

ONCE YOU START ASKING



Aviation 2

Flight Control Systems

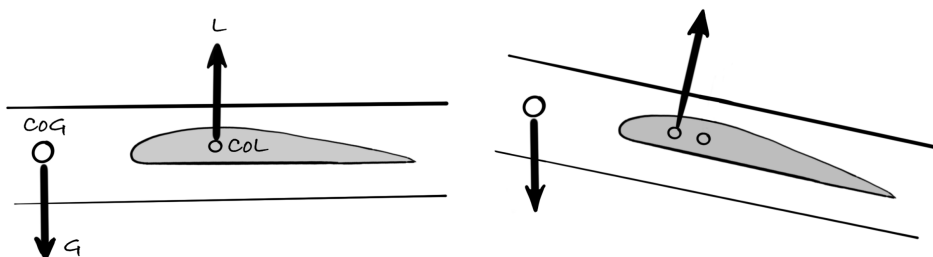


INSIGHTS, STORIES AND EXPERIENCES
from ten years of reporting on science and engineering
MARKUS VOELTER

Lockheed F-16 . Let's talk about the F-16, one of my all-time favorite aircraft. The Viper was "born" in the same year as I was, 1974, and today, with over 2,000 flying and continued production, is still the world's most widely used fighter jet. Single engine, single seat, single tail, and just overall absolutely beautiful. In 2017 I had the opportunity to get to know one personally at Nellis Air Force Base. A few months earlier I received an email from one of my listeners, Lt Col Jan Stahl, which basically said "Hey Markus, I am working for the US Air Force, flying the F-16 as an aggressor at Nellis. If you ever happen to be in the area, you are welcome to visit the base, check out the airplane, and maybe record a podcast." That's the kind of listener initiative I love, and of course I made sure that I was "in the area" a few months later. Let me tell you: it's quite cool to sit in the cockpit of an F-16 while an F-15 taxis past majestically a few meters away. While I was in the US I also visited NASA Armstrong Flight Research Center at Edwards Air Force Base, the place where much of the military flight research that I was so fascinated by as a child and teenager was conducted. Hallowed ground for an "avgeek": a wonderful week. But I digress.

The F-16 is a relaxed-stability airplane: it is intentionally designed to be unstable. More specifically, it has slightly negative static stability and negative dynamic stability. This is useful for a fighter aircraft because you want it to be highly maneuverable. Say you want to fly a turn. Because of negative dynamic stability, you basically just have to start a roll "oscillation", and it will accelerate, making the roll very quick. Once you get to the desired roll angle you actively stop the oscillation. Because of its relaxed static stability, there is no tendency to roll back to zero degrees bank—it will stay in whatever attitude you've put it in.

How do you make an aircraft unstable? We previously saw that a short tail leads to directional instability. Making an aircraft "tail-heavy" also helps.



In the illustration above the left part shows a stable aircraft: the center of gravity is in front of the center of lift, the (imaginary) point at which the wing's lift connects to the aircraft. This keeps the nose down, slightly accelerating the aircraft. As the AoA increases, the center of lift moves forward, which is a fundamental characteristic of lift-generating airfoils. It's not a problem if your center of gravity is sufficiently forward, as it will remain in front of the centre of lift and work to move the nose down; this arrangement of CoG and CoL is stable: it counteracts a pitch-up movement. But if the CoG is even slightly behind the CoL, increasing AoA means that the CoL will move further forward, which, because of the increasing moment arm, leads to an even more increasing AoA. This positive feedback causes instability.

An unstable airplane, especially one that is seriously unstable, is not manually controllable by a pilot, especially at high speeds. And the real job of a fighter pilot is to focus on the tactical situation and the mission rather than on the mechanics of flying. This is where flight control computers come into the picture. Even if the pilot keeps the stick centered, the controller actively actuates the various control surfaces to prevent divergent oscillations. I read somewhere—I can't find the reference anymore, so take this with a pinch of salt—that the F-16 self-destructs from these oscillations within one second if the flight control system fails. To avoid this fate the flight control system has lots of redundancy in its computers, and is supplied by five different power sources, including batteries, the emergency power unit and the F-16's regular electrical buses.

The F-16's flight control system processes two sets of inputs: data about the current state—attitude, speed, plus various data about the airstream, such as angle of attack, side-slip angle, total pressure and total temperature—as well as the pilot's input about how he wants this state to change, as expressed

through stick and rudder. It then computes appropriate deflections of the elevator, aileron, rudder and flaps to effect this change. There is no direct connection between the pilot's controls and the control surfaces—it is all mediated by a computer. A computer is fast enough to create artificial stability in an unstable aircraft.

This kind of flight control system is called fly-by-wire, because electrical wires are used instead of cables, rods and cranks to pass signals to the hydraulic actuators that drive the surfaces. But maybe it should be called fly-by-computer, because it's probably more important that there's a computer between the inputs and the surfaces to interpret the pilot's commands in light of the current flight state. The F-16 was the first operational fighter that used fly-by-wire, which contributed to its exceptional maneuverability. Today all fighters use this approach. Many of the newer ones have additional control surfaces: for example, the Eurofighter has canards, little wings at the front of the fuselage, while the F-22 has thrust-vectoring nozzles, both intended to help with high-AoA flight. It would not be feasible for the pilot to manually use these additional controls effectively: an integrated computer-based flight-control-system is essential.

The fly-by-wire system ensures that the pilot experiences a stable airplane. In fact, he experiences an airplane that is much more stable than a traditional one would be, while at the same time being extremely agile. Jan recalls: "One thing that hit me about the F-16 from the get-go is how easy it is to learn and how easy it is to fly compared to any other platform that I flew previously." Jan transitioned to the F-16 from the non-fly-by-wire F-15. Of course, even in non-fly-by-wire aircraft like the Eagle, the stick and pedals are not connected directly to the control surfaces by wires, rods and cranks, because the aerodynamic forces are much too great for a pilot to be able to move the surfaces manually. As in the F-16, they are hydraulically actuated, but the valves that control the hydraulics are themselves controlled by cables, making the deflection of the control surfaces directly proportional to the pilot's input. No electrical signals and computers are involved in the primary flight controls, although there is one involved in the automatic trim system of the F-15. In traditionally-controlled aircraft like my ASG-29 the pilot can feel the state

of the aircraft to some degree. For example, when I fly faster the stick becomes heavier to move, because the aerodynamic forces required to move the surfaces become greater. As I slow down towards the minimum speed, the stick becomes “wobbly”; when I approach stall the stick starts shaking a bit, because the buffet created by the wing’s separating airflow strikes the elevator. My reptile brain uses these cues a lot when controlling the aircraft, but if you have no direct “connection to the air” this feedback loop is not available. Aircraft like the F-15 therefore have artificial feedback systems, essentially springs on the stick that are preloaded with a force that is proportional to airspeed. Transport aircraft have stick shakers that indicate an approaching stall. The F-16 has no such feedback at all. Jan: “In the F-16 the buffet cues are mostly absent, so you grow very reliant on displays that precisely indicate the state the jet is in.” In fact the stick did not move at all in the original F-16, it only sensed the force that the pilot applied to it. Pilots didn’t like this, because, in addition to stick force, the stick’s position also provides important feedback for a pilot’s subconscious flight control algorithm. General Dynamics subsequently changed it and the stick now moves a few millimeters, but there is still no force feedback.

