DOI:10.1145/3376127

Nancy Leveson

▸ Peter G. Neumann, Column Editor

Inside Risks

Are You Sure Your Software Will Not Kill Anyone?

Using software to control potentially unsafe systems requires the use of new software and system engineering approaches.

FROM WHAT I have seen, heard, and read, confusion and misinformation abound about software and safety. I have worked in this area for nearly 40 years, starting around the time when computers were beginning to be introduced into the control of safety-critical systems. I want to share what I have learned. Too many incorrect beliefs are being promoted, which are inhibiting progress and, in some cases, unnecessarily costing lives. This column clarifies this topic so that the solutions we propose are more likely to have a significant impact on safety.

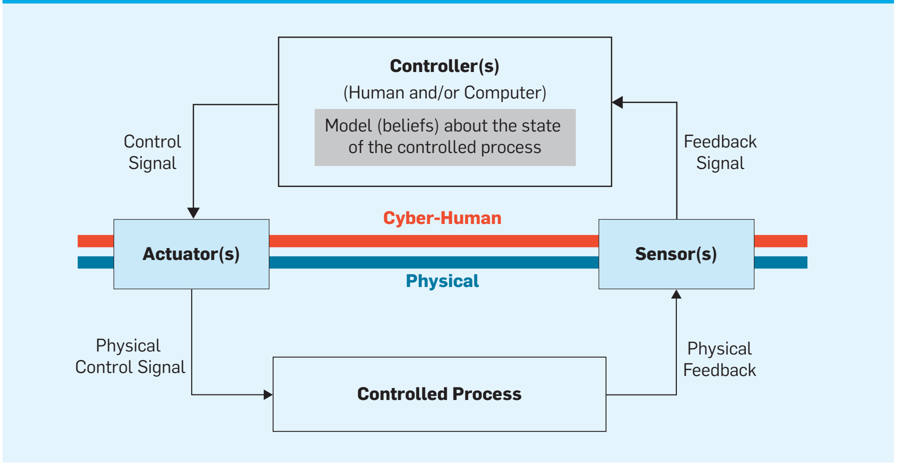
With only a few exceptions, software was not used to directly control safety-critical systems until approximately 1980, although it was used to provide computational power for complex systems, such as spacecraft. Direct control was very limited, but the hesitation has now almost completely disappeared and software is used to control most systems, including physical systems that could involve potentially large and even catastrophic losses.

Originally, "embedded software" was used to denote these new control roles for software, but more recently the term "cyber-physical systems" has come into vogue. The figure here shows a standard cyber-physical control loop. Note that, for some reason, cyber-physical systems usually forget that control can be, and often is, provided by humans. In a little more realistic model (but more complicated than necessary here), there would be two controllers where a human controller is providing control signals to a computer controller. To cover more than the unusual case where there are no human controllers, we should actually talk about "cyber-human-physical" systems. Even so-called "unmanned" air vehicles, for example, usually have a human controller on the ground. A more realistic and complete model is provided in Appendix G of the *STPA Handbook*.[4]

As illustrated in the figure here, a controller (or controllers, which may be human, automated or both) compares the current state of the controlled process with the control goals and sends control signals to an actuator, which in turn may be automated or human. The actuators translate the control signals into physical actions on the controlled process. Sensors provide feedback about the state of the controlled process to the controller so it can determine the state of the controlled system and decide whether further control signals are needed. The actuators and sensors may be software, hardware, physical devices, or humans.

In order to decide on what control actions to provide in order to satisfy its goals (requirements), the controller must have a model (often called a mental model when the controller is human) of the current state of the controlled process. The most common cause of accidents stemming from unsafe controller action is that the model of the controlled process is incorrect: the pilot thinks the aircraft is not in a stall when it is and does not issue a

A cyber-human-physical control loop.



required control action to escape from the stall, the driver does not see the pedestrian and does not brake in time to prevent a collision, the weapon controller thinks that friendly troops are the enemy and initiates friendly fire. The pilot, driver, and weapon controller can be human or computerized, or a combination of both.

