The Therac-25: 30 Years Later

Nancy G. Leveson, MIT

A widely cited 1993 Computer article described failures in a software-controlled radiation machine that massively overdosed six people in the late 1980s, resulting in serious injury and fatalities. How far have safety-critical systems come since then?

FROM THE EDITOR

As part of our 50th anniversary celebration, this special feature revisits influential *Computer* articles from the past. This month, the author of an article on the investigation of the Therac-25 accidents between 1985 and 1987 reflects on the current state of practice of safety-critical systems. *—Ron Vetter, Editor in Chief Emeritus*

Between June 1985 and January 1987, a software-controlled radiation therapy machine called the Therac-25 massively overdosed six people, resulting in serious injury and deaths. A widely cited paper published in 1993 detailed the causes of these accidents.[1,2] It wasn't an anomaly but simply the first of many, as these types of radiation-overdose accidents continue today. In 2010, a series of articles were published describing related accidents in radiation therapy.[3] Many more such events have occurred since then. Indeed, the problems aren't limited to medical linear accelerators.

Many lessons should have been learned from the Therac-25 events, most of which are generalizable to almost every industry that employs safety-critical devices. In this article, I examine each of the identified factors in these accidents to determine what progress has been made in the past 30 years. In the interest of space, the specifics of the Therac-25 events aren't detailed—instead, I focus on the state of practice 30 years after the events.

OVERCONFIDENCE IN SOFTWARE

There's still widespread belief that software doesn't fail, unlike the hardware devices it replaces. In safety-critical systems today—including medical devices, aircraft, nuclear power plants, and weapons systems—standard hardware backups, interlocks, and other safety devices are often replaced by software. Where hardware backups are still used, they're often controlled by software.

Many hardware engineers (and some software engineers) have so much confidence in software that they

0018-9162/17/$33.00 © 2017 IEEE

EDITOR **RON VETTER**
University of North Carolina Wilmington; vetterr@uncw.edu

eliminate the hardware protection that was common before software took over. Engineers don't purposely create designs where a hardware single-point failure could lead to a catastrophe. The same needs to be true for the software in these systems. Protection against software errors can and should be built into both the system and the software itself.

ARCHIVED ARTICLES

The original article remains very popular as indicated by the number of downloads it receives from the IEEE Computer Society Digital Library. All of the articles mentioned in this special column are free to view at www.computer.org/computer-magazine/from-the-archives-computers-legacy.

CONFUSING RELIABILITY WITH SAFETY

When systems were primarily electromechanical, nearly exhaustive testing was possible and design errors could be eliminated before operational use. What was left during operations were random wearout failures. Safety, then, could be assumed to be effectively approximated by reliability.

Software is by design abstracted from its physical realization. Although the hardware on which the software is executed might fail, the design itself doesn't fail. In fact, software by itself isn't safe or unsafe—safety depends on context. Much of the Therac-25 software had been used on an earlier version of the machine called the Therac-20. The same flaws that killed people with the Therac-25 weren't dangerous because of the design of the Therac-20 hardware that the software was controlling. Software safety always depends on the context in which the software is used. In fact, nearly all accidents involving software have resulted from flawed software requirements, not implementation. There's still a widespread but untrue belief that software will be safe if it satisfies its requirements, or that the safety of a specific piece of software can be evaluated apart from its use environment.

Another widespread misunderstanding is that software safety is enhanced by assuring that the software satisfies the requirements. Almost all software-related accidents have involved requirements flaws, not coding or implementation errors. Perhaps this misguided reliance on assurance stems from the fact that we have many solutions proposed for software assurance, but few that identify safety-critical software requirements. Not much has changed in 30 years in this regard. Nearly all standards for safety-critical software focus on implementation assurance. If we truly want to reduce software-related accidents, we have to focus less on assurance and more on identifying the safety-critical requirements and building safety into these machines from the beginning of development. Safety can't be ensured if it isn't already there; it has to be built in from the beginning.

LACK OF DEFENSIVE DESIGN

The Therac-25 software didn't contain self-checks or other error-detection and error-handling features that would have detected problems. There were no independent checks that the machine was operating correctly. Such verification can't be assigned to operators without providing them with some means of detecting errors. Operators are often blamed for medical device accidents when the problems were actually in the machine design.