Since the computer has a say in the translation of the pilot’s inputs to what the control surfaces do, it might as well prevent the pilot from doing stupid stuff, such as trying to increase AoA over what the wing is capable of, or stressing the aircraft beyond the g-limit it can tolerate. Such protections are common in all fly-by-wire aircraft. Jan: “It is much easier to perform the F-16 to the maximum capability of the flight envelope than it is a jet like the F-15, because I really don’t need to think about it. All I need to do is to pull on the stick: if the computer calculates that in the current state I can get to 9 g it will get me to 9 g and it will stay there, as long as the aerodynamics permit. In the F-15 it was a much more complex process in the sense that I had to check my airspeed, and based on the airspeed I had to essentially forecast how much g is available, and if I know that g-available is higher than the maximum g the airframe can permit, I have to make sure that I’m not pulling my stick all the way back. I have to pull a little bit slowly to a more restrictive position, until my g-available becomes less than my g-allowable, at which point I can bring the stick all the way back.” To help the pilot, the F-15 has

an Overload Warning System, which sounds beepers in the pilot's headset to indicate what proportion of g-allowable the pilot is pulling: 91–94% is denoted by a single-rate beeper, 95–97% by a double-rate beeper, and 98–100% by a solid tone. Beyond 100% a voice annunciation system (traditionally called “Bitchin’ Betty”) tells the pilot “Over-G, Over-G”. In my glider there are no protections: if I fly at 250 km/h and pull the stick fully back, something will break—probably the elevator. You just don’t do that, just as you don’t jerk the steering wheel all the way to the right at 200 km/h on a German autobahn.

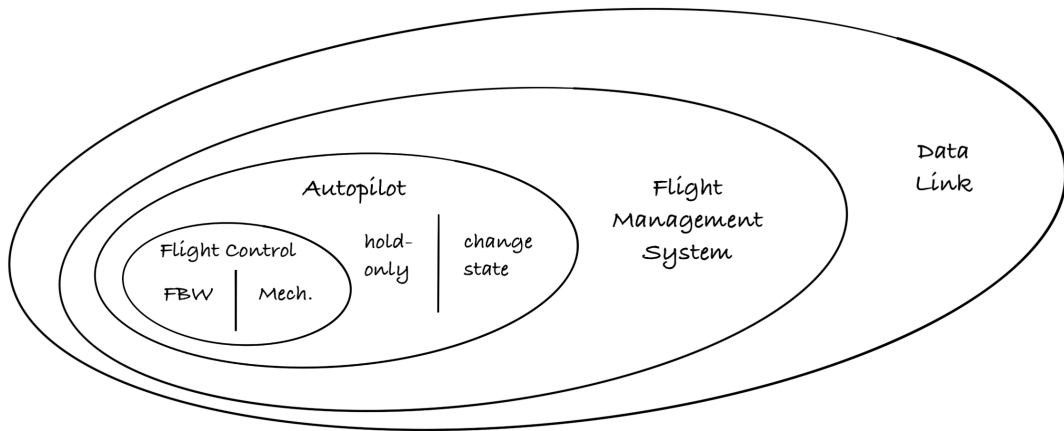
We will look at the workings of a fly-by-wire system in more detail below, but before we do I want to differentiate the flight control system from other automated systems in the cockpit. It’s important to realize that in fly-by-wire aircraft you still manually control the aircraft’s speed, altitude, heading, bank angle or g-level. It’s just that you don’t fly the unstable and hard to control physical airplane, but instead a more stable, more care-free “artificial” airplane that is simulated by the flight control system.

Autopilots are different, and they have existed since well before fly-by-wire systems. The simplest forms just maintain aspects of the current attitude: altitude hold, airspeed hold, heading hold. Once you have manually put the aircraft into a particular state, the autopilot holds it for you. This is very useful if you have to fly long straight legs. The SR-71, on its Mach 3 cruise, used basically such a system. More advanced autopilots can also make changes to the flight state. For example, in modern airliners you can dial in headings, speeds, altitudes and rates of climb and descend and the aircraft will “get you there”. Today’s autopilots can also fly towards a radio beacon or towards a GPS location, or fly “down” an instrument approach. Since we mentioned Bitchin’ Betty above, I should probably mention that the autopilot is traditionally called George.

Airliners also use what is called a flight management system. This allows pilots to plan their complete route, with multiple waypoints, speeds and altitudes, taking weather and fuel consumption into consideration. The aircraft will then fly the route automatically. The autopilot is part of the flight management system. Today, flight management systems also have various forms of datalink, such as ACARS, to allow for remote monitoring by an airline’s

maintenance department. Pilots interface with all this through a set of displays, keyboard and the like. In Airbus aircraft, for example, this system is called the Multi-function Control and Display Unit. It also provides access to “systems pages” that allow pilots to monitor and control many of the other systems on a large aircraft, such as fuel—which might have to be pumped between tanks during a flight to manage the center of gravity—the electrical systems or the hydraulics.

Military aircraft have systems that are similar to a flight management system, usually called a Mission Management System. This also takes care of weapons and other stores, integrates the various sensors (typically radar or infra-red) and supports the pilot in aiming and releasing weapons. For data exchange, NATO aircraft use Link16, a networking system that allows fighters to exchange sensor data, or an AWACS to “push” the big picture to the fighters. However, these are separate systems and are not technically part of the mission management system.



None of this is related to whether or not an aircraft uses fly-by-wire. Traditionally the autopilot used servo motors to control the wires, rods and cranks that operated the surfaces. For example, a 747-400 has a modern flight management system with all the “fly-me-there” automation on top of an aircraft that uses conventional cables to connect the cockpit controls to the hydraulic actuators that deflect the control surfaces. Similarly, an aircraft can have a glass cockpit (screens instead of “steam gauges”) even if it does not feature a

fly-by-wire control system.

A final word about automation. There is an ongoing discussion in the commercial pilot community about how much automation is too much. There are several arguments here. One is that if pilots only “dial in” altitudes, headings and speeds, and routinely let the aircraft land automatically, they might lose their manual flying skills. This is certainly true—practice makes perfect, and lack of practice is bad. On the other hand, it is probably a good thing if pilots get help from automation if they have to land an aircraft in adverse weather after a long and fatiguing transatlantic flight. A second argument, however, concerns the transparency of the system. As long as everything works, “magic” is fine. But when things start to fail, it is often crucial for pilots to understand the interdependencies between systems to be able to diagnose problems correctly and react appropriately. If you read the training manuals, there is indeed a lot of mode- or fault-dependent behavior: “If X happens, then Y is no longer done automatically”, where X and Y are functionalities that are not obviously related. Instead, the dependency stems from details of the underlying system. For example, two functionalities might run on the same computer. Understanding these dependencies certainly takes a while. In the end, an accident is always a combination of several causes—the famous Swiss cheese model in which all the holes have to align to lead to a catastrophic outcome. One of the holes can certainly be lack of practice, or the lack of understanding of a complex system. We will return to this problem after we have discussed how fly-by-wire systems work.

