Resilient Mixed-Criticality Systems

Dr. Lui Sha
University of Illinois at Urbana-Champaign

Most complex cyber-physical systems (CPSs) are mixed-criticality systems that have to be resilient against software design faults, hardware failures, and physical hazards under software control. This article reviews useful design principles and architecture patterns for the development of such systems.

Complex CPSs are typically mixed-criticality systems. They have to be resilient against not only faults and failures in a cyber subsystem but also hazards in a physical subsystem (plant, or device) under software control. Consider the following examples:

Controlling Human Errors and Hazards

After a major surgery, the patient is allowed to operate an infusion pump (patient-controlled analgesia [PCA]) with potentially lethal painkillers such as morphine sulfate. When pain is severe, the patient can push a button to get more pain-relieving medication. This is an example of a safety-critical device controlled by an error-prone operator (the patient). Nevertheless, the PCA system as a whole needs to be certifiably safe in spite of mistakes made by the patient. To solve this problem, medical instruments (sensors) are used to monitor the vital signs. An important element to the sensors is this: A safe dosage, with respect to the patient's conditions, will be delivered for a fixed duration only if all vital signs are within thresholds. Otherwise, an alarm sounds and the infusion stops.

In this example, the cyber subsystem is the computing hardware and software, the *plant* is the patient, the infusion pump is the safety-critical *actuator device*, and the vital signs are the states of the device (or plant) being monitored by sensors. Finally, a CPS is said to be certifiably safe if we can verify that the plant can remain in a safe state (the pain-killer concentration in patient's blood is below a dangerous threshold) with respect to a given set of internal system faults and external safety hazards including a patient's incorrect commands, power failure, and the loss of vital signs signals due to sensor and/or connection failures.

Dangers of Implicit Assumptions and the Need for Both Worst and Average Case Analysis

The safety certification procedure includes the assumptions and specifications of the operational environment, the set of devices and their configurations, the software, and the faults and hazards model. Note that a common cause of system failures in the field is that environmental assumptions embedded in software are implicit.

For example, the Ariane 5 rocket (also known as Flight 501) reused Ariane 4's software, which had correctly assumed that the rocket's (Ariane 4's) horizontal velocity could not overflow a 16-bit variable. Unfortunately, this was not true for Ariane 5 and led to its explosion during its maiden flight [1]. Making assumptions explicit and preferably machine-checkable is an important aspect of building resilient systems. In the development of a system of systems, the new integrated system typically contains many reused subsystems with implicit assumptions embedded in the software.

In addition to safety, the manufacturer of a resilient system must demonstrate the effectiveness of the system under nominal operational conditions. Note that safety is a worst-case analysis, while effectiveness is an average-case analysis. For example, if all components work normally, the PCA pump should deliver the painkiller according to the prescription. Furthermore, it should never overdose the patient even if the patient pushes the deliver button too many times, sensors fail and/or disconnect, and/or there is power failure.

Impractical Correctness

In a typical flight-control system, the autopilot is classified at DO-178B, Level A—the highest safety-critical level—while the flight guidance system (FGS), because of its complexity, is only certified to Level C [2]. Nevertheless, the Level C guidance system issues commands to steer the Level A autopilot. This is an example of safely using a component whose correctness is impractical to verify under current technologies. The overall flight control has to be certified to Level A again.

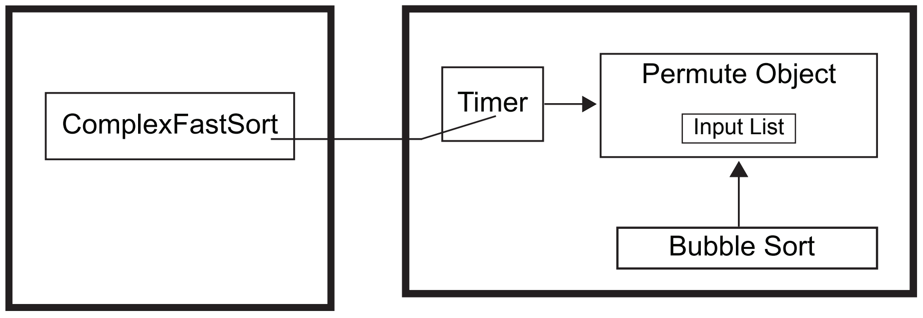
To solve this problem, the control authority of the FGS is first constrained so that the dynamics of the airplane cannot be changed abruptly. This gives the pilot enough time to detect the problem(s) and to take control in time. In addition, a Level A monitor can be used to 1) monitor stability margin in the control of the plane, and to 2) monitor if the plane closely follows the flight path. If any one of the thresholds is violated, an alarm will be sounded and the control is transferred to the pilot.