Accidents involving computers (and humans) most often occur when their models of the current state of the controller do not match the actual state of the controlled process; the controller issues a control action that is appropriate for a different state but not the one that currently exists. As an example, the software controller thinks the aircraft is in a stall when it is not and issues a control action to escape the nonexistent stall only to inadvertently put the aircraft into a dangerous state.

Starting from this foundation, let's consider some of the most common misconceptions with respect to software and safety.

Misconception 1: Software Itself Can Be Unsafe

Software cannot catch on fire or explode; it is an abstraction. Only physical entities can inflict damage to life and property: physical energy is usually required to inflict physical harm. In the figure in this column, software sends control signals to a physical process, which may have physical effects. Nuclear power plants can release radiation, chemical plants can release toxins, weapon systems can explode or inadvertently target a friendly object, for exsmple. One old model of an accident describes it as uncontrolled energy. Software does not release energy; it simply releases bits, which can be used to send a control signal.

To avoid misconceptions that arise from the term "software safety," sometimes safety engineers speak of "software system safety," to denote the contribution of software behavior to a dangerous process. An alternative conception is to speak of the contribution of software to *system safety*. Either way, by considering software in isolation, without including the controlled physical process, it is not possible to assure anything about the safety of the system the software is controlling.

The Ariane 4 software Inertial Reference System was perfectly safe in that launcher. However, when reused in the Ariane 5, it led to an explosion and loss of a satellite. Many accidents involve reused software.[3] It is not the software that is unsafe, but the entire system controlled by the software.

Misconception 2: Reliable Systems Are Safe; That Is, Reliability and Safety Are Essentially the Same Thing. Reliability Assessment Can Therefore Act as a Proxy for Safety

Reliability and safety are different system properties and sometimes even conflicting. This is true also with respect to the contribution of software to accidents. System components (including software) can operate 100% reliably and accidents may still result, usually from unsafe interactions among the system components. In addition, the larger environment (including social policies and decision making) beyond the system boundaries is important. As a simple, real-world example, consider going out to the middle of a large deserted area, pointing a gun away from oneself, and firing. If there is nobody or nothing in the vicinity, the gun could be considered to be both reliable and safe. Consider, however, doing the same thing in a crowded mall. The gun has not changed, the gun's reliability has not changed, and the action (pulling the trigger) has not changed. But the safety certainly has.

The accompanying sidebar highlights three examples out of hundreds of similar losses.[4] Considering reliability only at the system level (instead of the component level) does not help. Complex systems almost always have many requirements (or goals) while there are constraints on how those goals can be achieved. As an example, a chemical plant may very reliably produce chemicals (the goal or mission of the plant) while at the same time polluting the environment around the plant. The plant may be highly reliable in producing chemicals but not safe. Most safety-critical systems have both mission (non-safety) requirements and safety constraints on how the mission or goals can be achieved. A "system failure" or inability to satisfy its requirements is not equivalent to a hazard or an accident. One exception is if safety is the only goal of the system; however, even for systems such as air traffic control, there are usually non-safety goals such as optimizing throughput in addition to the safety goals.

A common approach to assessing safety is to use probabilistic risk assessment to assess the reliability of the components and then to combine these values to obtain the system reliability. Besides the fact that this assessment ignores accidents that are caused by the interactions of "unfailed" components (see Misconception 3), most of these assessments include only random hardware failures and assume independence between the failures. Therefore, they provide anything close to a real safety assessment when the systems are just hardware and relatively simple. Such systems existed 50+ years ago when these probabilistic risk methods were developed; virtually all systems today (particularly complex ones) contain non-stochastic components including software logic and humans making cognitively complex decisions.

We need to stop pretending that these probabilistic estimates of safety have anything to do with reality, and not base our confidence about safety on them. I have examined hundreds of accident reports in my 40 years in system safety engineering. Virtually every accident involved a system with a probabilistic risk assessment that showed the accident could/would not occur, usually exactly in the way it did happen.