~ ~ ~

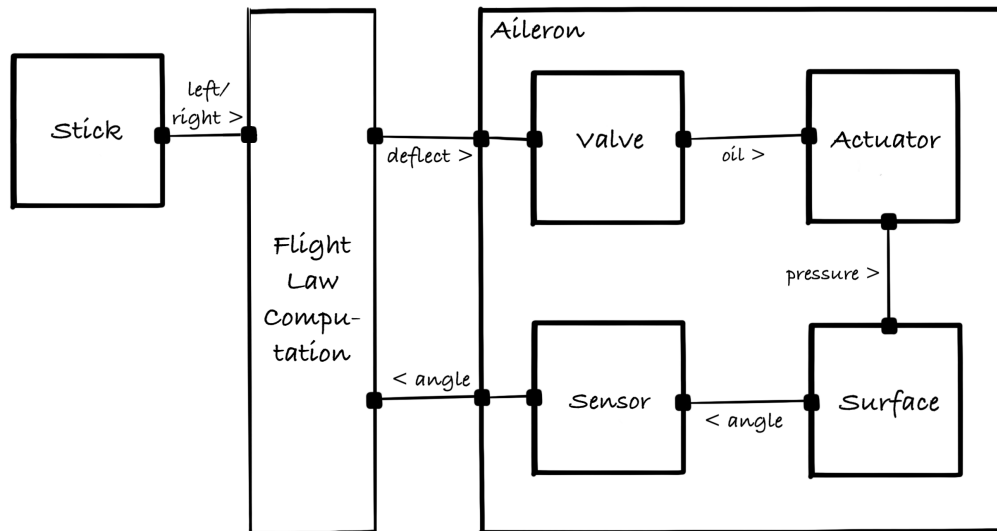
Airbus A320. The Airbus A320 is one of the most successful airliners in history; only the 737 has sold better than the A320 and its brethren, the shorter A318 and A319 and the longer A321. The A320 family was also the first commercial fly-by-wire airliner. The A320 is not dynamically unstable, so its reasons for employing fly-by-wire are different. First, a fly-by-wire system is lighter than cables and cranks, which saves fuel over the lifetime of the aircraft. Second, if you have ever looked into the wheel well of a traditionally controlled airliner, you will probably agree that it is a benefit if you can get rid of the mechanical complexity of the control cables; they produce quite a

bit of maintenance effort. The primary driver for fly-by-wire systems in airliners is safety: assuming that you can build a reliable fly-by-wire system, it can provide lots of protection mechanisms and benign flight characteristics, similar to the F-16's just-pull-and-I-will-give-you-appropriate-g-levels feature Jan mentioned earlier. A secondary reason is family commonality: because the system simulates a virtual aircraft, you can simulate the same virtual aircraft for several real aircraft types. In other words, the A318, A319, A320 and A321, and to some degree the A330 and A340, all handle very similarly, simplifying pilot training and currency. The system can presumably also fly more precisely than a pilot, which may lead to increased efficiency because of fewer unnecessary control surface deflections or imperfect trim. Finally, you might be able to build a lighter structure, because the system can actively compensate for gusts and turbulence, reducing the load on the physical airframe—search for “aileron waltz” on YouTube for a nice demonstration of this on the A380. In 2014 I chatted with Reinhard Reichel, professor for aircraft systems at the University of Stuttgart and an expert in fly-by-wire systems, who has a deep knowledge of the A320 flight control system. In the following I will give you a sense of how the A320 flight system works and how it has been engineered to be reliable and safe.

The design of a fly-by-wire system is an exercise in systems engineering, involving hardware, electronics and software. Engineers start out by capturing the requirements that describe the intended functionality of the system, as well as its quality attributes. A particularly important quality attribute is reliability, which in turn can be broken down into safety and availability. Safety means that the system does not produce any erroneous, dangerous behaviors (such as fully deflecting the rudder during high-speed flight). Availability means that it is extremely unlikely that the system fails to provide its functionality. For commercial airliners, the European Air Safety Agency's CS-25 certification specification requires a reliability of 10^6 for an aircraft as a whole. This means that a fatal accident is acceptable once per million flying hours. This number governs the design of the flight control system. Reinhard Reichel summarized the challenge as follows. CS-25 requires 10^6 . The pilot is statistically responsible for 80% of accidents, the structure for 10%, so there are around 10% remaining for all other critical systems. Taken together, they

can therefore only fail once in every 10^7 flight hours. The aircraft has around 100 critical systems: one of them is the flight control system. This means that the flight control system has to achieve a reliability of no more than one failure in 10^9 flying hours. However, many of the components, such as the computers, have a much lower reliability: 104 is typical. So the challenge of engineering the flight control system is to assemble a system that achieves a reliability of one failure in 10^9 hours from components that individually have a much lower reliability. To understand how this is achieved, we have to zoom out a little bit.

System engineers work a lot with models. A model is an abstracted, simplified, less-detailed representation of a system that supports analytic methods that make reliable predictions about the real system: we will discuss models much more in the next chapter. Maybe the most important model is the functional model, in which a system is hierarchically broken down into subsystems and the interactions between these subsystems are captured. The following representation resembles block diagrams that are found in many systems engineering modeling languages, and in particular SysML.



The system shows three functions: the stick, operated by the pilot, the flight control law computation, which takes the pilot's input and determines which

control surfaces to deflect and by how much, as well as one of these control surfaces, an aileron. The flight law computation tells the aileron by how much to deflect and receives feedback from the aileron about its actual deflection. It then runs a control loop to keep the aileron at the angle that it wants it to be based on the stick position (and lots of other sensor inputs that we will ignore here). The flight law considers the aileron as a black box: commanded deflection in, actual angle out. But the aileron system is itself built from several subsystems: the electrical deflection request arrives at a valve that opens or closes according to the received command. This allows oil into the actuator, which then puts pressure on the actual surface, which then moves. An independent sensor measures the angle of the surface and reports it back to the flight law (indirectly—the actuator law is a separate system). For the engineer who designs the flight law, the aileron is a black box. But for the engineer who designs the aileron, it is a whole (potentially complex) subsystem in its own right. This hierarchical decomposition is crucial to being able to understand and design complex systems.