The following section reviews useful architecture patterns and tools to build resilient systems against software faults and hazards in the physical plants under software control. We (that is, organizations using these methods) begin with software design and/or implementation faults.

Architecture Patterns for Resiliency

Logical complexity is a major driver of software defects. I begin with a simple example to illustrate the idea of "using simplicity to control complexity" to build resilient applications [3]. Consider the problem of sorting. In sorting, the *safety* property is to sort items correctly. The *effectiveness* property is to sort them fast. Suppose that we could formally verify a Bubble Sort program but were unable to verify a ComplexFastSort program. Can we *safely* use the unverified ComplexFastSort for *effectiveness*? Yes, we can.

Figure 1: *Always Correct Sorting System*



As illustrated in Figure 1 (on the previous page), to guard against all possible faults of ComplexFastSort, these two programs are put into two virtual machines. In addition, a verified object called *permute* is developed that will: 1) allow ComplexFastSort to perform all the list operations that are useful in sorting related operations, but not to modify, add, or delete any list item; and 2) check in linear time that the output of ComplexFastSort is indeed sorted. Finally, a timer is set based on the promised speed of ComplexFastSort that is supposed to be faster than the Bubble Sort. If ComplexFastSort does finish in time and the answer checked is correct, then the result is given as output; if it does not finish in time or does so with an incorrect answer, then Bubble Sort sorts the data items. Note that if ComplexFastSort works with time complexity *n log(n)*, the system has the same time complexity *n log(n)*. That is, the lion's share of the effectiveness offered by unverified ComplexFastSort is captured with a small amount overhead when ComplexFastSort works. In addition, we guarantee the safety (items sorted correctly) with respect to the given set of specified safety hazards and faults, namely arbitrary application software errors. So far, there is no protection against virtual machine and/or hardware failures. Such limitations must be noted explicitly. If the application requires the tolerance of hardware failures, then fault-tolerant hardware must be used.

The moral of this story is that we can safely exploit the features and performance of complex components, even if it may have residual defects, as long as we can guarantee the critical properties by simple software and an appropriate architecture pattern. This is the idea of using simplicity to control complexity, which is the guiding principle of building resilient mixed-criticality systems. We want an architecture that can safely utilize the features and performance of lower-criticality components that are impractical to fully verify.

Checking the correctness of an output before using it—such as in the sorting example—belongs to a fault-tolerant approach known as a recovery block [4]. However, in CPS applications, it is often not possible to determine if every command from a complex controller is correct (meeting the specifications).

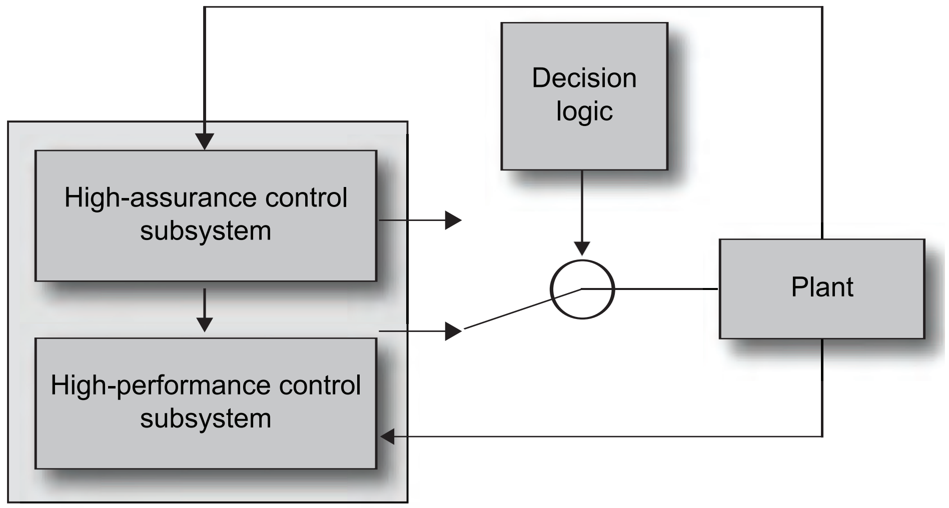
Simplex Architectures

A Simplex Architecture [3] is an architecture pattern for resilient control systems. As illustrated in Figure 2, a Simplex Architecture consists of 1) a safety core with a simple and verifiable high assurance controller and decision logic, and 2) a complex high performance system that cannot be fully verified. There are many failure modes of software. To protect the safety core, it should be run in a different real-time virtual machine.