Misconception 3: The Safety of Components in a Complex System Is a Useful Concept; That Is, We Can Model or Analyze the Safety of Software in Isolation from the Entire System Design

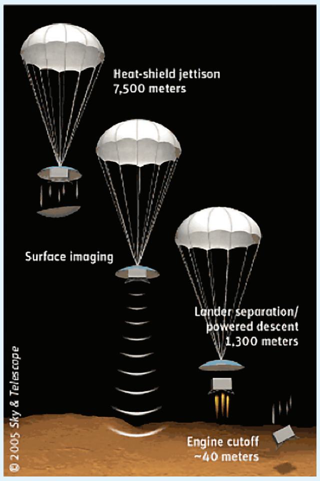
While the components of a more complex system can have hazards (states that can lead to some type of loss), these are usually not of great interest when the component is not the entire system of interest. For example, the valve in a car or an aircraft can have sharp edges that could potentially lead to abrasions or cuts to those handling it. But the more interesting hazards are always at the system level—the sharp corners on the valve do not impact the hazards involved in the role of the valve in the inadvertent release of nuclear radiation from a nuclear power plant or the release of noxious chemicals from a chemical plant (for example).

In other words, safety is primarily a system property and the hazards of interest are system-level hazards. The component's behavior can, of course, contribute to system hazards, but its contribution cannot be determined without considering the behavior of all the system components as a whole. Potentially effective approaches to safety engineering involve identifying the system hazards and then eliminating or, if that is not possible, preventing or mitigating them at the system level. The system hazards can usually be traced down to behavior of the system components, but the reverse is not true. One cannot show that each component is safe in isolation and then use that analysis to conclude the system as a whole will be safe.

Another way of saying this is that a system component failure is not equivalent to a hazard. Component failures can lead to system hazards, but a component failure is not necessary for a hazard to occur. In addition, even if a component failure occurs, it may not be able to contribute to a system hazard. This is simply another way of clarifying misconception #2 concerning the difference between reliability and safety.

Three Examples of Accidents Due to Unsafe Interactions between Systems Components

Some Navy aircraft were ferrying missiles from one point to another. One pilot executed a planned test by aiming at the aircraft in front (as he had been told to do) and firing a dummy missile. Apparently, nobody knew that the "smart" software was designed to substitute a different missile if the one that was commanded to be fired was not in a good position. In this case, there was an antenna between the dummy missile and the target, so the software decided to fire a live missile located in a different (better) position instead. What aircraft component(s) failed here?



This loss involved the Mars Polar Lander. It is necessary to slow the spacecraft down to land safely. Ways to do this include using the Martian atmosphere, a parachute and descent engines (controlled by software). As soon as the spacecraft lands, the software must immediately shut down the descent engines to avoid damage to the spacecraft. Some very sensitive sensors on the landing legs provide this information. But it turned out that noise (sensor signals) is generated when the legs are deployed. This expected behavior was not in the software requirements. Perhaps it was not included because the software was not supposed to be operating at this time, but the software engineers decided to start early to even out the load on the processor. The software thought the spacecraft had landed and shut down the descent engines while the spacecraft was still 40 meters about the planet surface. Which spacecraft components failed here?

It is dangerous for an aircraft's thrust reversers (which are used to slow the aircraft after it has touched down) to be activated when the aircraft is still in the air. Protection is designed into the software to prevent a human pilot from erroneously activating the thrust reversers when the aircraft is not on the ground. Without going into the details, some of the clues for the software to determine the plane has landed are weight on wheels and wheel spinning rate, which for a variety of reasons did not hold in this case. For example, the runway was very wet and the wheels hydroplaned. As a result, the pilots could not activate the thrust reversers and the aircraft ran off the end of the runway into a small hill. What aircraft components failed here?

Misconception #4: Software Can Be Shown to Be Safe by Testing, Simulation, or Standard Formal Verification

Testing: Exhaustive testing of software is impossible. The problem can be explained by examining what "exhaustive" might mean in the domain of software testing:

- Inputs: The domain of possible

inputs to a software system includes both valid and invalid inputs, potential time validity of inputs (an input may be valid at a certain time but not at other times), and all the possible sequences of inputs when the design includes history (which is almost all software). This domain is too large to cover any but a very small fraction of the possible inputs in a realistic timeframe.

▸ System states: Like the number of potential inputs, the number of states in these systems is enormous. For example, TCAS—an aircraft collision avoidance system—was estimated to have 10^{40} possible states.[5] Note that collision avoidance is only one small part of the automation that will be required to implement autonomous (and even non-autonomous) vehicles.