The various subsystems exchange data. For example, the flight law computation sends a deflection request to the aileron and the aileron sends the measured angle back. In early stages of system design this level of precision is enough to allocate basic responsibilities to subsystems (and recursively, to their sub-subsystems). But as we make progress in the design we have to become much more precise: is the angle in degrees or radians? What is the valid range? How many decimal digits? What is the error in each value? Once we think about representing these values in software we have to decide how to encode them. Let's say the angle is a value between -30.00 and $+30.00$ degrees. Should we encode this as a floating point number or as 16-bit integer, multiplied by 100, as a range between $-3,000$ and $+3,000$? Finally, we will have to define how often a new value must be sent. This is very important: if the rate is too low the control algorithm might not be able to run at the required precision, and if the rate is unexpectedly high the receiver might be overloaded, for example because data buffers could fill up too quickly. The documents that capture all of this are called interface descriptions. They might be just words initially, but as the system design progresses they are

expressed precisely in various modeling or programming languages. Interfaces are agreed on by the producer(s) and consumer(s) of signals. Conceptually they are then attached to ports: ports are the “connection points” of the boxes that represent the functions in the diagram above. Two ports can only be connected if they have the same interface, to ensure that both data producer and consumer have the same understanding about the data they must exchange. The notion of an interface also extends to the physical world; in this case they do not specify the exchange of data, but the exchange/flow of heat or materials or forces.

Clear, unambiguous and agreed interfaces are the first ingredient for building a reliable system. They avoid any number of errors from “misunderstandings”, and they also help with finding problems during integration testing. For example, if we specify the rate at which the flight law is able to receive and process the angle signal, we can monitor this rate as the system runs. If the rate is too high or too low, we can log an error that specifically indicates this problem. If we didn’t do this we’d eventually see the flight control become less accurate due to reduced algorithmic precision because the signal is not received at the minimum rate, or fail entirely because the input buffer overflows and the program crashes. The more specific log entries are about their root cause, the easier it is to fix the problem. A crash, and even an overflowing buffer, can result from all kinds of problems.

We can also detect flaws in the functioning of the aileron; if the aileron reports an angle that is outside the $-30/+30$ degree range, something must be wrong. Here it is not obvious what the root cause is, but we know that something is wrong with the aileron. This approach is known as plausibility checking. The flight law computation could detect this, because it is a consumer of the angle signal produced by the aileron. There are also more advanced forms of plausibility checking. For example, one could say that, if the valve is opened and the sensor does not detect a change in the angle of the surface after some specified time delay, then something must be wrong. In fact the A320 flight control system runs a model of how the surface is supposed to behave as a consequence of the valve opening, and if the surface behaves differently it constitutes a fault. If this kind of plausibility checking were to

run inside the aileron function, then the aileron would have to report an additional signal back to the flight law computation that reports its “valid” status to notify the flight law computation of the failure of the aileron subsystem.

The flight control system in the F-16 works similarly. Jan explains: “If the four channels of the flight control computer don’t return the same flight control command for a given set of inputs, it will filter out the offending channel and pick the control deflections the other three channels calculated. If the flight control system is unable to do that, then it will lock out the affected servo actuator and give me reduced flight control authority around that axis until I can verify that the control system isn’t somehow getting invalid data from the air data computer. If I can isolate the sensor that’s feeding incorrect data to the air data computer, I can then reset the appropriate actuator and continue my mission. If not, I may have to terminate the mission and try to land with degraded flight controls around that axis, which the flight control system can partly compensate for with the remaining flight control surfaces and the trim.”

Now let’s look at different kinds of faults. A fault might happen spuriously and then disappear: a bit flip caused by solar radiation is an example. Such faults do not necessarily indicate a problem with the design or construction of the system, but the system has to be able to cope with them nevertheless. The classical approach to detecting such errors is to use a checksum: for a four-digit number XXXX you compute an additional value Y from the digits and append it to the number, XXXXY. The receiver performs the same computation from the received digits X’X’X’X’, and if one has changed (because of a bit flip or any other reason) the checksum will be different and won’t match the one received with the number. To “fix” the fault you just try again, and very likely the problem will disappear.

Then there are faults that result from wear or because a mechanical part is not strong enough. Importantly, these kinds of faults become more likely as the system ages. For a system to be able to function in the face of such faults, critical components are either significantly over-designed, closely and regularly inspected over the lifetime of the system, or they are installed more than once. For example, in the A320 there are in fact two actuators for each ai-

laron, so that if one fails the other can take over. A third kind of fault is systematic, which means that it always occurs if a particular condition is met. Such faults often occur in software and are usually a design or implementation error. For example, the internal model of the control surface might expect the aileron to react faster to an opening valve than it actually does and report an (invalid) error, for example at high speed. Finally, there are logic errors where the system correctly does what it was programmed to do, but that program specified inappropriate or unsafe behaviors. Such a logic error was partially to blame for the accident to Lufthansa Flight 2904 when landing in Warsaw, which overshot the runway in poor weather. Consider this paragraph from the accident report:

The aircraft automatics comprises, for basic landing configuration if the aircraft (i.e. with flaps extended to FULL position), the programme which subjects actuation of all braking devices to some specific conditions. Ground spoilers, when selected, will extend provided that either shock absorbers are compressed at both main landing gears (the minimum load to compress one shock absorber being 6300 kgs), or wheel speed are above 72 kts at both main landing gears. Engine reversers, when selected, will deploy provided that shock absorbers are compressed at both main landing gears. Such a logic results in the lack of possibility of immediate actuation of two mentioned above aircraft's braking devices without meeting the conditions described.

While mistakes by the crew also contributed to the accident, the aircraft's logic that determines when it is considered to be on the ground, and that therefore brakes and ground spoilers are allowed, was partially to blame: basically, a bumpy landing and hydroplaning “confused” the system. As a consequence of this accident, Airbus issued a software update that changed the required weight from 6.3 tons to only 2 tons. Preventing these kinds of faults is extremely hard: essentially they are a result of failure to anticipate the conditions during landing correctly. Testing helps, but you still have to test in conditions where the fault reveals itself. In the case of Flight 2904 the system did exactly what it was designed to do—it's just that it was designed to do the wrong thing.

We can now return to the CS-25 requirement of a reliability of 10^7 . Systems

engineers will assign reliability factors to each of the sub(sub(sub))systems. The respective factors have to be justified either by experiment, long-term experience or through a fault-tree analysis that mathematically derives the reliability. Let's look at our aileron again. Say the actuator fails. The rest of the aileron cannot compensate for this; there's nothing it can do to keep the surface moving. To make it more reliable, we can use redundancy and add another actuator that can take over if the first one fails. Assuming an actuator fails with some probability P_A , then the probability of both failing is $P_A * P_A$. Since probabilities are numbers between zero and one, this product is smaller than the original P_A : that is, the probability of two actuators failing at the same time is lower. However, for this to be true, the failures of the two actuators must be independent. For mechanical systems this assumption is usually valid, but if they for example get their hydraulic pressure from a single supply, they suddenly have a common failure mode if the hydraulic system fails. Finally, the assumption requires that a failing actuator does not prevent the other one from performing its job by blocking the control surface.