To guard against real-time operating systems failures and/or security attacks that may breach the firewalls, we can put the safety core into a field programmable gate array (FPGA), a programmable hardware device. For protection purposes, FPGA devices should not be allowed to be reprogrammed during runtime. To ensure the correctness of the FPGA programming, we first perform model checking on the safety core design and then directly generate very high-level hardware description code to program the FPGA.

As shown in Figure 3, this hardware and software co-design approach is known as System-Level Simplex Architecture [5], which was developed for the design of a prototype pacemaker for patients with heart diseases. Pacemakers are also mixed-criticality systems. The safety core is a simple timer for rest rate pacing. If heartbeat is detected, the timer is reset. Otherwise, it sends out a pulse. This simple safety core can be done directly in hardware but it must be safely interfaced with microprocessor-based adaptive pacing, which will pace the heart faster if built-in sensor and motion detection software detects that a patient is exercising. Additional effectiveness features may include the detection and storage of the most important abnormal heartbeats. The rest rate pacing must work even if the microprocessor and its software fail. This is an example of discrete control of a *plant* (in this case, a human heart). The following will illustrate how a Simplex Architecture handles incorrect control commands from the complex high-performance controller for continuous dynamics.

Figure 2: *Simplex Architecture*


Decision logic
High-assurance control subsystem
High-performance control subsystem
Plant

Operational Constraints

In the operation of a plant with continuous dynamics, there is a set of state constraints called operational constraints that represent the safety, device physical limitations, environmental, and other operational requirements. Consider the example of controlling an inverted pendulum mounted on a cart that runs on a track (see Figure 4). The controller must actively move the cart left or right to keep the rod balanced in the upright position. The safety constraint is that it cannot fall down. That is, the angle of the rod must be always less than 90 degrees from the upright position. Device constraints include the length of the track and the limited torque of the motor that runs the cart. Intuitively—to keep the pendulum balanced near the center of the track—the angle should be kept so it doesn't deviate too far from the upright position, and the cart doesn't veer too far from the center of the track. In addition, we need to keep the angular velocity and cart velocity limited. Otherwise, the cart may hit the end of the track, or the rod may fall down with too large of an angle and too large of an angular velocity, such that the inverted pendulum mounted on the cart becomes impossible to keep from falling down.

In control theory, this intuition is represented by the notion of a stability envelope within all of the operational constraints. This envelope represents a subset of the plant states in control of the pendulum. They are: angle, angular velocity, track position, and track velocity, within which the controller can keep the rod upright without violating any of the operational constraints. This envelope is a function of the plant model, the controller design, and the operational constraints. It can be computed by following the steps outlined next.

The operational constraints are represented as a normalized polytope in the N-dimensional state space of the system under control (as shown in Figure 5). Each line on the boundary represents a constraint. The states inside the polytope are called *admissible states* because they obey the operational constraints. We must ensure that the system states are always admissible. This means that we must be able to: 1) take the control away from a faulty control subsystem and give it to the high assurance control subsystem before the system state becomes inadmissible; 2) ensure that the system is controllable by the high assurance control subsystem after the switch; and 3) keep the future trajectory of the system state, after the switch, within the set of admissible states. Note that we cannot use the boundary of the polytope as the switching rule just as we cannot stop an out-of-control car, inches from a wall, from colliding with it. Physical systems, just like a moving car, have inertia.

