Using Simplicity to Control Complexity

Lui Sha, *University of Illinois at Urbana-Champaign*

Does diversity in construction improve robustness? The author investigates the relationship between complexity, reliability, and development resources, and presents an approach to building a system that can manage upgrades and repair itself when complex software components fail.

According to a US government IT initiative, "As our economy and society become increasingly dependent on information technology, we must be able to design information systems that are more secure, reliable, and dependable."[1] There are two basic software reliability approaches. One is fault avoidance, using formal specification and verification methods[2] and a rigorous software development process. An example of a high-assurance software development process is the DO 178B standard

adopted by the US Federal Aviation Administration. Fault avoidance methods allow computer-controlled safety-critical systems such as flight control, but they can only handle modestly complex software. The trend toward using a large network of systems based on commercial-off-the-shelf components (COTS) also makes applying fault avoidance methods more difficult.

Another approach is software fault tolerance through diversity (for example, using the N-version programming method[3]). Many believe that diversity in software construction results in improved robustness, but is that true? Would the system be more reliable if we devoted all our effort to developing a single version? In this article, I show that dividing resources for diversity can lead to either improved or reduced reliability, depending on the architecture. The key to improving reliability is not the degree of diversity, per se.

Rather, it is the existence of a simple and reliable core component that ensures the system's critical functions despite the failure of non-core software components. I call this approach *using simplicity to control complexity*. I will show how to use the approach systematically in the automatic-control-applications domain, creating systems that can manage upgrades and fix themselves when complex software components fail.

The power of simplicity

Software projects have finite budgets. How can we allocate resources in a way that improves system reliability? Let's develop a simple model to analyze the relationship between reliability, development effort, and software's logical complexity. Computational complexity is modeled as the number of steps to complete the computation. Likewise, we can view logical complexity as the number of

steps to verify correctness. Logical complexity is a function of the number of cases (states) that the verification or testing process must handle. A program can have different logical and computational complexities. For example, compared to quicksort, bubble sort has lower logical complexity but higher computational complexity.

Another important distinction is the one between logical complexity and residual logical complexity. For a new module, logical complexity and residual logical complexity are the same. A program could have high logical complexity initially, but if users verified the program before and can reuse it as is, the residual complexity is zero. It is important to point out that we cannot reuse a known reliable component in a *different* environment, unless the component's assumptions are satisfied. Residual complexity measures the effort needed to ensure the reliability of a system comprising both new and reused software components. I focus on residual logical complexity (just "complexity" for the remainder of the article) because it is a dominant factor in software reliability. From a development perspective, the higher the complexity, the harder to specify, design, develop, and verify. From a management perspective, the higher the complexity, the harder to understand the users' needs and communicate them to developers, find effective tools, get qualified personnel, and keep the development process smooth without many requirement changes. Based on observations of software development, I make three postulates:

- P1: *Complexity breeds bugs*. All else being equal, the more complex the software project, the harder it is to make it reliable.
- P2: *All bugs are not equal*. Developers spot and correct the obvious errors early during development. The remaining errors are subtler and therefore harder to detect and correct.
- P3: *All budgets are finite*. We can only spend a certain amount of effort (budget) on any project.

P1 implies that for a given mission duration t, the software reliability decreases as complexity increases. P2 implies that for a given degree of complexity, the reliability function has a monotonically decreasing improvement
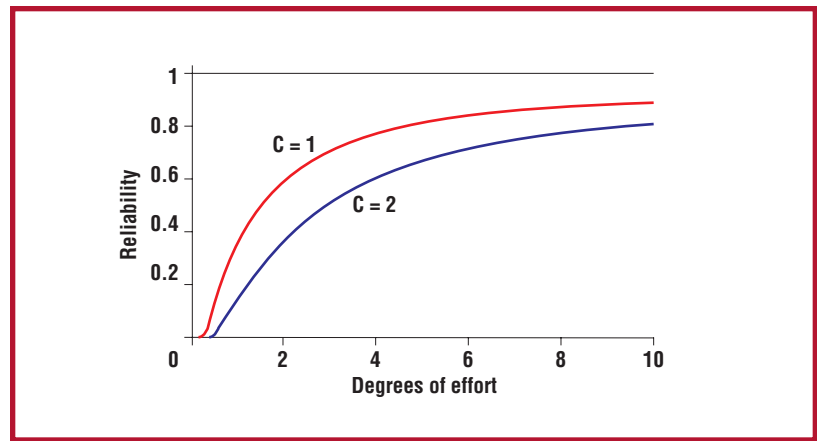

rate with respect to development effort. P3 implies that diversity is not free (diversity necessitates dividing the available effort).

A simple reliability model

The following model satisfies the three postulates. We adopt the commonly used exponential reliability function $R(t) = e^{-\lambda t}$ and assume that the failure rate, λ, is proportional to the software complexity, C, and inversely proportional to the development effort, E. That is, $R(t) = e^{-kCt/E}$. To focus on the interplay between complexity and development effort, we normalize the mission duration t to 1 and let the scaling constant $k = 1$. As a result, we can rewrite the reliability function with a normalized mission duration in the form $R(E, C) = e^{-C/E}$. Figure 1 plots the reliability function $R(E, C) = e^{-C/E}$ with $C = 1$ and $C = 2$, respectively. As Figure 1 shows, the higher the complexity, the more effort needed to achieve a given degree of reliability. $R(E, C)$ also has a monotonically decreasing rate of reliability improvement, demonstrating that the remaining errors are subtler and, therefore, detecting and correcting them requires more effort. Finally, the available budget E should be the same for whatever fault-tolerant method you use.

We now have a simple model that lets us analyze the relationship between development effort, complexity, diversity, and reliability.

The two well-known software fault tolerance methods that use diversity are *N*-version programming and recovery block.[3–5] I'll use them as examples to illustrate the model's application. For fairness, I'll compare each method under its own ideal condition. That is, I assume faults are independent under *N*-version programming and acceptance test is perfect under recovery block. However, neither assumption is easy to realize in practice (leading to the forward-recovery approach,[6] which I'll discuss later).

Figure 1. Reliability and complexity. C is the software complexity.
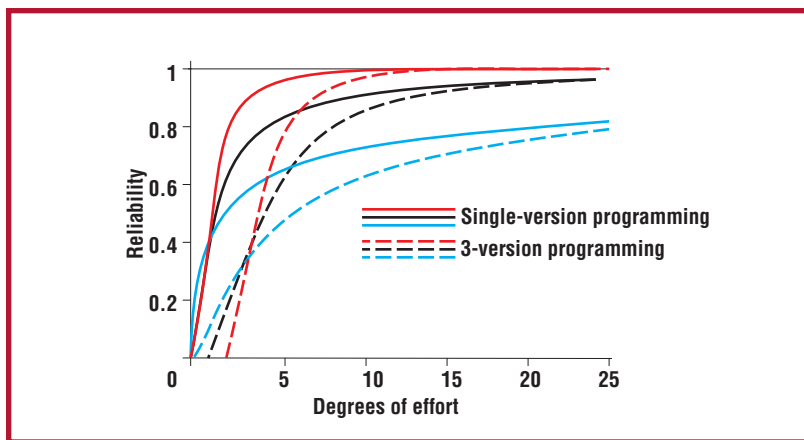
Figure 2. Effect of divided effort in three-version programming when the failure rate is inversely proportional to effort (black), the square of effort (red), and the square root of effort (blue).
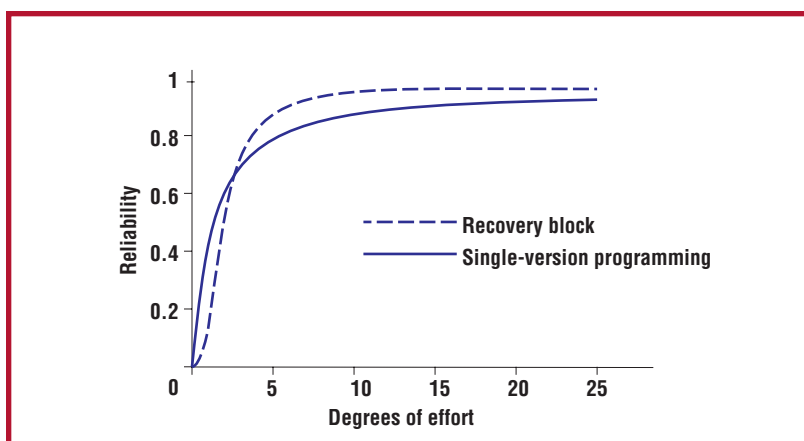
Figure 3. Effect of divided efforts in recovery block.

N-version programming

To focus on the effect of dividing the development effort for diversity, I assume that the nominal complexity is $C = 1$. First, consider the case of N-version programming with $N = 3$. The key idea in this method is to have three teams to independently design and implement different program versions from the same specification, hoping that any faults resulting from software errors are independent. During runtime, the results from different versions are voted on and the majority of the three outputs is selected (the median is used if outputs are floating-point numbers).

In the case $N = 3$, the reliability function of the three-version programming system is $R_3 = R_{E/3}^3 + 3R_{E/3}^2(1 - R_{E/3})$. Replacing E with $(E/3)$ in $R(E, 1) = e^{-1/E}$ provides the reliability function of each version $R_{E/3} = e^{-3/E}$, because the total effort E is divided by three teams. Each team is responsible for a version. The black lines in Figure 2 show that the reliability of single-version programming

with undivided effort is superior to three-version programming over a wide range of development effort.

This result counters the belief that diversity results in improved reliability. To check the result's sensitivity, I make two assumptions. First, I make the optimistic assumption that the failure rate is inversely proportional to the square of software engineering effort; that is, $R(E, 1) = e^{-1/E^2}$ (plotted in red in Figure 2). Second, I make the pessimistic assumption that the failure rate is inversely proportional to the square root of software engineering effort, $R(E, 1) = e^{-1/E^{1/2}}$ (plotted in blue in Figure 2). The plots show that a single version's reliability is also superior to three-version programming under the two assumptions over a wide range of efforts.

However, single-version programming might not always be superior to its N-version counterpart. Sometimes, we can obtain additional versions inexpensively. For example, if you use a Posix-compliant operating system, you can easily add a new low-cost version from different vendors. This is a reasonable heuristic to improve reliability in non–safety-critical systems. The difficulty with this approach is that there is no method that assures that faults in different versions are independent. Nor is there a reliable method to quantify the impact of potentially correlated faults. This is why FAA DO 178B discourages the use of N-version programming as a primary tool to achieve software reliability.

Recovery block

Now, consider diversity's effect in the context of recovery block, where we construct different alternatives and then subject them to a common acceptance test. When input data arrive, the system checkpoints its state and then executes the primary alternative. If the result passes the acceptance test, the system will use it. Otherwise, the system rolls back to the checkpointed state and tries the other alternatives until either an alternative passes the test or the system raises the exception that it cannot produce a correct output.

Under the assumption of a perfect acceptance test, the system works as long as any of the alternatives works. When three alternatives exist, the recovery block system's reliability is $R_B = 1 - (1 - R_{E/3})^3$, where $R_{E/3} = e^{-1/(E/3)}$. Figure 3 shows the reliability of sin-

gle-version programming and of the recovery block with a three-way-divided effort. When the available effort is low, single-version programming is better. However, recovery block quickly becomes better after $E > 2.6$.

Recovery block scores better than N-version programming because only one version must be correct under recovery block—easier to achieve than N-version programming's requirement that the majority of the versions be correct. Diversity in the form of recovery block helps, but to what degree? Figure 4 compares system reliability under recovery block when the total effort E is divided evenly into two, three, and 10 alternatives. (An alternative is a different procedure—called when the primary fails the acceptance test.) All the alternatives have the same nominal complexity $C = 1$. Clearly, dividing the available effort in many ways is counterproductive.

Next, consider the effect of using a reduced-complexity alternative. Figure 5 shows the reduced-complexity-alternative effect in a two-alternative recovery block. In this plot, I divide the total effect E equally into two alternatives. RB2 has two alternatives with no complexity reduction; that is, $C_1 = 1$ and $C_2 = 1$. RB2L2 has two alternatives with $C_1 = 1$ and $C_2 = 0.5$. RB2L10 has two alternatives with $C_1 = 1$ and $C_2 = 0.1$. Clearly, system reliability improves significantly when one alternative is much simpler. (The recovery block approach recommends using a simpler alternative.)

To underscore the power of simplicity, let's consider the effect of a good but imperfect acceptance test. Suppose that if the acceptance test fails, the system fails. Figure 6 plots two reliability functions: recovery block RB2 with a perfect acceptance test and two alternatives, where each has complexity $C = 1$; and recovery block *RB2L5, with an imperfect acceptance test whose reliability equals that of the low-complexity alternative. The two alternatives' complexities are $C_1 = 1$ and $C_2 = 0.2$. As we can see, with a five-fold complexity reduction in one alternative, *RB2L5 is superior to recovery block with a perfect acceptance test but without the complexity reduction in its alternatives.
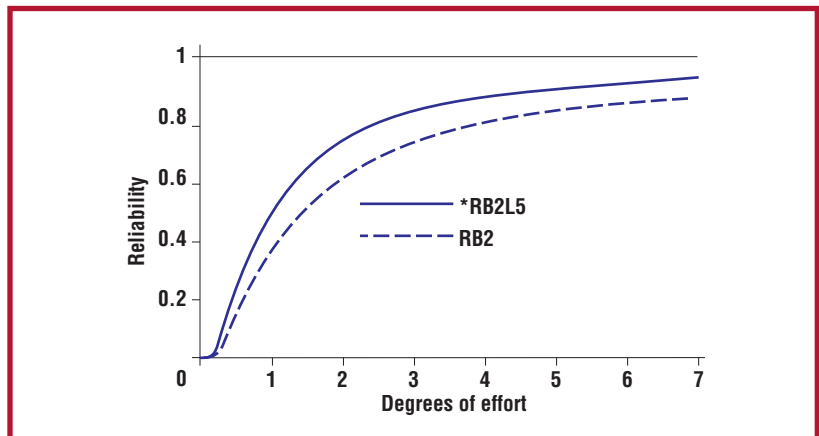
You should not be surprised by the observation that the key to improving reliability is having a simple and reliable core component with which we can ensure a software sys-
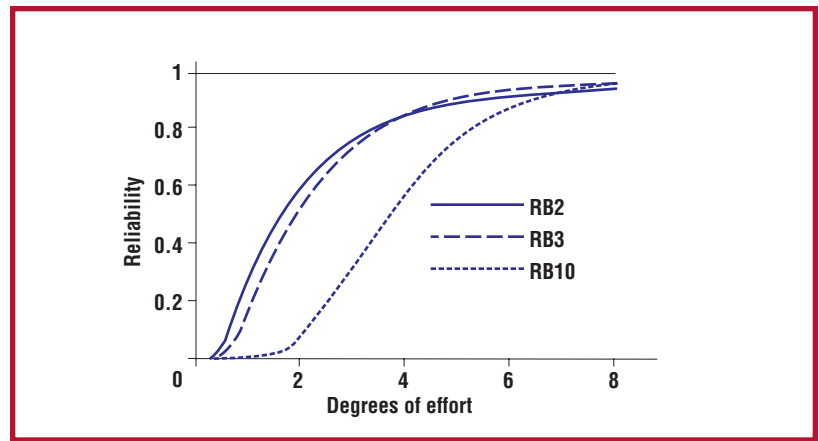
Figure 4. The results of dividing total effort into two (RB2), three (RB3), and 10 (RB10) alternatives in recovery block.
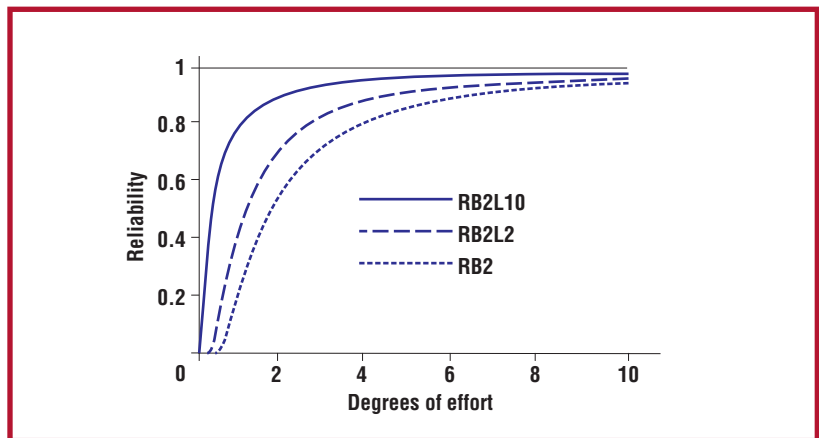
Figure 5. Effect of reducing complexity in a two-alternative recovery block.

Figure 6. Effect of reducing complexity in a two-alternative recovery block with an imperfect acceptance test.

tem's critical functions. After all, "Keep it simple" has long been reliability engineering's mantra. What is surprising is that as the role of software systems increases, we have not taken simplicity seriously in software construction.

> **We can exploit the features and performance of complex software even if we cannot verify them, provided we can guarantee the critical requirements with simple software.**

A two-alternative recovery block with a reduced-complexity alternative is an excellent approach whenever we can construct high-reliability acceptance tests. Unfortunately, constructing effective acceptance tests that can check each output's correctness is often difficult. For example, from a single output, determining whether a uniform random-number generator is generating random numbers uniformly is impossible. The distribution is apparent only after many outputs are available. Many phenomena share this characteristic: diagnosing from a single sample is difficult, but a pattern often emerges when a large sample is available. Unfortunately, many computer applications are interactive in nature; they do not let us buffer a long output sequence and analyze the outputs before using them. Fortunately, we can leverage the power of simplicity using forward recovery.

Using simplicity to control complexity

The wisdom of "Keep it simple" is self-evident. We know that simplicity leads to reliability, so why is keeping systems simple so difficult? One reason involves the pursuit of features and performance. Gaining higher performance and functionality requires that we push the technology envelope and stretch the limits of our understanding. Given the competition on features, functionality, and performance, the production and usage of complex software components (either custom or COTS) are unavoidable in most applications. Useful but unessential features cause most of the complexity. Avoiding complex software components is not practical in most applications. We need an approach that lets us safely exploit the features the applications provide.

A conceptual framework

From a software engineering perspective, using simplicity to control complexity lets us separate critical requirements from desirable properties. It also lets us leverage the power of formal methods and a high-reliability software development process. For example, in sorting, the critical requirement is to sort items correctly, and the desirable property is to sort them fast. Suppose we can verify the bubble sort program but not the quicksort program. One solution is to use the slower bubble sort as the watchdog for quicksort. That is, we first sort the data items using quicksort and then pass the sorted items to bubble sort.

If quicksort works correctly, bubble sort will output the sorted items in a single pass. Hence, the expected computational complexity is still $O(n \log(n))$. If quicksort sorts the items in an incorrect order, bubble sort will correct the sort and thus guarantee the critical requirement of sorting. Under this arrangement, we not only guarantee sorting correctness but also have higher performance than using bubble sort alone, as long as quicksort works most of the time. The moral of the story is that we can exploit the features and performance of complex software even if we cannot verify them, provided that we can guarantee the critical requirements with simple software. This is not an isolated example. Similar arrangements are possible for many optimization programs that have logically simple greedy algorithms (simple local optimization methods such as the nearest-neighbor method in the traveling salesman problem) with lower performance and logically complex algorithms with higher performance.

In the following, I show how to systematically apply the idea of using simplicity to control complexity in the context of automatic control applications. Control applications are ubiquitous. They control home appliances, medical devices, cars, trains, airplanes, factories, and power generators and distribution networks. Many of them have stringent reliability or availability requirements.

The forward recovery solution

Feedback control is itself a form of forward recovery. The feedback loop continuously corrects errors in the device state. We need feedback because we have neither a perfect mathematical model of the device, nor perfect sensors and actuators. A difference (error) often exists between the actual device state and the set-point (desired state). In the feedback control framework, incorrect control-software outputs translate to actuation errors. So, we must contain the impact resulting from incorrect outputs and keep the system within operational constraints. One way to achieve those goals is to keep the device states in an envelope established by a simple and reliable controller. An example of this idea in practice is the

Boeing 777 flight control system, which uses triple-triple redundancy for hardware reliability.[7] At the application-software level, the system uses two controllers. The normal controller is the sophisticated software that engineers developed specifically for the Boeing 777. The secondary controller is based on the Boeing 747's control laws. The normal controller is more complex and can deliver optimized flight control over a wide range of conditions. On the other hand, Boeing 747's control laws—simple, reliable, and well understood—have been used for over 25 years. I'll call the secondary controller a simple component because it has low residual complexity. To exploit the advanced technologies and ensure a high degree of reliability, a Boeing 777, under the normal controller, should fly within its secondary controller's stability envelope. That is a good example of using forward recovery to guard against potential faults in complex software systems.

However, using forward recovery in software systems is an exception rather than the rule. Forward recovery also receives relatively little attention in software fault tolerance literature[8] due to the perceived difficulties. For a long time, systematically designing and implementing a forward recovery approach in feedback control has been a problem without a solution. Using the recent advancement in the linear matrix inequality theory,[9] I found that we can systematically design and implement forward recovery for automatic control systems if the system is piecewise linearizable (which covers most practical control applications).

The Simplex architecture

With my research team at the Software Engineering Institute, I developed the Simplex architecture to implement the idea of using simplicity to control complexity. Under the Simplex architecture, the control system is divided into a high-assurance-control subsystem and a high-performance-control subsystem.

The high-assurance-control subsystem. In the Simplex architecture's HAC subsystem, the simple construction lets us leverage the power of formal methods and a rigorous development process. The prototypical example of HAC is Boeing 777's secondary controller. The HAC subsystem uses the following technologies:

- Application level: well-understood classical controllers designed to maximize the stability envelope. The trade-off is performance for stability and simplicity.
- System software level: high-assurance OS kernels such as the certifiable Ada runtime developed for the Boeing 777. This is a no-frills OS that avoids complex data-structure and dynamic resource-allocation methods. It trades off usability for reliability.
- Hardware level: well-established and simple fault-tolerant hardware configurations, such as pair-pair or triplicate modular redundancy.
- System development and maintenance process: a suitable high-assurance process appropriate for the applications—for example, FAA DO 178B for flight control software.
- Requirement management: the subsystem limits requirements to critical functions and essential services. Like a nation's constitution, the functions and services should be stable and change very slowly.

The high-performance-control subsystem. An HPC subsystem complements the conservative HAC core. In safety-critical applications, the high-performance subsystem can use more complex, advanced control technology. The same rigorous standard must also apply to the HPC software. The prototypical example of HPC is Boeing 777's normal controller.

Many industrial-control applications, such as semiconductor manufacturing, are not safety critical, but the downtime can be costly. With high-assurance control in place to ensure the process remains operational, we can aggressively pursue advanced control technologies and cost reduction in the high-performance subsystem as follows:

- Application level: advanced control technologies, including those difficult to verify—for example, neural nets.
- System software level: COTS real-time OS and middleware designed to simplify application development. To facilitate application-software-component upgrades, we can also add dynamic real-time component-replacement capability in the middleware layer, which supports advanced upgrade management—replacing

> With high–assurance control in place to ensure the process remains operational, we can aggressively pursue advanced control technologies and cost reduction.
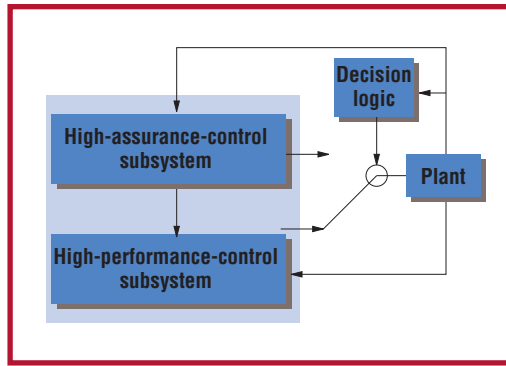
software components during runtime
without shutting down the OS.[10]

- Hardware level: standard industrial
hardware, such as VMEBus-based hardware or industrial personal computers.
- System development and maintenance
process: standard industrial software
development processes.
- Requirement management: the subsystem handles requirements for features
and performance here. With the protection that the high-assurance subsystem
offers, requirements can change relatively fast to embrace new technologies
and support new user needs.

Figure 7 diagrams the Simplex architecture, which supports using simplicity to control complexity. The high-assurance and highperformance systems run in parallel, but the
software stays separate. The HPC can use the
HAC's outputs, but not vice versa. Normally,
the complex software controls the plant. The
decision logic ensures that the plant's state under the high-performance controller stays
within an HAC-established stability envelope.
Otherwise, the HAC takes control.

Certain real-time control applications
such as manufacturing systems are not safety
critical, but they still need a high degree of
availability, because downtime is very expensive. In this type of application, the main concern is application-software upgradability
and availability. For such non–safety-critical
applications, we can run Simplex architecture middleware on top of standard industrial hardware and real-time OSs. A number
of applications have used this technique, including those performed in a semiconductorwafer-making facility.[11]

For educational purposes, I had my group
at the University of Illinois at Urbana-Champaign develop a Web-based control lab—
the Telelab (www-drii.cs.uiuc.edu/download.
html)—which uses a physical inverted pendulum that your software can control to explore

this article's principles. Once you submit your
software through the Web, Telelab dynamically replaces the existing control software
with your software and uses it to control the
inverted pendulum without stopping the normal control. Through streaming video, you
can watch how well your software improves
the control. You can also test this approach's
reliability by embedding arbitrary applicationlevel bugs in your software. In this case, Telelab will detect the deterioration of control performance, switch off your software, take back
control, and keep the pendulum from falling
down. Also, it will restore the control software
in use prior to yours. Telelab demonstrates the
feasibility of building systems that manage upgrades and self-repair.

Forward recovery using high-assurance controller and recovery region

In plant (or vehicle) operation, a set of
state constraints, called *operation constraints*,
represent the devices' physical limitations and
the safety, environmental, and other operational requirements. We can represent the operation constraints as a normalized *polytope*
(an *n*-dimensional figure whose faces are hyperplanes) in the system's *n*-dimensional state
space. Figure 8 shows a two-dimensional example. Each line on the boundary represents
a constraint. For example, the engine rotation
must be no greater than k rpm. The states inside the polytope are called *admissible states*,
because they obey the operational constraints. To limit the loss that a faulty controller can cause, we must ensure that the system states are always admissible. That means

1. we must be able to remove control from
a faulty control subsystem and give it to
the HAC subsystem before the system
state becomes inadmissible,
2. the HAC subsystem can control the system after the switch, and
3. the system state's future trajectory after
the switch will stay within the set of admissible states and converge to the setpoint.

We cannot use the polytope's boundary as
the switching rule, just as we cannot stop a
car without collision when it's about to
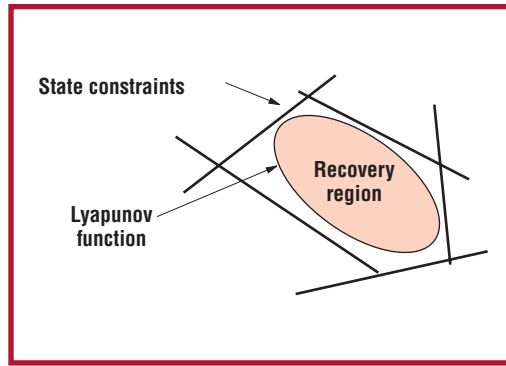touch a wall. Physical systems have inertia.

A subset of the admissible states that satisfies the three conditions is called a *recov-*

ery region. A Lyapunov function inside the state constraint polytope represents the recovery region (that is, the recovery region is a stability region inside the state constraint polytope). Geometrically, a Lyapunov function defines an *n*-dimensional ellipsoid in the *n*-dimensional system state space, as Figure 8 illustrates. An important property of a Lyapunov function is that, if the system state is in the ellipsoid associated with a controller, it will stay there and converge to the equilibrium position (setpoint) under this controller. So, we can use the boundary of the ellipsoid associated with the high-assurance controller as the switching rule.

