Editor in Chief: **Diomidis Spinellis**
Athens University of Economics
and Business, dds@computer.org

Software Reliability Redux

Diomidis Spinellis

SOFTWARE-INTENSIVE SYSTEMS with high reliability requirements typically are implemented through heroic (and expensive) engineering efforts. Control systems in planes, nuclear reactors, trains, pacemakers, and spaceships are developed by highly trained personnel through strictly managed software development processes with a dose of formal methods. This approach has worked admirably up to now, but its strains are beginning to show.

We're Not in Kansas Anymore

Start with ubiquity and cost. With "software eating the world," the requirement for high reliability is no longer restricted to a few specialized and proven domains. Instead, ever more functions whose failure can hurt humans and damage property are cropping up in new areas. Critical software appears in applications ranging from hobbyist drones and Wi-Fi routers to lithium-ion battery charging circuits and personal health monitors, to automated trading and door locks. Frighteningly, the software development budget for some application areas might be too low to cover fancy reliability engineering. So, the organizations that develop the software might lack the people, processes, and tools to deliver the required reliability.

Then there's the risk from end-user programming. Software applications increasingly offer users the ability to configure and program them. This can be helpful when we use a spreadsheet to automate submission of our travel expenses or use a content management system to simplify editing our school's website. However, letting untrained users program in critical application areas could be like letting a drunk pilot fly a jumbo jet.

This state of affairs often develops gradually, in ways that are difficult to manage. An enthusiastic amateur programmer realizes he or she can use a small Visual Basic or Python script to easily automate a peripheral but tedious process. Over the years, the process becomes more important to the amateur programmer's organization, and the script grows multiple tentacles as it gets connected to other services. Then, a user mistakenly enters a negative price or the script runs on 29 February, and multiple

0740-7459/17/$33.00 © 2017 IEEE

services fail catastrophically because the script was never properly tested.

Critical software with high reliability requirements is also growing bigger and more complex. This happens because, spurred by advancements in other application areas and increased hardware capabilities, we demand more from it. For example, we expect a car's console to be at least as friendly as our smartphone, not realizing that a software crash on our phone is an inconvenience, whereas a car crash can be a tragedy.

In addition, managing the development of critical software becomes more difficult because the way we build software is changing, with third-party components providing much of an application's required functionality. The Apollo program's spacecraft software ran on bare metal, and each part of it could be carefully verified. In contrast, a modern critical-application software stack might include an OS kernel with many millions of lines; third-party device drivers and firmware in binary form; large middleware components; and open source libraries handling data compression, HTTP communication, or cryptography developed by thousands of volunteers.

As if handling the size and complexity wasn't challenging enough, many software applications requiring high reliability comprise a multitude of interconnected systems. Parts of an application might run in an embedded device; other parts might run on a cloud provider's servers; and yet other elements might depend on queuing, geolocation, image recognition, messaging, or database functionality provided by third parties as a service. These complex systems' failure modes are difficult to predict and handle. Famously, when some of Amazon's cloud services failed a few months ago, the status indication dashboard didn't work as expected because the necessary red or green images were stored on Amazon's failed Simple Storage Service.

To top it all, critical software often must be actively maintained for decades. As Mike Milinkovich, the Eclipse Foundation's executive director, said, "The software you're writing today may have to be maintained by your great-granddaughter."[1] This has always been the case because the time span from design to the end of the corresponding hardware's life can indeed be more than half a century. What has changed is the type of required maintenance. Systems connected over the Internet require regular updates to face new threats and to handle protocol evolution. It was admirable that Microsoft had in place a build environment and an infrastructure to release a Windows XP patch for the EternalBlue vulnerability later exploited by the WannaCry ransomware. However, the organizations whose operations relied on the long-unsupported system were treading on thin ice. Also, the hardware of modern large complex systems depends on so many manufacturers that maintaining it in its original state for decades is hard. The necessary upgrades bring with them new device drivers and fresh whole OS releases—a verification nightmare for critical systems.

Somewhere, over the Rainbow, Skies Are Blue

Avoiding problems and catastrophes in the new software reliability landscape won't be easy. Consider the ubiquity of software performing critical functions and of devices whose software isn't appropriately maintained. Unfortunately, for software that's developed with opaque,

EDITORIAL STAFF

Lead Editor: Meghan O'Dell, m.odell@computer.org
Content Editor: Dennis Taylor
Staff Editors: Lee Garber and Rebecca Torres
Publications Coordinator: software@computer.org
Lead Designer: Jennie Zhu-Mai
Production Editor: Monette Velasco
Webmaster: Brandi Ortega
Multimedia Editor: Erica Hardison
Illustrators: Annie Jiu, Robert Stack, and Alex Torres
Cover Artist: Peter Bollinger
Director, Products & Services: Evan Butterfield
Senior Manager, Editorial Services: Robin Baldwin
Manager, Editorial Content: Carrie Clark
Senior Advertising Coordinator: Debbie Sims, dsims@computer.org

potentially slapdash, processes, part of the answer will likely have to be regulation. Currently, the cost of misbehaving software is passed to users (in the form of failures) and the environment (as devices discarded owing to faulty unmaintained software). Left on its own, the market is unlikely to solve this problem. This is because users have insufficient information regarding the software's reliability and because most software isn't marketed in time frames that allow the establishment of trustworthy brands. So, regulation that increases transparency regarding the software's reliability and makes manufacturers of critical software liable for failures and responsible for maintenance over clearly specified periods will result in better outcomes for all parties involved.