So how does the A320 do it? It indeed has two aileron actuators, but they are supplied by two independent hydraulic systems. Each actuator has an override valve that connects the two chambers of the cylinder to each other, ensuring that a failing actuator cannot block the other from taking over. That valve is commanded through a separate signal. It is similar in the F-16. Jan: "Two channels of the flight control computer control one set of actuators, and two channels control the other. At the same time, the actuator is actually comparing the inputs from both opposing channels, and locking out the actuator as described before. Getting a servo actuator warning light is actually relatively common when 'assaulting the aerodynamic limiters', such as in a visual fight at high g."

We can now see how redundancy in the aileron subsystem helps us build a reliable system (the aileron) from components that do not have the required reliability (the actuators). In this case it was local component redundancy: we simply duplicated a component. When designing an aircraft, however, redundancy must be seen relative to the capabilities of the system and its overall reliability. This is beneficial, because it has the potential to increase reliability without too many additional components (which add weight, complexity and

maintenance effort). This can be nicely illustrated with roll control. Roll control is a capability provided by the flight control system, but it cannot just be realized through ailerons; it can also be realized using the spoilers. These are a bit less efficient, but the A320 can tolerate the failure of an aileron because it can compensate with the spoilers. The flight law computation, when detecting a fault in an aileron, will automatically fail-over to the spoilers. They are mechanically independent and have no common mechanical failure modes. As we will see, ailerons and spoilers are also controlled by different computers, another ingredient to overall system reliability.

Software does not only have logic errors such as those that lead to the crash of Lufthansa Flight 2904: the underlying computer hardware might have its own low-level logic errors, or the compilers that generate the machine code from high-level programming languages might have errors. To avoid fatal effects from such errors, dissimilarity is used: redundant computers that have the same responsibilities don't use the same hardware architecture, programming language or compiler. This avoids common failure modes that might otherwise result from low-level logic errors in the hardware or compiler. This principle is used in all the computers in the A320's fly-by-wire system. In total the system has five full-authority computers that control pitch, roll and yaw, all of them active all the time. They perform both the flight law computation as well as control of the actuators; there are no dedicated actuator control electronics. There is enough redundancy in the system to provide full functionality if one computer fails, and safe operation of the aircraft is possible even with only one computer working. However, the system degrades in steps between the two extremes of full and minimum functionality, as we will see below. In the A320 there is also mechanical backup for trimming the horizontal stabilizer and controlling the rudder; it has since been replaced by an independent electrical system in the newer A330/A340 family. However, the intention here is only to stabilize the aircraft while the computers are restarted in an emergency; landing with only the mechanical systems is not part of the requirements, even though some airlines try it at least once as part of the type rating syllabus—in a simulator, of course.

The five computers are comprised of two ELACs (elevator and aileron computers) and three SECs (spoiler and elevator computers). Notice how ELAC

and SEC overlap in the sense that they can both control the elevators. However, they are distinct in control of ailerons and spoilers but, as we have seen above, these back each other up to some degree. For dissimilarity the ELACs are based on 68010 processors and the SECs on 80186s. These are produced by different companies, and a total of four different architectures are used for the software running on the computers. There are three independent hydraulic systems and two independent electrical buses, as well as three identical air data computers that measure AoA, sideslip and speed, plus a number of attitude parameters. The number three is not coincidental: the five central computers consume the data from all three air data computers. If the data from one of them differs from that from the other two, it is deemed unreliable and ignored. This kind of majority voting is a common strategy for deciding on the correct data values among multiply redundant data sources.

To increase the reliability of each of the five computers, each consists internally of two channels. These are implemented with two identical computers that perform the same computations in parallel: one commanding, the other monitoring. The monitoring computer compares the results of each computation with the other channel's results, and if they diverge beyond a predefined threshold this constitutes a failure of the overall computer. It would then report the problem, stop its work and electrically disconnect itself from the system. The other ELAC, or eventually, one of the SECs, takes over. The two channels use the same hardware architecture but different software.

Let's look at the degradation in functionality that I mentioned above. The flight control system provides three different flight control laws. In accordance with the two main reasons for using fly-by-wire systems, each law provides a particular set of flight characteristics and protections. If all computers are operational, the A320 operates in normal law. The aircraft is flight path-stabilized: the system provides stability in the sense that, with the stick at neutral, the aircraft retains its current flight path; it controls for 1 g, or the absence of any acceleration (this is true only up to a point: if a divergence is large enough pilot input will be required—it's not an autopilot attitude-hold mode). The pilot's stick input governs the first derivative of the flight path. Pulling the stick requests a particular load (g-level). Deflecting the stick to the right rolls the aircraft to the right with a roll rate that is proportional to the

deflection; full deflection results in a 15 degree/second roll rate. Importantly, the mapping from stick deflection to load factor or roll rate is independent of speed, which makes the A320 quite different than a conventional aircraft. The roll behavior changes at 33 degrees bank, however: instead of controlling the roll rate, the stick now controls the roll angle. As with the steering wheel of a car, the stick has to be kept deflected to continue a turn of more than 33 degrees. The intention behind this behavioral change is to make the pilot “force” the aircraft into such a tight turn. Bank angles of more than 67 degrees are not possible—the law protects against overly steep turns. Pilots can also ignore the rudder pedals in normal law, because the law automatically applies the right amount of rudder to fly a coordinated turn for the commanded bank angle. It also does not allow flying with more than 30 degrees of nose-up pitch or 15 degrees nose-down, protects against overspeed and limits the AoA to what the wings can do. The system is quite invasive: it will automatically add power and/or push the nose down to keep the aircraft safe. Finally, it also protects against over-stressing the airframe: it won’t permit more than +2.5 g or less than –1.0 g.

If subsystems start to fail, the flight control system performs a stepwise degradation to simpler laws. Let’s say that both ELACs fail and the SECs take over. Since these have less computing power, simpler control laws are used. If the air data computers fail to provide data reliably, then the automatic flight path stabilization can no longer work and the system degrades. The specific rules that govern when which law will be engaged depend on the specific failures, but generally at least two failures are required in either computers, hydraulics or sensor systems for normal law to become unavailable.