▸ Coverage of the software design: Taking a simple measure of coverage like "all the paths through the software have been executed at least once during testing" involves enormous and impractical amounts of testing time and does not even guarantee correctness, let alone safety.

▸ Execution environments: In addition to the problems listed so far, the execution environment becomes significant when the software outputs are related to real-world states (the controlled process *and* its environment) that may change frequently, such as weather, temperature, altitude, pressure, and so on. The environment includes the social policies under which the system is used.

In addition, as seen in the much-repeated Dijkstra quote, testing can show only the presence of errors, not their absence.

Finally, and perhaps most important, even if we could exhaustively test the software, virtually all accidents involving software stem from unsafe requirements.[2,6] Testing can show only the consistency of the software with the requirements, not whether the requirements are flawed. While testing is important for any system, including software, it cannot be used as a measure or validation of acceptable safety. Moving this consistency analysis to a higher level (validation) only shifts the problem, but does not solve it.

Simulation: All simulation depends on assumptions about the environment in which the system will execute. Autonomous cars have now been subjected to billions of cases in simulators, and have still been involved in accidents as soon as they are used on real roads. The problems described for testing apply here, but the larger problem is that accidents occur when the assumptions used in development and in the simulation do not hold. Another way of saying this is that accidents occur because of what engineers call "unknown unknowns" in engineering design. We have no way to determine what the unknown unknowns are. Therefore, simulation can show only that we have handled the things we thought of, not the ones we did not think about, assumed were impossible, or unintentionally left out of the simulation environment.

System and software requirements development are necessarily a system engineering problem, not a software engineering problem.

Formal verification: Virtually all accidents involving software stem from unsafe requirements, not implementation errors. Of course, it is possible that errors in the implementation of safe requirements could lead to an accident; however, in the hundreds of software-related accidents I have seen over 40 years, none have involved erroneous implementation of correct, complete, and safe requirements. When I look at accidents where it is claimed the implemented software logic has led to the loss, I always find the software logic flaws stem from a lack of adequate requirements. Of the three examples shown in the sidebar in this column, none of these accidents (nor the hundreds of others that I have seen) would have been prevented using formal verification methods. Formal verification (or even formal validation) can show only the consistency of two formal models. Complete discrete mathematical models do not exist of complex physical systems, that is, the controlled process shown in the figure in this column.

Conclusion

All of this leads to the conclusion that the most effective approach to dealing with safety of computer-controlled systems is to focus on creating the safety-related requirements. System and software requirements development are necessarily a system engineering problem, not a software engineering problem. The solution is definitely not in building a software architecture (design) and generating the requirements later, as has been surprisingly suggested by some computer scientists.[7]

Some features of a potential solution can be described. It will likely involve using a model or definition of the system. Standard physical or logical connection models will not help. For most such models, analysis can identify only component failures. In some, it might be possible to identify component failures leading to hazards, but this is the easy part of the problem and omits software and humans. Also, to be most effective, the model should include controllers that are humans and organizations along with social controls. Most interesting systems today are sociotechnical.

Using a functional control model, analysis tools can be developed to analyze the safety of complex systems. Information on an approach that is being used successfully on the most complex systems being developed today can be found in *Engineering a Safer World*[1] and on the related website http://psas.scripts.mit.edu/home/. C

References

1. Leveson, N.G. *Engineering a Safer World.* MIT Press, 2012.
2. Leveson, N.G. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.
3. Leveson, N.G. The role of software in spacecraft accidents. *AIAA Journal of Spacecraft and Rockets 41*, 4 (July 2004).
4. Leveson, N.G. and Thomas, J.P. *STPA Handbook* (2018); http://psas.scripts.mit.edu/home/materials/
5. Leveson, N.G. et al. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering SE-20*, 9 (Sept. 1994).
6. Lutz, R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the International Conference on Software Requirements.* IEEE (Jan. 1992).
7. National Research Council. *Software for Dependable Systems*, 2007.

Nancy Leveson (leveson@mit.edu) is a professor of Aeronautics and Astronautics at the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA.