example, memory scrubbing, which consists of reading from each computer memory location, correcting bit errors (if any) with an error-correcting code (ECC) and writing the corrected data back to the same location, parity bits, checksums, and other error detection and correction schemes.

Computing resources on spacecraft have been historically more modest than resources on ground-based computers, although this is a gap that has been consistently shrinking in recent years. On-Board computers are advancing rapidly, with more resources than ever before: more storage, more floating-point operations per unit of time (FLOPS), more millions of instructions per second (MIPS), and the like. On-board computers still remain close to what are called "embedded systems", which are computers with specific roles and functionalities. Unlike a laptop — which is sold without really knowing or caring for what the laptop will be finally used for, whether it's accounting, graphic design, or music production — spacecraft computers tend to be optimized for the specific function they are ought to perform: attitude control, thermal control, command and data handling, payload data acquisition, etc. Each one of these requires specific interfaces, specific libraries, sensors, actuators, etc. For example, attitude control needs to perform intricate calculations about rotations, matrix operations, sensor data fusion, and a lot more. And it has to talk to a variety of sensors with different characteristics, such as sun sensors, star trackers, gyros, magnetometers, and so forth.

How is software updated or changed in orbit?

By means of... software. Yes, software can be coded to help modify software, including to modify itself. Spacecraft computers are usually equipped with what is called a bootloader. A bootloader is a program whose sole function is to fetch a piece of executable code — typically an application — from some location — for example a memory — and place it in a way the microprocessor can find it and run it. The bootloader is invoked during the power-up sequence of the processor, and usually loads and executes all the time the same application, over and over. Under special circumstances, the bootloader might be commanded to load an image from some other location or take a newimage through some interface such as a serial port or similar. Mind that bootloaders can also create some headaches. For example, if during the process while a bootloader is flashing a new application in non-volatile memory something odd happens and the image gets corrupted, then the application program will never execute because it is only partially present. This is typically called "bricking" the poor thing — because the device becomes as useful as a brick — and it's not exclusive for space. You can brick your smart TV, your Wi-Fi router or your phone while updating firmware. The key is to ensure the bootloader can always recover from a failed flashing procedure. Goes without saying, bootloaders should not be able to overwrite themselves, at least not the lowest level, critical bootloaders.

What type of languages are used for coding flight software?

In theory, any language can be used for coding flight software. Languages never make it to orbit, only machine code does (unless the compilation happens in-orbit). Then, if the processor being used has a compiler which can translate whatever obscure language of your choice to its machine code, you're good to go. Because flight software has historically stayed quite "embedded" and close to the hardware, compiled languages have been more popular, notably assembly and C. These languages offer to the programmer the tightest control of what's happening as the software runs, while paying the penalty of a lower code readability, reuse and whatnot. Some other compiled alternatives to improve reusability is to employ Object Oriented Programming (OOP) techniques, although this claim tends to be disputed by the old guard. For example, in the book Coders at Work[5], Joe Armstrong — creator of Erlang — says on software reusability and OOP:

> ***"I think the lack of reusability comes in object-oriented languages, not functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."***

On the other hand, interpreted languages such as Python are possible in spacecraft. Provided the on-board computer can run an interpreter, all the rest comes reasonably easy. Interpreted languages tend to be relatively slower and sit on higher abstraction layers (don't ask an interpreted language to handle hardware interrupts or hard-real time deadlines) but are highly flexible for scripting and automating non-critical things on-board.

Can I run Linux on a satellite? What about Windows?

If you can, then you must. If the spacecraft computer is capable of handling Linux — this means, it has a certain number of resources and a Memory Management Unit, or MMU — then Linux is your friend. Mind that with great power comes great responsibility: a full-blown Linux OS needs to be administered accordingly, and this may require some sort of shell for accessing and monitoring the operating system's resources. Since satellites are remote systems —there is a lousy radio link between the spacecraft and the ground — this means that such a noisy link will be the only way to shell the remote operating system. Therefore, remote shells such as SSH (encrypted) or Telnet (unencrypted and very basic) have to be used on top of radio, which calls for using IP stacks on top of — for example — CCSDS. Not such a big deal if the link budget is sound, this means, if the energy per bit sent back and forth is reasonably higher than the overall noise. For high-latency links, for example for distant orbits, conversational protocols such as TCP can be problematic, therefore shell access to a remote Linux-based spacecraft can be cumbersome. Once a proper link is established, then the handling of the remote Linux system running on board the satellite becomes more of a sysadmin task. Using Linux and IP based links blurs the line between operating a satellite and operating a server on a network.