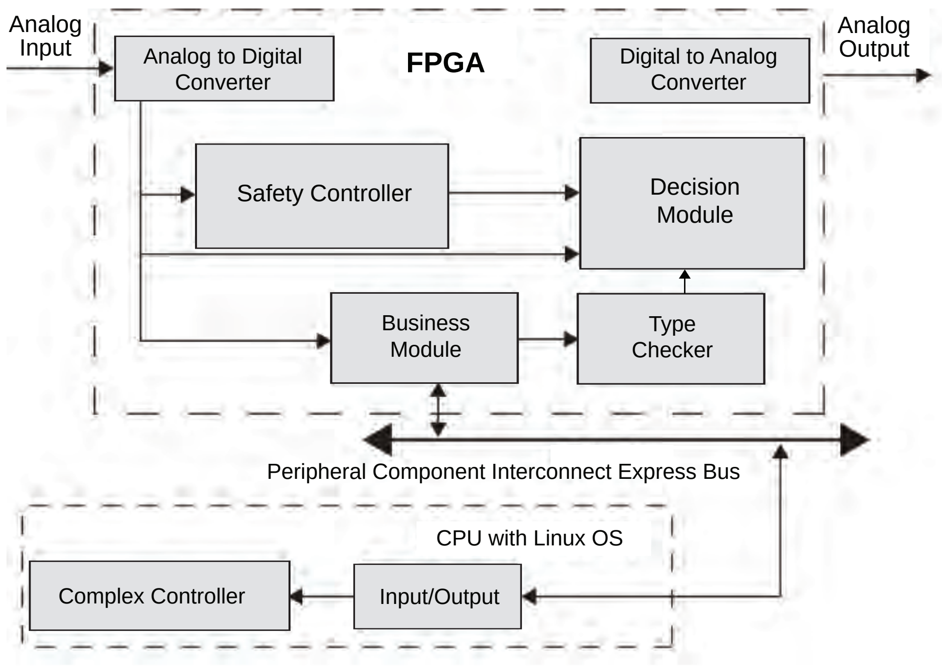


Figure 3: *System-Level Simplex Architecture*

The Recovery Region

A subset of the admissible states that satisfies these three conditions is called a recovery region. The recovery region is represented by a Lyapunov function[1] within the admissible states. Geometrically, a Lyapunov function defines an N-dimensional ellipsoid in the N-dimensional system state space. The boundary of this ellipsoid corresponds to the system's stability envelope. A two-dimensional example is illustrated in Figure 5. A Lyapunov function has the important property that as long as the system state is inside the ellipsoid associated with a controller, the system states under that controller will stay within the ellipsoid and converge to the set point. The largest ellipsoid inside a polytope can be found by using the Linear Matrix Inequality method [6]. The inner ellipsoid is the recovery region used for operation. The shortest distance between the outer and inner ellipsoids is the stability margin. The stability margin allows us to compensate for approximation errors in the plant model, measurement errors in sensing, actuation errors during operation, and disturbance to the plant (e.g., such as wind gusts against an airplane in a storm).

Controllers

During runtime, the plant is normally under the control of a high-performance-control subsystem. The high-assurance controller is a simple and well-understood classical controller. It is executing in parallel to the high-performance controller (HPC). A typical design is to run the two controllers at the same rate, for example, at 100 hertz. Schedulability analysis is performed to ensure that both can finish before the end of the period. As a further precaution, the safety controller is run first. Should the HPC not finish by its deadline, it will be terminated and the command from the safety controller is used. Otherwise, for each command from the HPC, the decision logic estimates the next state if the command was used. If the next state is within the recovery region, the command will be executed. Otherwise, the high-assurance controller takes over and the HPC is terminated (a small number of restarts are typically permitted). In addition, the operator may terminate the HPC for other reasons such as certain features not being suitable for the current operation.

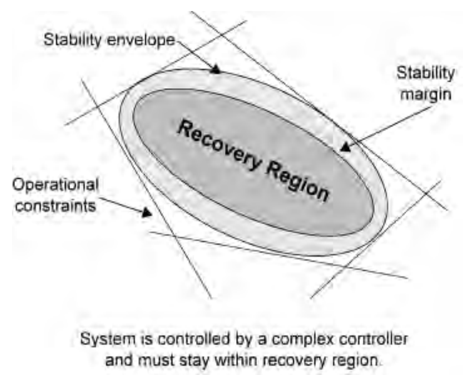
Switching from one controller to the other (hybrid control) may introduce transient errors in the control. A common example is the transient *jump* of a car's velocity when the transmission shifts gears. In the stability analysis, such transient errors must add to the fault and hazard model. That is, in the design of stability margin, it is assumed that when the plant state is at the boundary of the inner-recovery region, the HPC fails and the system switches to the safety controller and the control error, due to switching, reaches its maximal value. As well, the plant model approximation error is maximal, and the actuation errors and external environment disturbances such as wind gusts against the plane are also maximal. In a storm, for example, wind gusts reach a maximal value according to a storm model. The safety controller and the stability margin are designed to accommodate the worst-case scenarios with respect to the fault and hazard model. As a result, a Simplex Architecture tolerates concurrent software faults and disturbance to the physical plant.

A Real-World Example

A noteworthy example of using simplicity to control complexity is the flight control system of the Boeing 777 [7]. It uses triple-triple redundancy for hardware reliability. At the software application level, it uses two controllers. The sophisticated control software, specifically developed for the Boeing 777, is the normal controller because it has many new effectiveness fea-

Figure 4: *An Inverted Pendulum*

Figure 5: *Recovery Region*


Stability envelope
Stability margin
Recovery Region
Operational constraints
System is controlled by a complex controller and must stay within recovery region.