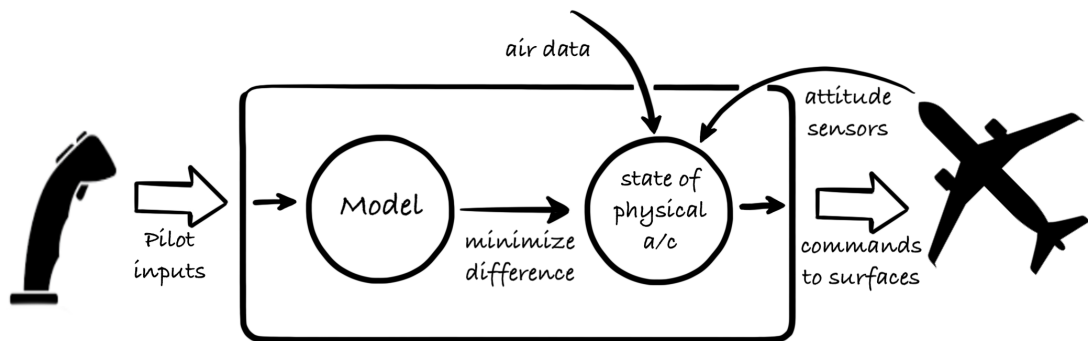
Let’s briefly look at the two simpler laws. The first is called alternate law. In pitch, the stick continues to express a load factor demand, but in roll there is a (virtual) direct stick-to-surface connection; it basically commands a non-stabilized roll rate. There’s no more turn coordination and pilots have to use their feet. There are also fewer protections: over-stress protection remains active but AoA protection is gone, and so is the 67-degree maximum bank angle limit. “Direct” law is the final degradation step: the computers do not modify any of the pilot’s commands, there are no more protections, and it’s back to manual flying in every respect.

There are many more details. For example, normal law has a ground mode that provides a direct stick-to-surface connection. This is to allow pilots to perform checks of the controls and rotate the aircraft during takeoff. There's also a landing mode: at 50 feet above ground the ELACs memorize the pitch attitude; at 30 feet they add a little nose-down to the memorized pitch attitude, which the pilot then compensates by pulling on the stick. This is supposed to make the aircraft feel more conventional during flare, the final part of a landing, when pilots are used to pulling on the stick. Finally there is also an abnormal attitude law: if the aircraft leaves its protected performance envelope, for example because of severe turbulence, abnormal law stabilizes its attitude but without most of the protections. This is designed to make sure that the computers never prevent the pilots from recovering from an abnormal attitude.

Remember our brief discussion of parabolic flight earlier? After injection, the pilots control the aircraft manually to maintain exactly 0 g. I asked chief pilot Eric what they plan to do once they have to replace their A310 with something newer, probably something that has a fly-by-wire control system. He said that they tried parabolic flights with an A380 together with Airbus, "and it worked". I assume they used basic law, because the others would not allow such a maneuver. He also said that, in principle, because Airbus' flight control system works to maintain 1 g when there are no pilot inputs, one would just have to replace 1 g with 0 g in the control loop: "You'd pull up to 50 degrees pitch and then just press a button that engages the 0 g law and the system would automatically control for a 0 g trajectory." He added that one would have to remove and adapt all kinds of protections, but in principle, he suggested, the architecture of the flight control system should allow such a modification with acceptable effort.

So what exactly is a "law"? We have already said that a law governs how the pilot's inputs are translated into attitude changes and which protections are available. But how does it do that? The best way to summarize it is that the flight law contains a mathematical model of the ideal airplane. When the pilot makes a control input, the attitude of the model is changed based on how the aircraft should behave, and the initial deflections of the control sur-

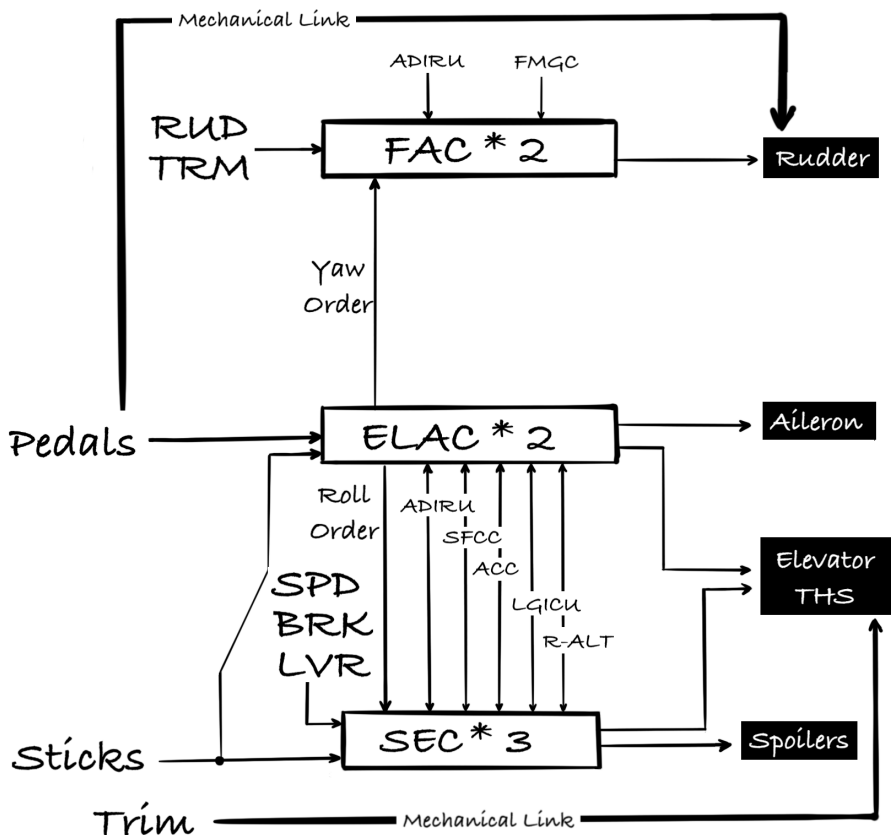
faces are determined by predictions based on that model. There is also a representation of the actual attitude of the aircraft. When the two differ (as they will initially when the pilot makes a control input), the control algorithm tries to align the real aircraft with the model. It uses the available control surfaces to do so—for example, aileron and spoilers if everything works, or just the spoilers if an aileron has failed. This also explains why the system degrades to simpler laws if air data computers or attitude sensors fail: the computer does not have a trusted representation of the aircraft’s current state, so it cannot reliably use the control surfaces to “align” the real world with the model.



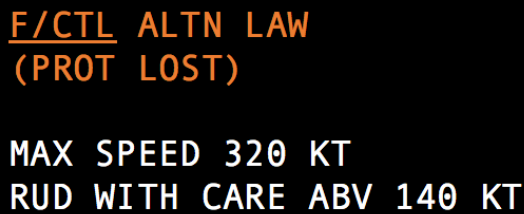
The aircraft has many more computers than ELACs and SECs. There are the flight control data concentrators (FCDCs) that acquire data from the primary computers so that they can be logged and shown to the pilots. There is also the electronic centralized aircraft monitor, or ECAM, a system that monitors the health of aircraft functions and communicates them to pilots; an independent monitoring infrastructure is an additional means of achieving reliability. There is a whole set of computers in the flight management and guidance system (FMGS) which, among other functions, is also home to the autopilots. Two more systems, the flight augmentation computers (FACs), deal with the flying itself: they control yaw damping, turn coordination and rudder trim, and also compute minimum and maximum speeds and handle AoA protection. Because all of their functionality is optional, however, they are not part of the core reliability architecture.