As for Windows, technically speaking, you could run it on a satellite. But because Windows is an operating system which heavily relies on graphical user interfaces, the only way you could possible use it is to either use something like VNC to remote desktop it (although considering the radio link speeds and the amount of data VNC would generate would turn it very challenging), or put a screen on board (which needs to withstand launch and vacuum) and hook a camera looking at the screen while you send videos frames at a lower rate to the ground. We recommend you just discard the idea for now, but if you insist, don't go without this if you want to live.

What kind of skills are required for doing Flight Software?

It does require some level of acquaintance to the typical satellite architectures, interfaces, and whatnot. But all that can be learned. Granted, switching to flight software will be less of a leap for those coming from the embedded world. And because a satellite is a physical and dynamical system, the software developed for it must be designed to be aware of such dynamics. In short, the (minimal) complexity of software which controls a physical system is given by the complexity of the underlying physical system. It says minimal because of course software engineers can always overengineer it. And they will if given the chance.

How is Flight Software designed?

It's an incremental work. But, if you take two satellites from two different operators and you somehow reverse engineer the machine code into source code, those two systems will show similarities. Every flight software needs some of these modules or building blocks: telemetry handling, telecommand handling, FDIR (failure detection, isolation and correction), thermal control, power control, etc. Flight software can be designed and developed from absolute scratch —software engineers will always try to convince you it's the only way to go, or it can rely on existing software frameworks which already contain some of the typical building blocks. Some of these frameworks are even flight-proven and open source, which makes them very attractive for low-budget New Space missions or University projects.

What are software-defined satellites?

The software-defined fad has gotten a bit out of hand lately. It is used too loosely and stretched to levels where it stops having any meaning. But, in the strict sense of the term, when we say software-defined anything, we mean "something which has been done traditionally by means of mechanical or electrical devices but now is done by means of software". Sounds obvious, but it's not. For example, take software-defined radio. Historically, modulating/demodulating, mixing and filtering radio signals has been done on discrete electronic —active and passive— devices. With the progress of semiconductor technology in areas such as analog-to-digital converters and digital signal processing, now it is possible to move such things into software. How so? Well, fiddling with signals is ultimately a mathematical problem. Filtering, mathematically speaking, is done by computing the convolution of a signal (or, in the discrete jargon, a sequence) with the step response of the filter. Or, equivalently, multiplying their frequency responses in the frequency domain. This means, a superheterodyne receiver is nothing else but a sequence of mathematical manipulations of the input signal. So, if you manage to digitize the signal as soon as possible in the chain (this is, as close to the antenna as possible), then you can do the math operations fully on the digital domain. You get the gist of it.

On the same line, spacecraft have had some things "hard-coded" by hardware elements, for example communication protocols, namely CCSDS. Traditionally hardwired in highly specialized ASICs and FPGAs, now the stacks can sit on software, which adds a lot of flexibility when it comes to adding security layers and applying patches in case of cybersecurity threats. Also, software-defined protocols allow to scale up or down resources depending on the mission requirements. For data intensive space applications, software-defined techniques allow to readjust mission configurations post-launch, providing a more cost-effective way to optimize operations once the spacecraft faces the elements.

Although there are some particularities about running software inside a metallic box which is orbiting at hundreds of kilometres above the ground while flying at thousands of meters per second of velocity, software is still software. Software itself remains unaware about the altitude at which it's executing. The software engineers are the ones in charge of adding the correct safeguards to make the software execute seamlessly while flying alone in space. Regardless of where the software runs, the work follows the same flow: defining data structures and their relationships. You can over-engineer it by adding fancy methodologies and diagrams, but space software engineering, at the end of the day, still boils down to data and how it behaves over time.

Linus Torvalds stated years ago:

"Talking is cheap, show me the code".

1. A veteran space engineer once said:

   “There are two main reasons why space systems fail: one is attitude control, the other one is funding”.

2. http://sunnyday.mit.edu/nasa-class/Ariane5-report.html

3. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

4. https://en.wikipedia.org/wiki/List_of_software_bugs#Space

5. https://codersatwork.com/

reorbit

At ReOrbit, we develop technologies to make spacecraft platforms modular and configurable. Through a software-defined architecture, we unlock new functionalities such as built-in autonomous orbital capabilities.

Our main focus in all of this is streamlining data flow, as we deliver timely and flexible missions at any orbit. With your data being transmitted fast and with it, your mission being executed, while keeping cost and time-to-orbit low, you can even grow the value of your satellite after launch, as we can reconfigure the software in-orbit.

This transforms technology and enables space applications of our current and future society where efficiency, sustainability and security are unlocked to their full potential. The solutions to reinforce the mission of helping humans and machines interact and exchange data seamlessly are ready and waiting for their turn to shoot for the stars.

Connectivity in space. Simplified.