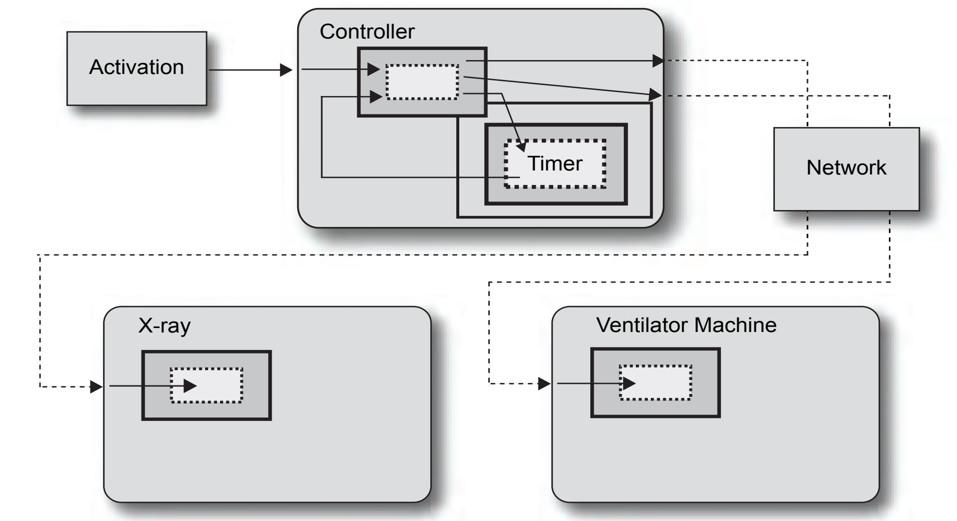


Figure 6: *Configuration 1 of Ventilator Machine With X-ray Machine and Controller*

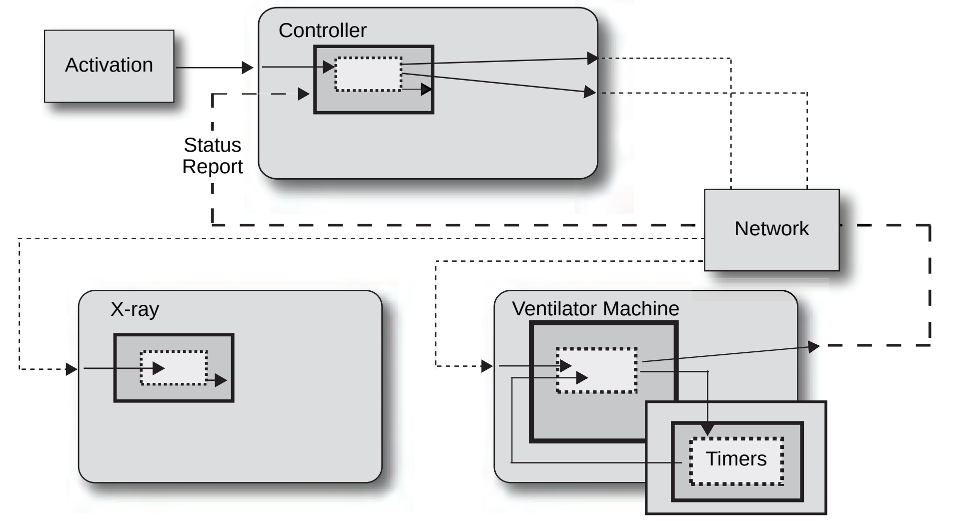


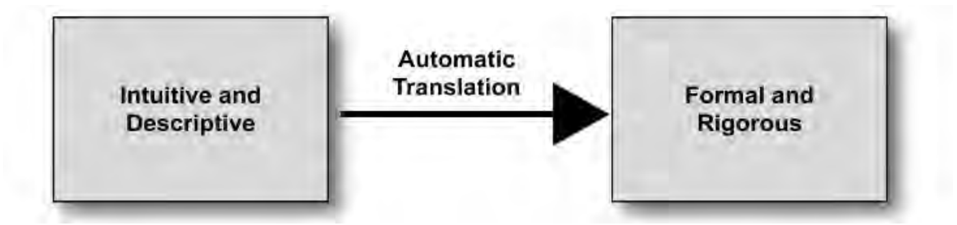
Figure 7: *Configuration 2 of Ventilator Machine With X-ray Machine and Controller*

tures such as highly effective automatic stabilization against wind gusts, advanced fuel-savings, and reduced wear-and-tear to mechanical actuators.