To summarize, the diagram below shows the final architecture. Stick and

pedals feed into the ELACs, which control the ailerons and elevators. They also use the spoilers to help with roll, which is why they send roll orders to the SEC. To fly nicely coordinated turns, the ELACs send yaw orders to the FACs. To build an accurate picture of the aircraft's attitude and flight path that is crucial for the flight laws, the ELACs receive air data and data from the inertial reference unit (ADIRU), slats and flaps (SFCC), accelerometers (ACC), landing gear (LGICU) and the radio altimeter (R-ALT). If the ELACs fail, the SECs take over. The ailerons are then dead, but the aircraft can roll with its spoilers. Note that the SECs cannot pass yaw orders to the FACs, so the rudder becomes inactive. However, spoilers also provide a yawing moment, so the missing rudder is not a huge problem; also, the mechanical linkage is still there. Both the ELACs and the SECs can talk to the elevator and the trimmable horizontal stabilizer (THS).



Even though they are not essential, a failure of both FACs has significant consequences and the flight law will degrade to alternate. The AoA protections are lost and speed is limited due to loss of high-speed protections and the yaw damper. The rudder travel limiter may be stuck in the wrong place, so the rudder must be used with care. During landing, control automatically reverts into direct law, because alternate law does not provide the landing mode mentioned above. Here's what the ECAM will show the pilots:



F/CTL ALTN LAW
(PROT LOST)

MAX SPEED 320 KT
RUD WITH CARE ABV 140 KT

Let me very briefly revisit the discussion about pilots not getting enough manual flying practice. As I said earlier, this discussion concerns higher-level automation such as autopilots and flight management systems more. These are also available in traditionally controlled aircraft. However, there is a connection to fly-by-wire control. The “virtual aircraft” represented by the normal law of the A320 is simpler to fly manually than the unprotected, direct-law physical aircraft. If a pilot who is used to lots of automation and at most flies the virtual aircraft is suddenly forced by system failures to fly manually in direct law, I do think that this might introduce a risk.

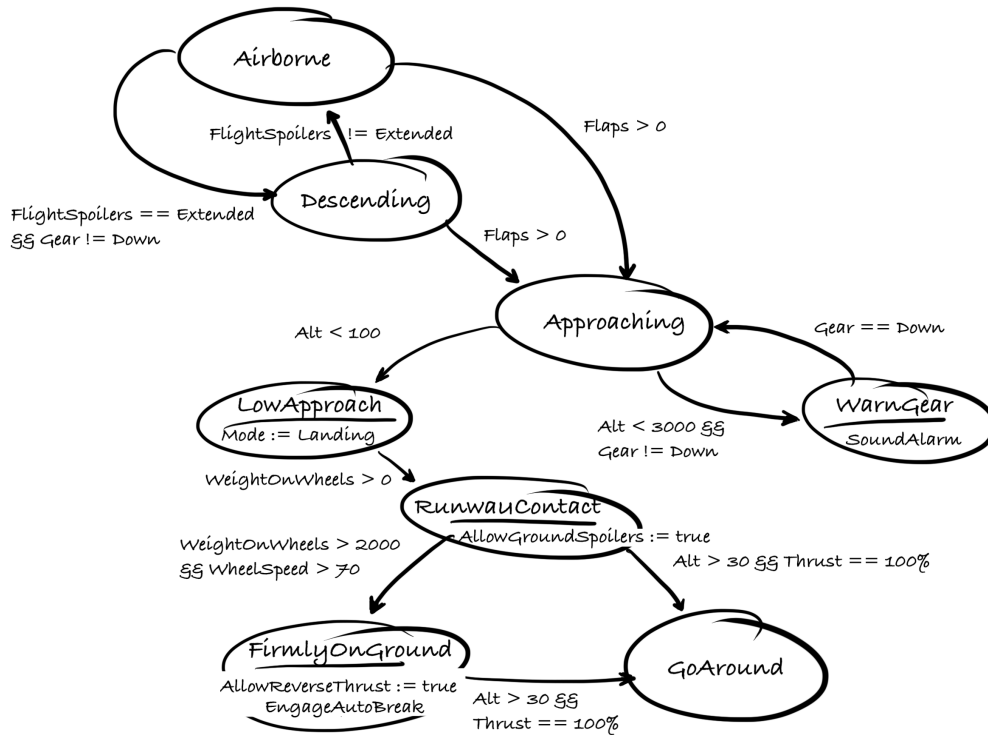
Let's revisit the Swiss cheese model. The loss of an A330 performing flight Air France 447 over the Atlantic, in which 228 people lost their lives, can be attributed to (1) the icing of pitot tubes, which invalidated the airspeed signal, (2) the autopilot switching off and the flight control system switching to alternate law, (3) a misunderstanding of the situation by the pilots, who basically pulled the stick back until the aircraft stalled, then ignored the stall warning, (4) the pilots not sticking to the predefined procedures for loss of airspeed indication, and (5) the problem that the two pilot and copilot's sticks don't move together, so it was hard for the pilot who was not in control of the aircraft to realize that the pilot who was flying it was pulling continuously on his stick. This account of the causes is slightly simplified, but the resulting

accident was definitely a combination of strange, unintuitive and suboptimal system behavior together with the pilots making mistakes. An initially similar problem occurred with an A320 over Spain where two of the three angle-of-attack sensors reported faulty values because of icing—the flight control system followed its algorithm and voted 2:1 against the remaining good one! Because the two sensors were stuck at around 4.5 degrees, the angle-of-attack protection kicked in and tried to lower the nose. Since the sensors still reported an unchanging high AoA, the flight control system pushed more and more, forcing the aircraft into a descent. The pilots, together with technicians on the ground, understood the problem, switched off the air data computers and flew the aircraft home manually in alternate law. The holes in the cheese didn’t align in this case.

~ ~ ~

Safe Software. Let’s conclude our discussion of how to make a fly-by-wire system reliable with a brief look at how one builds reliable software. There are two ingredients: verification and validation. Verification is about ensuring that the software behaves exactly as specified (nothing less, nothing more). For example, ensuring that buffers never overrun if a sender of data stays below its maximum sending rate and the receiver takes data out of the buffer at its minimum guaranteed processing rate is an example of verification. This is very different from validation, which is about ensuring that those behaviors are appropriate with regard to the real world. The problem of detecting the “on ground” state by Lufthansa Flight 2904 was a validation failure.

Consider the following diagram, which represents what software engineers call a “state machine”. This captures various states of the aircraft (the ellipses), as well as the conditions under which the aircraft transitions from one state to another (the arrows). The boxes also contain actions, denoting things that happen when the system enters that particular state. Note that this is just a simple example, not at all an attempt at replicating actual logic from the A320. However, state machines and the verification and validation techniques we discuss below are used in the development of flight control software.



