Editor: Bertrand Meyer, EiffelSoft, 270 Storke Rd., Ste. 7, Goleta, CA 93117; voice (805) 685-6869; ot-column@eiffel.com

Design by Contract: The Lessons of Ariane

Jean-Marc Jézéquel, IRISA/CNRS
Bertrand Meyer, EiffelSoft

Several contributions to this department have emphasized the importance of *design by contract* in the construction of reliable software. Design by contract, as you will recall, is the principle that interfaces between modules of a software system—especially a mission-critical one—should be governed by precise specifications, similar to contracts between humans or companies. The contracts will cover mutual obligations (*preconditions*), benefits (*postconditions*), and consistency constraints (*invariants*). Together these properties are known as *assertions*, and are directly supported in some design and programming languages.

A recent $500 million software error provides a sobering reminder that this principle is not just a pleasant academic ideal. On June 4, 1996, the maiden flight of the European Ariane 5 launcher crashed, about 40 seconds after takeoff. Media reports indicated that a half-billion dollars was lost—the rocket was uninsured.

The French space agency, CNES (Centre National d'Etudes Spatiales), and the European Space Agency immediately appointed an international inquiry board, made up of respected experts from major European countries, which produced a report in hardly more than a month. These agencies are to be commended for the speed and openness with which they handled the disaster. The report is available on the Web, in both French and English (http://www.cnes.fr/actualites/news/rapport_501.html).

It is a remarkable document: short, clear, and forceful. The explosion, the report says, is the result of a software error, possibly the costliest in history (at least in dollar terms, since earlier cases have cost lives).

Particularly vexing is the realization that the error came from a piece of the software that was *not* needed. The software involved is part of the Inertial Reference System, for which we will keep the acronym SRI used in the report, if only to avoid the unpleasant connotation that the reverse acronym has for US readers. Before liftoff, certain computations are performed to align the SRI. Normally, these computations should cease at –9 seconds, but because there is a chance that a countdown could be put on hold, the engineers gave themselves some leeway. They reasoned that, because resetting the SRI could take several hours (at least in earlier versions of Ariane), it was better to let the computation proceed than to stop it and then have to restart it if liftoff was delayed. So the SRI computation continues for 50 seconds after the start of flight mode—well into the flight period. After takeoff, of course, this computation is useless. In the Ariane 5 flight, however, it caused an exception, which was not caught and—boom.

The exception was due to a floating-point error during a conversion from a 64-bit floating-point value, representing the flight's "horizontal bias," to a 16-bit signed integer: In other words, the value that was converted was greater than what can be represented as a 16-bit signed integer. There was no explicit exception handler to catch the exception, so it followed the usual fate of uncaught exceptions and crashed the entire software, hence the onboard computers, hence the mission.

This is the kind of trivial error that we are all familiar with (raise your hand if you have never done anything of this sort), although fortunately the consequences are usually less expensive. How in the world can it have remained undetected and produced such a horrendous outcome?

> How in the world could such a trivial error have remained undetected and cause a $500 million rocket to blow up?

YOU CAN'T BLAME MANAGEMENT

Although something clearly went wrong in the validation and verification process (or we wouldn't have a story to tell), and although the Inquiry Board does make several recommendations to improve the process, it is also clear that systematic documentation, validation, and management procedures were in place.

The software engineering literature has often contended that most software problems are primarily management problems. This is not the case here: the problem was a technical one. (Of course you can always argue that good management will spot technical problems early enough.)

YOU CAN'T BLAME THE LANGUAGE

Ada's exception mechanism has been criticized in the literature, but in this case it could have been used to catch the exception. In fact, the report says:

> *Not all the conversions were protected because a maximum workload target of 80% had been set for the SRI computer. To determine the vulnerability of unprotected code, an analysis was performed on every operation which could give rise to an ... operand error. This led to protection being added to four of [seven] variables ... in the Ada code. However, three of the variables were left unprotected.*

YOU CAN'T BLAME THE DESIGN

Why was the exception not monitored? The analysis revealed that overflow (a horizontal bias not fitting in a 16-bit integer) could not occur. Was the analysis wrong? No! It was right *for the Ariane 4 trajectory*. For Ariane 5, with other trajectory parameters, it did not hold.

YOU CAN'T BLAME THE IMPLEMENTATION

Some may criticize removing the conversion protection to achieve more performance (the 80 percent workload target), but this decision was justified by the theoretical analysis. To engineer is to make compromises. If you have proved that a condition cannot happen, you are entitled not to check for it. If every program checked for all possible and impossible events, no useful instruction would ever get executed!

YOU CAN'T BLAME TESTING

The Inquiry Board recommends better testing procedures, and it also recommends testing the entire system rather than parts of it (in the Ariane 5 case the SRI and the flight software were tested separately). But even if you can test more, you can never test all. Testing, as we all know, can show the presence of errors, not their absence. The only fully realistic test is a launch. And in fact, the launch *was* a test launch, in that it carried no commercial payload, although it was probably not intended to be a $500 million test.

YOU CAN TRY TO BLAME REUSE

The SRI horizontal bias module was indeed reused from 10-year-old software, the software from Ariane 4. But this is not the real story.

BUT YOU REALLY HAVE TO BLAME REUSE *SPECIFICATION*

What was truly unacceptable in this case was the absence of any kind of precise specification associated with this reusable module. The requirement that the horizontal bias should fit on 16 bits was in fact stated in an obscure part of a mission document. But it was nowhere to be found in the code itself!

One of the principles of design by contract, as earlier columns have said, is that *any* software element that has such a fundamental constraint should state it explicitly, as part of a mechanism present in the language. In an Eiffel version, for example, it would be stated as

```
convert (horizontal_bias:
 DOUBLE): INTEGER  is
 require
   horizontal_bias
     <= Maximum_bias
 do
   ...
 ensure
   ...
 end
```

where the precondition `(require...)` states clearly and precisely what the input must satisfy to be acceptable.

Does this mean that the crash would automatically have been avoided had the mission used a language and method supporting built-in assertions and design by contract? Although it is always risky to draw such after-the-fact conclusions, the answer is probably yes:

- Assertions (preconditions and postconditions in particular) can be automatically turned on during testing, through a simple compiler option. The error might have been caught then.
- Assertions can remain turned on during execution, triggering an exception if violated. Given the performance constraints on such a mission, however, this would probably not have been the case.
- Most important, assertions are a prime component of the software and its automatically produced documentation ("short form" in Eiffel environments). In a project such as Ariane, in which there is so much emphasis on quality control and thorough validation of everything, assertions would have been the quality assurance team's primary focus of attention. Any test team worth its salt would have checked systematically that every call satisfies every precondition. That would have immediately revealed that the Ariane 5 software did not meet the expectation of the Ariane 4 routines that it called.

The Inquiry Board makes several recommendations with respect to software process improvement. Many are justified; some may be overkill; some would be very expensive to put in place. There is a more simple lesson to be learned from this unfortunate event: Reuse without a precise, rigorous specification mechanism is a risk of potentially disastrous proportions.

> There is a simple lesson here: Reuse without a precise specification mechanism is a disastrous risk.

It is regrettable that this lesson has not been heeded by such recent designs as IDL (the Interface Definition Language of CORBA)—which is intended to foster large-scale reuse across networks but fails to provide any semantic specification mechanism—Ada 95, or Java. None of these languages has built-in support for design by contract.

Effective reuse requires design by contract. Without a precise specification attached to each reusable component—precondition, postcondition, invariant—no one can trust a supposedly reusable component. Without a specification, it is probably safer to redo than to reuse. ❖