The issues associated with end-user programming will require multiple parties to do their part. Organizations must set up efficient methods to inventory and characterize their software assets and the assets' dependencies and importance. In parallel, developers of applications and frameworks that are often used for end-user programming must continue promoting the development of more reliable systems. Some avenues include increased reliance on static checking; runtime provisions for handling and recovering from failures; and built-in support and gentle encouragement for good software development processes such as modularization, unit testing, and configuration management. Given the ever-larger number of people involved in putting together algorithmic rules and systems, increased software engineering literacy among the general population will also help.

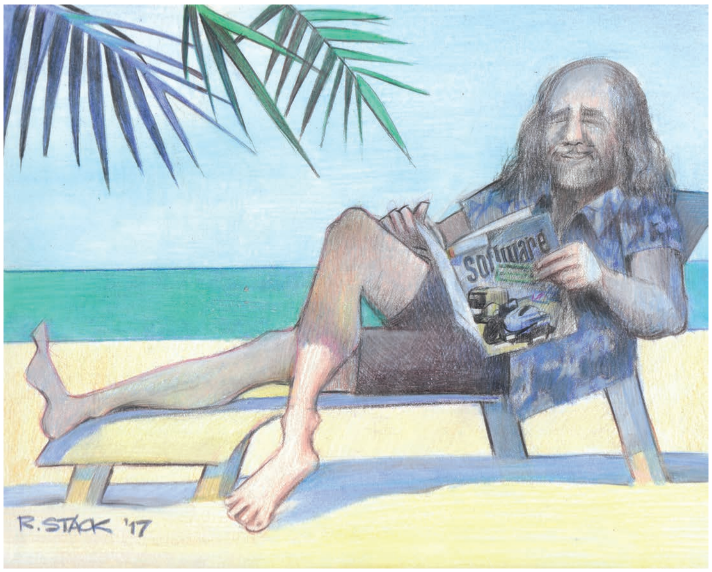
There are no easy answers to the reliability challenges arising from modern software's size and complexity. Making suppliers responsible for software maintenance and failures should result in the availability of more trustworthy components. As a bonus, in such an environment, we'll be more likely to see a business case for maintaining critical open source libraries and systems. Thankfully, systems software, which faces less pressure to evolve to changing requirements than applications do, becomes more reliable as it matures. So, designers should prefer us-

THANK YOU, GRADY!

Grady Booch published his first *IEEE Software* article in 1994[1] and graced our magazine with his On Architecture and On Computing columns from 2007 until last year. I've learned a lot from his thoughtful, original, and reflective writing, a feeling I'm sure all *IEEE Software* readers share. So, please join me in thanking Grady for his long, gallant service to our magazine and community.

Reference

1. G. Booch, "Coming of Age in an Object-Oriented World," *IEEE Software*, vol. 11, no. 6, 1994, pp. 33–41.



ing software components that have proved their mettle over the temptation to adopt whatever technology is in fashion each year.

Addressing reliability concerns is even more difficult with complex systems. Few organizations and groups have experience developing and running complex, large, reliable systems. Even those organizations with that experience have occasionally contended with spectacular failures.

Thus, the first lesson is to isolate the most critical functionality in stand-alone units rather than implement it as part of a complex system. We can also try to learn from experienced organizations. Commendably, some are publishing their practices[2] and failure postmortems. These lessons need to be generalized into scientific theory and make their way into university curricula. In the longer term, we can copy nature and build complex systems by combining multiple, diverse, interchangeable components with independent failure modes.

Some candidate solutions crosscut all problem areas. Innovations that reduce the cost and time to develop reliable software would help a lot, but we can't bank on them. Improved, probably longer, education with increased emphasis on software reliability can be a requirement for people developing critical software. As professionals, we should also assume more responsibility for the software we develop. Professional societies can do their part here by standardizing and promoting the state of the art. An admirable step in this direction is the IEEE Computer Society's *Guide to the Software Engineering Body of Knowledge* (available at www.computer.org/web/swebok/v3).

Throughout its 50-year history, software engineering has evolved splendidly through numerous crises. Modern software reliability challenges can also be solved by applying the two simple elements used in all past calamities: the courage to face the problem and the brain to solve it.

References

1. M. Milinkovich, "Open Collaboration: The Eclipse Way," keynote address at 2017 Int'l Conf. Software Eng. (ICSE 17), 2017.
2. B. Beyer et al., *Site Reliability Engineering: How Google Runs Production Systems*, O'Reilly Media, 2016.