A Lyapunov function is not unique for a given system–controller combination. To not unduly restrict the state space that high-performance controllers can use, we must find the largest ellipsoid in the polytope that represents the operational constraints. Mathematically, we can use the linear matrix inequality method to find the largest ellipsoid in a polytope.[9] Thus, we can use Lyapunov theory and LMI tools to solve the recovery region problem (to find the largest ellipsoid, we downloaded the package that Steven Boyd's group at Stanford developed). For example, given a dynamic system $\dot{\mathbf{X}} = \overline{\mathbf{A}}\mathbf{X} + \mathbf{BKX}$, where $\mathbf{X}$ is the system state, $\overline{\mathbf{A}}$ is the system matrix, and $\mathbf{K}$ represents a controller. We can first choose $\mathbf{K}$ by using well-understood robust controller designs; that is, the system stability should be insensitive to model uncertainty.

The system under this reliable controller is $\dot{\mathbf{X}} = \mathbf{AX}$, where $\mathbf{A} = (\overline{\mathbf{A}} + \mathbf{BK})$. Additionally, $\mathbf{A}^T\mathbf{Q} + \mathbf{QA} < 0$ represents the stability condition, where $\mathbf{Q}$ is the Lyapunov function. A normalized polytope represents the operational constraints. We can find the largest ellipsoid in the polytope by minimizing $(\log \det \mathbf{Q}^{-1})$ (det stands for determinant),[9] subject to the stability condition. The resulting $\mathbf{Q}$ defines the largest normalized ellipsoid $\mathbf{X}^T\mathbf{QX} = 1$, the recovery region, in the polytope (see Figure 8).

In practice, we use a smaller ellipsoid—for example, $\mathbf{X}^T\mathbf{QX} = 0.7$, inside $\mathbf{X}^T\mathbf{QX} = 1$. The shortest distance between $\mathbf{X}^T\mathbf{QX} = 1$ and $\mathbf{X}^T\mathbf{QX} = 0.7$ is the margin reserved to guard against model errors, actuator errors, and measurement errors. During runtime, the HPC subsystem normally controls the plant. The decision logic checks the plant state $\mathbf{X}$ every sampling period. If $\mathbf{X}$ is inside the *n*-dimensional ellipsoid $\mathbf{X}^T\mathbf{QX} = c$, $0 < c < 1$, it considers admissible the system the high-performance controller controls. Otherwise, the HAC subsystem takes over, which ensures that plant operation never violates the operational constraints. The software that implements the decision rule "if $(\mathbf{X}^T\mathbf{QX} > c)$, switch to high-assurance controller" is simple and easy to verify.

Once we ensure that the system states will remain admissible, we can safely conduct statistical performance evaluations of the HPC subsystem in the plant. If the new "high-performance" controller delivers poor performance, we can replace it online. I would point out that the high-assurance subsystem also protects the plant against latent faults in the high-performance control software that tests and evaluations fail to catch.

Application notes

The development of the high-assurance controller and its recovery region satisfies forward recovery's basic requirement: the impact caused by incorrect actions must be tolerable and recoverable. In certain applications such as chemical-process control, we typically do not have a precise plant model. In such applications, we might have to codify the recovery region experimentally.

When a controller generates faulty output, the plant states will move away from the setpoint. It is important to choose a sufficiently fast sampling rate so that we can detect errors earlier. Will the simple controller unreasonably restrict the state space that the high-performance controller can use? This turns out to be a nonproblem in most applications. The controllers' design involves a trade-off between agility (control performance) and stability. Because the high-performance controller often focuses on agility, its stability envelope is naturally smaller than the stability envelope of the safety controller that sacrifices performance for stability.
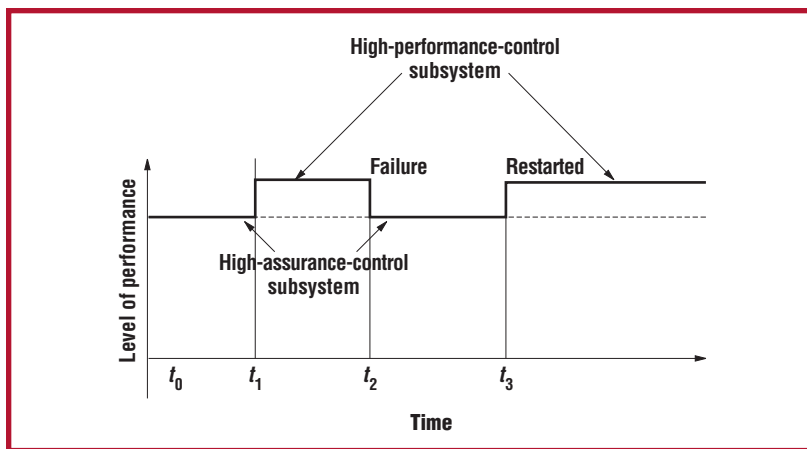
Figure 9. Using imperfect high-performance control. The high-assurance-control subsystem takes over after a failure at time t_2. When the system is stable again, the restarted high-performance-control subsystem resumes control, at time t_3.