Although audit trails were limited 30 years ago due to the technology of the time, they are common today. One problem, however, is that so much data is collected—for instance, in aircraft operations—that it's difficult to detect problems until after a serious event has occurred. The problem today isn't in collecting data, but rather in identifying trends and behaviors of the hardware, software, and operators that are increasing risk before an accident occurs. Collecting data alone won't help here. We need sophisticated modeling and analysis tools to analyze the data. Data—"big" or not—isn't the same as information.

UNREALISTIC RISK ASSESSMENTS

The Therac-25 had a probabilistic risk assessment—including an update after one of the early accidents—that led to dangerous complacency. Many (perhaps most) industries today make the same types of assumptions based on probabilistic risk assessments. In general, such calculations often exclude aspects of the problem that are difficult to quantify (such as software requirements inadequacies) but which might have a larger impact on safety than the quantitative factors that are included. There are few if any scientific evaluations of the correctness of such assessments. I wrote this in the original Therac-25 article and it's still true: "In our enthusiasm to provide measurements, we should not attempt to measure the unmeasurable."

Specific software flaws leading to a serious loss can't be assessed probabilistically. Even if they could, it

would require so much information about the flaw that it could be fixed instead of justified away as "will never or rarely occur." I've participated in accident investigations or read accident reports involving software for 30 years. In every case, there had been a probabilistic risk assessment that was used to convince decision makers that the accident couldn't occur, usually in the exact way that the accident did occur. Even after a loss or significant event, engineers and management often ignored the fact that the risk assessment was obviously wrong, and they continued to believe in it until the next event—and sometimes after several events—until a major loss occurred. For some reason, people seem to believe in a calculated number more than actual experience.

Engineers often argue that decisions can't be made without probabilistic risk assessment, but this is simply untrue. The safety program with the best historical track record—the US Navy's submarine safety program (SUBSAFE)—allows decision making about safety using only Objective Quality Evidence, which is defined as "any statement of fact, either quantitative or qualitative, pertaining to the quality of a product or service, based on observations, measurements, or tests that can be verified."[4] Probabilities about what will happen in the future can't be verified and so aren't allowed. No SUBSAFE submarine has been lost in the 54 years since SUBSAFE was created after the *Thresher* loss in 1963.[4] Before that time, one submarine was lost on average every three years.

If accurate probabilities aren't determinable (as is true for systems that contain software), then making them up or simply ignoring the software in the system risk assessment isn't the answer. A better solution is to design for the worst case (instead of assuming that only the average case will occur) and creating better decision-making tools that don't require unsupportable risk assessments.

INADEQUATE INVESTIGATION OF INCIDENTS OR FOLLOW-UP ON ACCIDENT REPORTS

Superficial accident/incident analyses, usually placing all or most of the blame on the operators, leads to patching symptoms but not to understanding the deeper underlying and systemic causes of a loss or near loss. This process leads to further accidents that could and should have been prevented.[4] We need to look at the role of the entire system in the accident to make progress in safety.

Engineers often argue that decisions can't be made without probabilistic risk assessment, but this is simply untrue.

INADEQUATE SOFTWARE AND SYSTEM ENGINEERING PRACTICES

The Therac-25 software was created in 1974 and used what are now considered obsolete software engineering practices. There are, however, factors in the accident related to software engineering that are still common today. Too often, practices that might be acceptable for website or productivity tools development are used and even promoted for safety-critical software. A few of these include:

- Software specifications and documentation are often an afterthought, and the development of requirements after the design is created is sometimes even touted as a benefit or necessity. This simply isn't true for safety-critical software where the loss might involve not only human life but physical property, critical mission loss, and damage to the environment. It should be considered malpractice for software to be created before the safety requirements are identified. Identifying these requirements will necessitate sophisticated system-engineering hazard analysis techniques.[4]
- Software designs are often unnecessarily complex. Whereas object-oriented design is appropriate for data-oriented systems, it's not appropriate for control-oriented systems. The resulting software is more difficult to test for safety, trace from requirements to code, maintain without affecting safety, and assure the correctness of changes to safety-critical requirements. We need to get over the fixation that there is one best way to design software for all types of systems.
- Defensive design is sometimes not emphasized. Ways to detect or prevent software errors should be designed in from the beginning. If written first, error-handling routines will get the most exercise. Many (and perhaps most) errors in operational software lie in the error-handling routines themselves. Programming languages that provide error protection, such as strong type checking, aren't always popular or used in safety-critical systems, but are necessary for safety-critical systems.
- Software engineers and human factors engineers don't work together enough. We create software today that's confusing to operators and can lead to accidents, but the operator is blamed rather than the software. Software design

can create operator mode confusion, situation awareness errors, and so on. These dangerous features of software need to be avoided or detected and handled in some way.