As a result, the 777 controller software is much larger and complex than the 747 controller software. The secondary controller is based on the control laws developed for the Boeing 747, which have been used for decades. It is a mature technology: simple, reliable, and well understood. It is a simple component since it has low residual complexity[2]. It should be noted that the 777 control software was certified to meet safety-critical requirements.

The use of the simpler 747 controller-based software as backup is a precautionary measure for added reliability. This is a best practice because it is uncertain if the process-oriented DO-178B can remain effective when used with increasingly complex software that cannot be exhaustively tested. And while formal model-checking technologies can be scaled up to practice systems and are effective in detecting many types of software defects in a design, it is not formal proof of software meeting all of the specifications—nor is it a verification of the software implementation that flies an airplane.

Figure 8: *AADL to Real-Time Maude Translation*



Formalized Architecture Patterns

Since architecture patterns often need to be adapted for new application requirements, we need to not only verify a collection of commonly used architectural patterns, but also provide computer-aided verification for the adaptation of architectural patterns. Furthermore, model-based approaches are the most common way of capturing architectural designs and architectural patterns; it is important to provide formal verification support for architectural patterns expressed in software modeling languages. The following example illustrates the use of:

- Complexity control architectures and design rules for a medical system.
- A formalized SAE International Architecture Analysis and Design Language (AADL) [8] subset to specify these architectures. AADL is a standard architecture analysis and description language.
- AADL models automatically transformed into algebraic expressions in Real-Time Maude [9] for formal analysis. Real-Time Maude is a model-checking language that supports the checking of hard real-time constraints in addition to temporal logic expressions.

Prevention Through Automation

One example comes from the Anesthesia Patient Safety Foundation:

> A 32-year-old woman was having a laparoscopic cholecystectomy (surgical removal of the gall bladder) performed under general anesthesia. During that procedure and at the surgeon's request, a plain film X-ray was shot during a cholangiogram. The anesthesiologist stopped the ventilator for the X-ray. The X-ray technician was unable to remove the film because of its position beneath the table. The anesthesiologist attempted to help the technician, but found it difficult because the gears on the table had jammed. Finally, the X-ray was removed, and the surgical procedure recommenced. At some point, the anesthesiologist glanced at the EKG and noticed severe bradycardia. He realized he had never restarted the ventilator. This patient ultimately died. [10]

This accident could have been prevented by automation. However, there are two candidate configurations:

- **Configuration 1.** As illustrated in Figure 6, the X-ray machine and ventila-

tor are networked together with a control station. The control station could command the ventilation to pause, the X-ray machine to take a picture, and then command the ventilator to resume. In addition, two watchdog timers are added to the control station. The first one limits the maximum duration of each pause. The second one ensures that pauses are separated by the minimum duration. Both of them are configuration time constants set by medical personnel. However, such a design is unacceptable because if either the network or the control station fail—after commanding the ventilator to pause—the ventilator will be stuck at the pause state. This is known as *dependency inversion*: The safety-critical component ventilator depends on less critical components, such as the network and operator console. Potential dependency inversion is easily detected by automated analysis of AADL design.
- **Configuration 2.** As illustrated in Figure 7, a better design is to put these two timers inside the ventilator. From an architecture perspective, this design minimizes the safety dependency tree into a single node: the ventilator. Under this design, as long as the ventilator is verifiably safe, the overall system is safe in spite of the faults and failures in the network, the command station, and the X-ray machine. From a safety perspective, we can now safely integrate the ventilator into different networks with different but interoperable consoles and X-ray machines without recertifying the safety of the system. This is because the network, X-ray machine, and console are not part of the safety dependency tree.

From the perspective of the Simplex Architecture in Configuration 2, the ventilator is required to be verifiably safe. Once this is done, it can safely collaborate with non-safety critical devices such as the network and a command station. The command station and network should be industrial grade, not certifiably safe, because certifying the operating system and the network is prohibitively expensive. Furthermore, if they were certified, any change in the operating system and/or network would trigger recertification. As well, any non-safety critical device or network information flows connected with this certified network would trigger recertification. Minimizing the use of certifiably safe components—especially the infrastructure components such as the operating system and/or the network—is critical to the economics of medical device networks.

