bashar nuseibeh
imperial college, london

Ariane 5: Who Dunnit?

A forum for exchanging ideas, philosophy, and experience.

Editor:
Tomoo Matsubara
Matsubara Consulting
1-9-6 Fujimigaoka,
Ninomiya, Naka-gun,
Kanagawa 259-01
Japan
tmatsu@xa2.so-net.or.jp

At a 1993 software symposium in inland China, a keynote speaker from the US joked that he had arrived safely because the transportation systems in China "were not yet heavily controlled by software." When disastrous accidents occur, computers are often singled out for blame—sometimes, rightly so. What have we learned from our many expensive lessons? Not much. One lesson is the need for risk management, but as Nuseibeh points out, it is not practiced even in mission critical projects like Ariane 5.

—Tomoo Matsubara

ON 4 JUNE 1996, THE MAIDEN FLIGHT OF the Ariane 5 launcher exploded about 37 seconds after liftoff. Scientists with experiments on board that had taken years to prepare were devastated. For many software engineering researchers, however, the disaster is a case study rich in lessons. To begin learning from this disaster, we need look no further than a report on it issued by an independent inquiry board set up by the French and European Space Agencies.

VARIED VIEWS. Here are some of the interpretations of the report that I have heard.

♦ *What the programmers said:* The disaster is clearly the result of a programming error. An incorrectly handled software exception resulted from a data conversion of a 64-bit floating point to a 16-bit signed integer value. The value of the floating point number that was converted was larger than what could be represented by a 16-bit integer, resulting in an operand error not anticipated by the Ada code. Better programming practice would have prevented this failure from occurring.

♦ *What the designers said:* The disaster is clearly the result of a design error. The system design specification accounted for random hardware failures only, and therefore the exception handling mechanism was unable to recover from a random *software* error. As a result, a correctly functioning processor in the Inertial Reference System (SRI) was shut down, and soon afterwards the backup processor "failed" in the same way. A better design, such as one that disallowed software exceptions from halting hardware units that were functioning correctly, would have prevented the failure.

The disaster is a case study rich in lessons; we need look no further than the inquiry board's report.

♦ *What the requirements engineers said*: The disaster is clearly the result of incorrect analysis of changing requirements. The requirements for Ariane 5 were different from earlier models of Ariane. However, the rogue piece of alignment code that resulted in the failure of Ariane 5 was not actually needed after liftoff, as it had been on earlier models. It remained operational in Ariane 5 without satisfying any (traceable) requirement. Better requirements analysis and traceability would have prevented this failure from occurring. Software maintenance researchers supported this view. The failure, they claimed, could have been prevented if the adaptive maintenance team had not adopted the approach of "if it ain't broke, don't fix it."

♦ *What the test engineers said*: The disaster is clearly the result of inadequate validation and verification, testing, and review. For example, as the report states, there was no test to verify that the SRI would behave correctly "when being subjected to the countdown and flight time sequence and the trajectory of Ariane 5." This, and many other ground tests that could have been performed by suppliers during acceptance testing or review, would have exposed the failure.

♦ *What the project managers said*: The disaster is clearly the result of ineffective development processes and project management. For example, the review process for Ariane 5 development was inadequate. In reviewing specifications, code, and rationale documents, no participants external to the pro-

0740-7459/97/$10.00 © 1997 IEEE

ject were involved, and code and its associated documentation were frequently inconsistent. Improved project management processes that facilitate closer engineering cooperation—with "clear-cut authority and responsibility"—would have increased the chances of exposing the failure.

ANOTHER VIEW. All these interpretations are valid. Certainly, a programming error triggered the failure. Certainly, more rigorous design could have prevented such a programming error from occurring. Certainly, better requirements analysis and specification would have made design verification more achievable. And certainly, improved project management could have provided a more effective organizational process that recognized the impact of changing requirements on system functionality and behavior.

I would also add inadequate *risk management* to the preceding list, since it partly explains the original decision not to handle the fatal exception. That decision was the correct one when it was made (when the primary concern was to keep the SRI processor workload below the chosen threshold of 80 percent). Unfortunately, the decision became the wrong one when the requirements changed. So, a requirements conflict—between a requirement for robustness and a requirement for processor load—was resolved one way, but as time passed the original resolution was invalidated (for more on this see my article with Steve Easterbrook, "Using Viewpoints for Inconsistency Management," in *BCS/IEE Software Engineering Journal*, Jan. 1996).

Jean-Marc Jézéquel and Bertrand Meyer suggest that this is a problem of specification reuse, which could have been mitigated through "design by contract" to specify interfaces between modules precisely ("Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997). I prefer to classify this as a failure of risk management because a risky decision was not reviewed as the project evolved (although, certainly, more precise specifications of affected and reused components could have helped to highlight dependencies and risks). Thus, the key lesson is that risk management should not be performed at the start of a project and then forgotten; risk can change as requirements change, and in software systems that change, risk is more likely to increase than decrease.

DEBUNKING MYTHS. Over the years, there have been many spectacular failures of safety-critical systems, the technical causes of which have been rigorously analyzed (see, for example, Nancy Leveson's *Safeware: System Safety and Computers*, Addison-Wesley, 1995). Although the Ariane 5 disaster is generally attributed to software failure, we court danger when we develop unreasonable expectations of software. For example, believing that "computers provide greater reliability than the devices they replace" or that they "reduce risk over mechanical systems" are but two of many myths about software.

Reuse does not necessarily increase safety, as the reuse of the "wrong" piece demonstrated.

Software fails and engineering trade-offs must be made. For example, an unhandled exception forces developers to protect only some variables. Changing software is not easy because doing so can introduce as many errors as changes. Reuse does not necessarily increase system safety, as the reuse of the "wrong" piece of software demonstrated in Ariane 5. Software testing is by its nature partial, because it only flags errors and cannot prove their absence. Conversely, formally verifying an entire software system such as Ariane is typically unfeasible.

Of course, improved fine-grain processes, such as better programming and design techniques, as well as coarse-grain processes, such as better project management, may help prevent such failures from occurring. However, as Anthony Finkelstein and John Dowell point out, for large-scale software systems development, the complexity of problems and solutions is such that the real reasons for failure are usually *systemic* ("A Comedy of Errors: The London Ambulance Service," *Proc. 8th Int'l Workshop Software Specification and Design*, IEEE Computer Society Press, 1996). That is, it is the combination and interaction of numerous related, overlapping activities and perspectives that result in failure. Clearly, research agendas should recognize the need for sound software engineering principles such as separating concerns (exemplified by object-orientation, multiple views, and component-based development). They must also recognize, however, the importance of relating concerns (exemplified by work on managing interference, interoperability, and coordination). Finally, recognizing that building software systems is an *engineering* process and requires precise specifications and trained engineers is, I believe, an obvious but fundamental first step toward developing safer systems.

Conclusion: Who dunnit? The butler did it. Who discussed it? The researchers. As my colleague Jeff Kramer pointed out, the claims I recount here were made by researchers, rather than practitioners—the latter being much less enthusiastic about accepting blame for the failure! ◆

A full text of the inquiry board's report, "Ariane 5: Flight 501 Failure," is available at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press33.html.

Bashar Nuseibeh is a lecturer and head of the Software Engineering Laboratory in the Department of Computing, Imperial College, London. His research interests are in requirements engineering, software process modeling and technology, and technology transfer. He is an editor-in-chief of the Automated Software Engineering Journal *and chair of the BCS Requirements Engineering Specialist Group. He can be reached at ban@doc.ic.ac.uk and http://www-dse.doc.ic.ac.uk/~ban.*