Title: The Ariane 5 bug and a few lessons

Author: Les Hatton, Oakwood Computing, U.K. and the Computing Laboratory, University of Kent, UK.

TOC: A fascinating example of a problem caused by a strength in a programming language and not a weakness.

ARTICLE:

On the 4th June, 1996, the maiden flight of the Ariane 5 launcher ended after 40 seconds with the launcher spread over a fairly large part of Kourou, greatly depressing the market in tiny pieces of launch vehicle. The enquiry board responded with commendable speed and on the 19th July, a report came out describing its deliberations and conclusions and fascinating reading it makes too.

This incident is one of a wide class of famous failures based on *precision problems*, (one of these was responsible for the Patriot missile failure in the Gulf War in 1991 which led to the death of 29 people; another led to the Bank of New York having to borrow $24 billion for a day from the Federal Reserve a few years ago). The interesting thing about this class of problem is that they are generally statically detectable and are therefore highly avoidable.

In the case of Ariane 5, the programmers had arranged the code such that a 64 bit floating point number was shoe-horned into a 16-bit integer. This is not easy in Ada, the programming language used, in fact you have to override the compiler's objections to achieve it, so they did. There were seven such occasions but only four of them protected against the possibility of overflow. The other three were not protected because the programmers thought they could never overflow. They were wrong. Now it gets interesting.

The offending piece of software was actually *re-used* from Ariane 4, (re-use was also implicated in the tragic software failure in Therac-25 which led to the death of 3 people after severe radiological overdose). In fact, this piece of software had no relevance to the flight of Ariane 5, its use ceasing at the point of lift-off. However, it continued to run. Approximately 37 seconds into the flight, the 16-bit integer overflowed. Now in a sloppier language like C, the program would have continued happily rumbling away to itself but would not in all probability have interfered with the flight. However, the Ada language is made of sterner stuff. Faced with this run-time exception, the program threw an exception as any reasonable language should in such a situation. The programmers did not handle the exception because the assumption was made that the program was correct until proved at fault, apparently a feature of the programming culture for this system, (this observation is worth an article in itself). The default action was regrettably to close the system down, including other components which *were* critical.

At this point, Ariane 5 then demonstrated the fundamental weakness of restricted 2-way diversity. The offending piece of software runs in an SRI (Inertial Reference System) of which there are two, a primary system and a 'hot' back-up. When the first fails, the backup jumps in and takes over. Each SRI duplicates the hardware *and* the software. When the primary SRI closed down, the backup SRI took over and not surprisingly, failed for the same reason 72 msec. later. From this point on,

Ariane 5 assumed the aerodynamic properties of an overhead projector and shortly afterwards turned it self into 12 km. of debris.

What can we learn from this ? There are several lessons:-

a) Type and precision mismatching is once again identified as a primary source of computer systems failure. For compilers which do not warn adequately of such precision faults, they are often statically detectable by reasonable tools. In stronger typed languages, for example, Ada, you actually have to go to significant lengths even to circumvent the compiler's objections to this dangerous practice. For the Ariane 5 programmers to subvert this apparently on the grounds that they can avoid problems by reasoning alone seems highly questionable.

b) Re-use without very detailed re-analysis can be a dangerous de-stabiliser of systems behaviour. Re-use is considered one of the great hopes of modern software engineering, but if you re-use a component inappropriately or you re-use a faulty component, the system certainly isn't going to get better. There are enough examples of this happening to cast considerable doubt on our current understanding of re-use.

c) Diverse systems running the same software are effectively useless given that the average piece of hardware is much more reliable than any software. Look at PCs for example, where a hard disc has a mean time between failures of around 1,000,000 hours these days, whereas the hapless Windows falls over with a mean failure time of a few hours, (minutes if you are doing software development). (This also gives me the chance to repeat something I have been saying for two years now and is becoming more and more obvious. Linux appears to be at least 2 orders of magnitude more reliable than any form of Windows, so it is possible to produce robust software).

d) Lastly, and the point I'm really coming to, is that the programming language is only one part of the jigsaw of systems behaviour. Here, a *strength* of the Ada language, inappropriately handled led directly to the spectacular failure, where a sloppier language might not have. The lesson here is that it is not so much the programming language which is important as the programmer's fluency in whatever language is in use and reasoning about systems behaviour from the point of view of the language alone can be a dangerously misleading practice.

Finally, I understand that because this was the first launch, some of the payload satellites travelled for free, which just goes to show that there is no such thing as a free launch.

AUTHOR BIO:
Les Hatton has been suffering from computer systems for 30 years. He is a software consultant and Professor of Software Reliability at the University of Kent. In October 1998, he was named in the 'world's leading 15 scholars of systems and software engineering' by the US *Journal of Systems and Software*.

SIDEBARS:
None.