Under the Simplex Architecture, non-safety critical devices can be added, modified, and replaced without jeopardizing safety invariance—provided that architecture design rules are followed. This is done by ensuring that the safety invariants are satisfied by the set of safety-critical components. In this example, the safety invariants of the ventilator are the limit on the maximal duration of each pause and the limit on the minimal duration of separation between pauses. These invariants are specified by means of a configuration time constant set by medical personnel and enforced by the two timers at runtime. Assuming that medical personnel set the constants correctly and the timers embedded in the ventilator design work, the ventilator is safe for all possible inputs from the command station because the timeouts are not a function of inputs from the commands.

The ventilator pause is instantiated from the command station and the command goes through the network. Thus, we say that the architecture *employs* the network, X-ray, and command station, but the safety does NOT depend on them. The idea of *employ but not depend* is a key principle of the Simplex Architecture, which minimizes the use of safety-critical components while maximizing the safe utilization of non-critical components. When critical components employ but do not depend on less critical components, the system safety dependency tree is defined as *well-formed*. Otherwise, it is defined as a (safety) *dependency inversion*.

Checking to see if a candidate configuration is well-formed is done by first developing a model of the composition in AADL with a behavior specification. The AADL model is then translated into Real-Time Maude (as illustrated in Figure 8) using its rewriting logic semantics. The fault model is a specification of possible incorrect state transitions. Using the Real-Time Maude models of faulty transitions in unverified components and systems, we are able to verify (by model-checking) that the AADL model of the ventilator operation satisfies the two safety invariants on maximum pause time and on minimum time between pauses and is, therefore, verifiably safe for such invariants. Furthermore, the effectiveness of the system (liveness property)—wherein the X-ray will be taken during the pause of the ventilator in the absence of faults—was also verified.

Conclusion

The convergence of sensing, control, communication, and coordination in CPS—such as with modern airplanes, power grids, transportation systems, and medical device networks—poses an enormous challenge because of its complexity. Work in all of the areas mentioned in this article is certainly relevant and useful. However, to address the hard challenges of CPS system design, the focus is on a synergistic combination of specific technologies to support the model-based design of highly reliable CPS systems. These combined technologies include: architectural patterns, fault-tolerant techniques, model-based software engineering, object-based formal specification, and the verification of real-time systems.◆

Software Defense Application

Many defense systems are mixed criticality systems with a high level of complexity. The reduced complexity architecture patterns provide an effective approach to address this challenge.

Acknowledgements

The works described in this article are sponsored in part by the Office of Naval Research, the National Science Foundation, Lockheed Martin, Rockwell Collins, and the SEI. Many have contributed to this work, and I thank all of our collaborators in developing these ideas and case studies, including Artur Boronat, Darren Cofer, Peter Feiler, Steve Miller, Peter Ölveczky, Joe Hendrix, Minyoug Nam, and Xiaokang Qiu.

References

1. "Ariane 5 Flight 501." *Wikipedia* <http://en.wikipedia.org/wiki/Ariane_5_Flight_501>.
2. Tribble, Alan C., and Steven P. Miller *Software Safety Analysis of a Flight Guidance System* <http://shemesh.larc.nasa.gov/fm/papers/Tribble-SW-Safety-FGS-DASC.pdf>.
3. Sha, Lui. "Using Simplicity to Control Complexity." *IEEE Software*. July/Aug. 2001 <https://agora.cs.illinois.edu/download/attachments/10581/IEEESoftware.pdf>.
4. Tyrrell, A.M. *Recovery Blocks and Algorithm-Based Fault Tolerance*. Proc. of the 22nd EU-ROMICRO Conference. Prague, Czech Republic: 2-5 Sept. 1996.
5. Bak, Stanley, et al. *The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety*. Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium. San Francisco: 13-16 Apr. 2009.
6. Boyd, S., et al. "Linear Matrix Inequality in Systems and Control Theory." *Studies in Applied Mathematics*. 1994.
7. Yeh, Y.C. *Dependability of the 777 Primary Flight Control System*. Proc. of the Dependable Computing for Critical

Applications Conference. Los Alamitos, CA: 1995.

8. SEI. "Model-Based Engineering with SAE AADL." 2009 <www.sei.cmu.edu/products/courses/p52.html>.
9. Ölveczky, Peter C., and Jose Meseguer. "Semantics and Pragmatics of Real-Time Maude." *Higher-Order and Symbolic Computation* 20(1-2): 161-196 (June 2007).
10. Lofsky, Ann S. "Turn Your Alarms On!" *ASPF Newsletter* 19.4:43. Winter 2004-2005 <www.apsf.org/assets/documents/winter2004.pdf>.