About the Author

Lui Sha is a professor of computer science at the University of Illinois at Urbana-Champaign. His research interests include QoS-driven resource management and dynamic and reliable software architectures. He obtained his PhD from Carnegie Mellon University. He is a fellow of the IEEE, awarded "for technical leadership and research contributions that transformed real-time computing practice from an ad hoc process to an engineering process based on analytic methods." He is an associate editor of *IEEE Transactions on Parallel and Distributed Systems* and the *Real-Time Systems Journal*. Contact him at 1304 W. Springfield DCL, Urbana, IL 61801; lrs@cs.uiuc.edu; www.cs.uiuc.edu/contacts/faculty/sha.html.

With the HAC subsystem in place, we can exploit the less-than-perfect HPC subsystem using COTS components. Reasonably good-quality software fails only occasionally when encountering unusual conditions. When the software restarts under a different condition, it will work again. When the HPC subsystem fails under unusual conditions, the HAC subsystem steps in until the condition becomes normal again, at which time we can resume using the HPC subsystem. Figure 9 illustrates the control selection from the two control subsystems. The vertical axis represents control performance levels and the horizontal axis represents time. At time t_0, the system starts using the HAC subsystem. At time t_1, the operator switches the system to the new HPC subsystem. Unfortunately, something triggers an error in the HPC subsystem, so the system automatically switches to the HAC subsystem at time t_2. As the HAC subsystem stabilizes the system, the system control goes back to the restarted HPC subsystem at time t_3. Thanks to the HAC subsystem, we can test new HPC software safely and reliably online in applications such as process-control upgrades in factories.

Forward recovery using feedback is not limited to automatic-control applications. Ethernet, for example, rests on the idea that correcting occasional packet collisions as they occur is easier than completely preventing them. The same goes for the transmission-control protocol: correcting occasional congestion is easier than completely avoiding it. Forward recovery is also the primary tool for achieving robustness in human organizations. Democracy's endurance does not rely on infallible leaders; rather, the system provides a mechanism for removing undesirable ones.

Given the success of forward recovery with feedback in so many engineering disciplines and human organizations, I believe we can apply it to other types of software application. The notion of using simplicity to control complexity ensures the critical properties. It provides us with a "safety net" that lets us safely exploit the features that complex software components offer. That, in turn, lets us build systems that can manage upgrades and fix themselves when complex software components fail. ✸

References

1. "Information Technology for the 21st Century: A Bold Investment in America's Future," www.ccic.gov/it2/initiative.pdf (current 14 April 2001).
2. E.M. Clarke and J.M. Wing, "Formal Methods, State of the Art, and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, Dec. 1996, pp. 626–643.
3. A. Avizienis, "The Methodology of N-Version Programming," *Software Fault Tolerance*, M.R. Lyu, ed., John Wiley & Sons, New York, 1995.
4. B. Randel and J. Xu, "The Evolution of the Recovery Block Concept," *Software Fault Tolerance*, M.R. Lyu, ed., John Wiley & Sons, New York, 1995.
5. S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis of Faults in an N-Version Programming Software Experiment," *IEEE Trans. Software Eng.*, Feb. 1990.
6. N.G. Leveson, "Software Fault Tolerance: The Case for Forward Recovery," *Proc. AIAA Conf. Computers in Aerospace*, AIAA, Hartford, Conn., 1983.
7. Y.C. Yeh, "Dependability of the 777 Primary Flight Control System," *Proc. Dependable Computing for Critical Applications*, IEEE CS Press, Los Alamitos, Calif., 1995.
8. *Software Fault Tolerance (Trends in Software, No. 3)*, M.R. Lyu, ed., John Wiley & Sons, New York, 1995.
9. S. Boyd et al., *Linear Matrix Inequality in Systems and Control Theory*, SIAM Studies in Applied Mathematics, Philadelphia, 1994.
10. L. Sha, "Dependable System Upgrade," *Proc. IEEE Real-Time Systems Symp.* (RTSS 98), IEEE CS Press, Los Alamitos, Calif., 1998, pp. 440–449.
11. D. Seto et al., "Dynamic Control System Upgrade Using Simplex Architecture," *IEEE Control Systems*, vol. 18, no. 4, Aug. 1998, pp. 72–80.