Let's walk through the diagram. Initially we are in the Airborne state, flying straight and level. If we now extend the flight spoilers while keeping the gear up, we go to the Descending state. If the spoilers are retracted (no longer extended) we are back to Airborne. If we extend the flaps in either of these two states, we move to the Approaching state. If we get lower than 3,000 feet above ground with the landing gear not down, we go into a GearWarning state and sound an alarm. The alarm stops when we extend the gear. Below 100 feet we enter the LowApproach state, in which the flight law's Landing mode is activated. Once we detect weight on wheels we enter the state RunwayContact; this allows the pilots to extend the ground spoilers. When we detect a load of more than 2 tons on the wheels and they have spun up to at least 70 knots, we decide we are FirmlyOnGround, engage auto braking and allow the thrust reversers to be used. Finally, we detect a GoAround if altitude increases and the engines are in full thrust.

How can we ensure that this state machine is correct? We essentially have

two tasks. We first have to validate that the state machine expresses the correct behavior—that we actually want the aircraft to behave as the state machine prescribes. Then we have to verify that the software program that implements this machine behaves exactly as the model says. Let’s start with validation. How can we do this?

First we can manually review it. For example, by looking closely at the diagram we might notice that if we do not extend the flaps (maybe they have failed and we cannot extend them) we never enter the Approaching state, so we will not get a gear warning below 3,000 feet! Or we might notice that extending the spoilers might not result in descent, but just in a slowdown (if we pull on the stick enough). However, this manual inspection is really tough once this sort of model becomes realistically complex.

The next best thing we can do is test it. We can “exercise” the model by scripting an execution, checking that the model reacts correctly based on the requirements provided by people who have an understanding of how the system should behave in particular scenarios. The following is an example that assures that we get a gear warning at 2,500 feet.

```
assert state == Airborne
set     Spoilers := 50%
assert state == Descending
set     Alt      := 4000 ft
set     Flaps    := 25°
set     Alt      := 2500 ft
assert SoundAlarm == true
```

Ideally tests are scripted once and then run automatically whenever we change something in the state machine model; this is to ensure we don’t accidentally break previously valid behaviors. But we still have to “manually” think of all relevant test scenarios, and it is easy to forget corner cases. Think of LH 2904: to catch the fault condition, we would have had to write a test that used less than 6 tons of wheel loading. That might be hard to think of, especially if you are the engineer who has just decided that you will always have at least 7 tons, and so a threshold of 6 tons is correct.

Can we do better than test? Yes. For example, we can write down invariants.

These are Boolean conditions that must always be true. Here are two examples; the $X \Rightarrow Y$ arrow represents implication, which says that if X is true, then Y must also be true, otherwise we don't care.

```
invariant !WeightOnBothWheels      => !AllowReverseThrust  
invariant Alt < 3000 && !GearDown => SoundAlarm  
invariant ReverseThrustEngaged     => Alt == 0 && AF[Speed == 0]
```

The first invariant expresses the fact that, unless we have weight on both wheels, we are not allowed to use reverse thrust. The second says that we will get an alarm whenever the altitude is less than 3,000 feet and the gear is up. The third is especially interesting. It says that if reverse thrust is active, the altitude must be 0 (that is, it is only allowed on the ground). It also says that the speed must become zero eventually: $AF(c)$ implies that for all (A) possible executions and branchings of the machine, in some future (F) state the condition c becomes true. AF is an example of a temporal operator, one that lets us express invariants that involve the execution of the state machine over time. There are many others; for example, we could express that, once the reversers are deployed, for all future states they have to stay deployed until (U) speed is zero:

```
ReverseThrustEngaged => A[ReverseThrustEngaged] U[Speed == 0]
```

Now that we have stated these invariants, what can we do with them? One thing we can do is to check them during tests. So whatever test scenario we run, we also check the invariants; if any fail, the test fails. In this way we can hopefully detect unexpected behaviors even though we didn't write a test that specifically checked for them.

However, we can do better than this by using a technique called model checking, which uses software tools to analyze the state machine model and tries to prove that the invariants hold for every possible execution of the machine. If the tool cannot find that proof it shows us an example execution of the machine where an invariant does not hold; this will hopefully help us to

fix the machine. Model checking is interesting, because it frees us from having to manually write tests that cover all possible (and/or relevant) executions of the system. On the other hand, we still have to come up with the invariants. Both testing and model checking rely on complete requirements and on the engineers' correct understanding of them. In practice a combination of reviews, testing and model checking is used, and more. For example, ensuring that tests are written by people other than those who designed the state machine in the first place, to ensure that they throw “nasty” tests at the machine.

Now that we have validated the model, how can we ensure that the actual software, written in C/C++, Ada or Rust, implements this model faithfully? The mapping from the model to the code is often non-trivial because, in addition to behaving identically in terms of the logic, the code also has to be fast enough to fit into the constraints demanded by a realtime control system. Developers also have to care about memory management, buffer sizes, integer overflows, concurrent executions of parts of the program and many other low-level issues that are not relevant for the behavior expressed in the model.

So what can we do? We can for example generate code from the model automatically. Assuming that the model is correct, and assuming the code generator does not make any mistakes, we will get a faithful implementation. But the assumption that the code generator will not make mistakes is a big one. We have moved our problem from verifying a particular model to validating and verifying the code generator. There are tools, however, that have been proven to be correct; Esterel's SCADE is an example. As far as I know SCADE is used for flight control software at Airbus. Another approach is to run the set of tests written for the model against the code as well, to check that the code behaves in the same way. We can also use software tools to extract tests from the model automatically and run them against the code. Finally, there are tools that allow the equivalent of model checking on the level of the program code. We will look further at the relationships of model and implementation code in the Models chapter.

How can modern gliders fly 100s of kilometers, and why do they take water ballast to do it • How can the SR-71 fly at Mach 3 at 80,000 feet • How does it feel to fly in an F-16 fighter jet • How do computers control an A-320 and why is it so hard to fly a helicopter • How do you control 17-ton telescope mounted on a 747, and why would you do that in the first place • How is life on a military survey ship, and how do multibeam sonars map the sea floor • How do you inter-ferometrically combine many telescopes into one • How do you engineer a system to measure gravitational waves • What is it like to stand right under the world's largest optical telescope, as the dome opens, and the milky way reflecting in its giant mirrors • How do you control the LHC's beam • Why are models so important in science and engineering?

Want to know?

Then check out Markus Voelter's book **Once You Start Asking**. It is based on 10 years of reporting on science and engineering for the omega tau podcast. 200,000 words, 160 illustrations and dozens of pictures spread over seven entertaining and sometimes technical chapters:

- Flying and observing with SOFIA
- Charting the Seas with HMS Enterprise
- Gliders and Other Flying Machines
- Detecting Gravitational Waves
- Engineering the Big Telescopes
- Models in Science and Engineering
- The LHC: Big Machines for Very Small Scales.



ONCE
YOU
START
ASKING
.COM

The omega tau Book