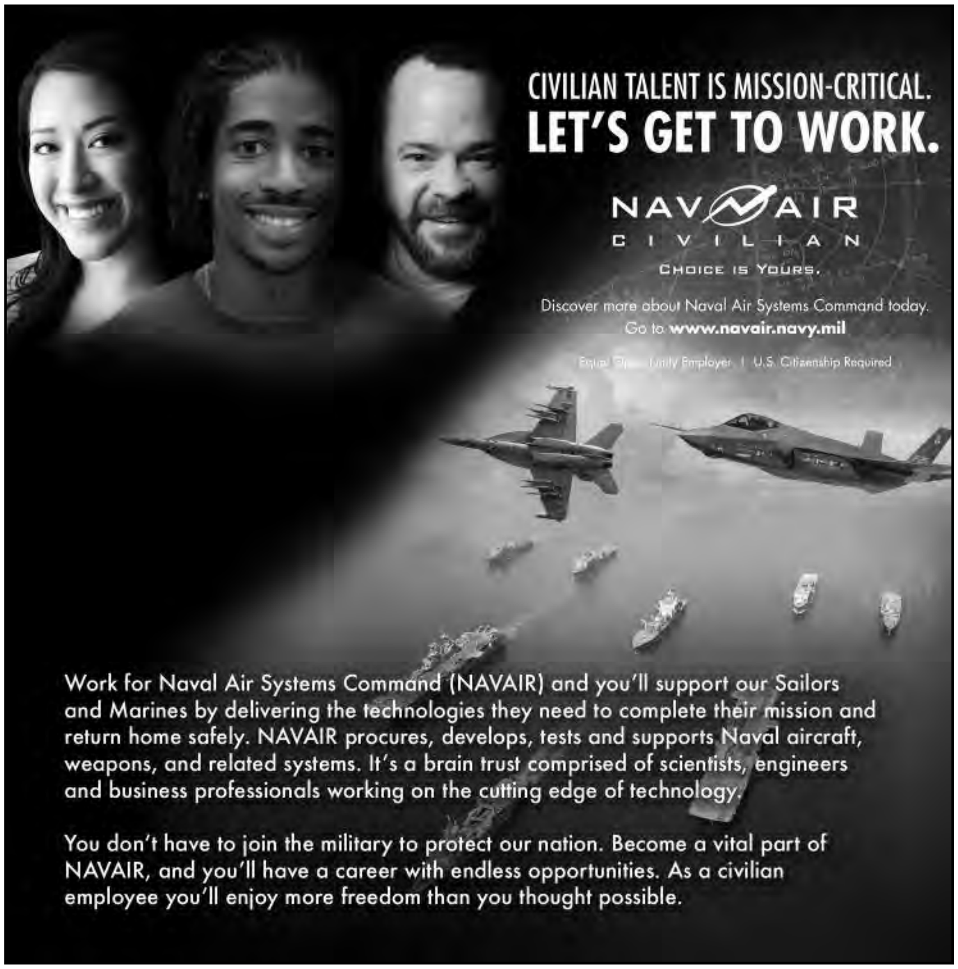
Notes

1. The Lyapunov function is a sufficient but not necessary condition to improve the stability of an equilibrium in an autonomous system. For details, see <http://mathworld.wolfram.com/LyapunovFunction.html>.
2. The logical complexity of a software system can be measured by the number of states that we need to check. A program could have high logical complexity initially. However, if it has been formally verified and can be used as is, then its residual logical complexity is zero.

About the Author

Lui Raymond Sha, Ph.D., is the Donald B. Gillies Chair and professor of computer science at the University of Illinois at Urbana-Champaign. He is active in dependable real-time computing systems research. Sha served on the National Academy of Science's committee on certifiably dependable software, served on the Office of Secretary of Defense's avionics advisory team, and is fellow for the Association for Computing Machinery and the IEEE. He has been a member of the National Science Foundation-sponsored CPS research initiative's planning group since its inception. He has also served on the technical staff of the SEI for 13 years.

University of Illinois
201 North Goodwin AVE
Urbana, IL 61801
Phone: (217) 244-1887
Fax: (217) 244-8363
E-mail: lrs@illinois.edu


CIVILIAN TALENT IS MISSION-CRITICAL.
LET'S GET TO WORK.
NAVAIR
CIVILIAN
Choice is Yours.
Discover more about Naval Air Systems Command today.
Go to www.navair.navy.mil
Equal Opportunity Employer | U.S. Citizenship Required

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.