SOFTWARE REUSE

The Therac-25 was an improvement of an earlier machine made by the same company called the Therac-20, and much of the software was reused. Today, overconfidence in reuse is still rampant. A false assumption might be made that reusing software or using commercial off-the-shelf software increases safety because the software has been exercised extensively. As stated earlier, software is only safe or unsafe within a specific context. It isn't possible to determine safety by looking at the software alone. Reusing software that was safe in one system doesn't mean it will be safe when used in a different system. Safety is a quality of the system in which the software is used; it's not a quality of the software itself. The belief sometimes built into practice or even government standards that reused software is safe or safer isn't justified.

SAFE VERSUS "FRIENDLY" USER INTERFACES

Although the interface equipment and principles used for the Therac-25 are obsolete, there are still potential issues even with today's more sophisticated interface tools. Sometimes making the interface easy to use conflicts with safety. For example, eliminating multiple data entry and assuming that operators would check the values carefully before pressing the return key was unrealistic for the Therac-25 and for most systems. I have been involved in reviews of several newer safety-critical system interfaces and have been surprised by how many included unsafe features. One example is allowing operators to turn off alarms with no indication showing that the alarms had been turned off. Although there can be a good reason for inhibiting alarms, a new operator needs to be aware that this has occurred. One general design principle is that actions to get into or maintain a safe state should be easy to do. Actions that can lead to an unsafe state (hazard) should be hard to do.[2]

Relying on operators to detect errors and recover before an accident isn't realistic, particularly when the operator isn't provided with the support to perform this function. Some of the radiation therapy accidents since the Therac-25 overdoses have involved operators not being able to see or react to error messages. The accidents were then blamed on the operators rather than on the machine and interface design. The same has occurred in other industries, where operators are blamed for accidents that are primarily the result of flawed engineering.

USER AND GOVERNMENT OVERSIGHT AND STANDARDS

Although the US Federal Drug Administration (FDA)'s original response to the Therac-25 accidents (after they were understood) was impressive, later follow-through was weak, as is current regulation of medical device software. The same is true for other industries. The FDA puts more emphasis on reporting adverse events in linear accelerators, for example, than on preventing them in the first place. One problem is the difficulty and time required to update standards. In general, standards should never include specific techniques (such as failure modes and effects analysis [FMEA] for medical devices) or they will become out of date almost immediately, without any possibility of being updated for perhaps a decade.

Standards can have the undesirable effect of limiting the safety efforts and investment of companies that feel their legal and moral responsibilities are fulfilled if they follow the standards. As the standards often represent the input of commercial companies, there are often conflicts of interest involved in producing effective standards.

My original account of the Therac-25 losses said that accidents are seldom simple. They usually involve a complex web of interacting events with multiple contributing technical, human, organizational, and regulatory factors. We aren't learning enough today from the events nor focusing enough on preventing them. It's time for computer science practitioners to be better educated about engineering for safety.

REFERENCES

1. N.G. Leveson and C.S. Turner, "An Investigation of the Therac-25 Accidents," *Computer*, vol. 26, no. 7, 1993, pp. 18–41.
2. N. Leveson, *Safeware: System Safety and Computers*, Addison Wesley, 1995.
3. W. Bogdanich and K. Rebelodec, "A Pinpoint Beam Strays Invisibly, Harming Instead of Healing," *The New York Times*, 28 Dec. 2010; www.nytimes.com/2010/12/29/health/29radiation.html?mcubz=0.
4. N.G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, MIT Press, 2012.

NANCY G. LEVESON is a professor in the Aeronautics and Astronautics Department at MIT. She is a member of the US National Academy of Engineering, and has received many awards for her research in software engineering, systems safety, and systems engineering. Contact her at leveson@mit.edu.