Computer Science 773

Robotics and Real-time Control

THINGS THAT GO WRONG

YOUR CONTROL SYSTEM WILL FAIL.

That is not necessarily absolutely true in all circumstances (for example, you can ensure that your system will not fail by never using it), but in practice it is the only sensible assumption to make when designing and constructing the system. It follows that you should give due attention to coping with failure when you design the system, and throughout the rest of the development and implementation. It is particularly important in real-time industrial systems, where computer programmes are controlling large, heavy, dangerous objects which move quickly and powerfully.

That being so, it is unfortunate that we have to admit that we don't know any really reliable ways to do it. We know of many precautions we can take to check our systems, both in design and in operation, to guard against the consequences of faults and errors, but so far we can never give a guarantee that a system will never fail.

A significant problem is that as we incorporate safety features in our systems the added complexity opens more opportunities for unexpected interactions, and the systems therefore can become more liable to failure rather than less. Failures can even be caused entirely by safety features. In 1996, the European Space Agency launched their new Ariane 5 rocket for the first time. It lifted off the ground, and rather quickly blew itself up at a cost denoted by a large number of zeros following a non-zero digit. (I've forgotten – it was a very large number.) As it happened, no one was hurt, which is perhaps the one bright spot in the affair. A few months after the event, I attended a seminar given by someone who had been involved in the post-mortem, and he described the cause of the failure in some detail. The sequence of events (partly from my memory, but recorded quite soon after the seminar so likely to be approximately right) was :

1 : The control software, written in Ada, was developed from the software from the foregoing Ariane 4 rockets.

2 : There is some component (call it X) in the Ariane 4 that isn't in the Ariane 5. (It's something to do with navigation, I think, but that part is hazy.)

3 : For one reason or other, the code for X was left in the software. As there was no X, it was obviously harmless, and in any case had been executed many times before in Ariane 4.

4 : But because of the new design the replacement for X was executed at a much higher altitude than X.

5 : So when the X code was executed, it received a value for the altitude which was much greater than it had ever had before. Of course, that didn't matter, because X wasn't being used.

6 : But within the X code there was an Ada calculation involving 64-bit to 32-bit conversion. This was the only Ada calculation left unchecked in the code; all others were explicitly checked for sensible values, but nothing could go wrong with the simple conversion. (It might even have been that no conversion was necessary in

the original version, as 32-bit numbers were sufficient there; that's another hazy bit.)

7 : Provided, that is, that the calculation was performed with the expected altitude figures. With the larger figures received in the Ariane 5 run, an overflow error of some sort occurred and an exception was signalled.

8 : As no specific check had been provided, the exception was handled by the general exception handler. Of course, as an unidentified exception, it had to be taken very seriously indeed.

9 : The exceptions check worked perfectly, so the exception was handled, even though it was meaningless.

10 : The check switched execution over to an alternative system – which was identical, so exactly the same thing happened again.

11 : The final exception check also worked perfectly, and blew up the rocket.

THINGS THAT CAN GO WRONG.

In discussing system failure, it is useful to distinguish between three different, though related, phenomena. Vocabulary usage seems to be not entirely standard in such discussion, but I'll try to stick to these definitions :

Errors are mistakes made when designing or constructing or using a system. They are in principle avoidable, though that isn't necessarily easy in practice. It was an error to leave the Ariane 4 code in the Ariane 5 software.

Faults are defects in a system which might cause it to misbehave. These might be the results of errors, or they might be accidental. The presence of the Ariane 4 code was a fault.

Failures are instances of the system behaving in a manner which is not intended in the design.

Any failure is caused by a fault of some sort, though a fault need not cause a failure under all conditions. A fault might even not be a fault under all conditions; the unchecked Ada operation wasn't necessarily a fault in the Ariane 4 code, because there were other good reasons for knowing that the potential exception would not occur. It only became a fault when it was used in conditions not expected when the code was written.

The aim of safety precautions so far as software design and development are concerned is therefore threefold :

1 : To guard against errors;

2 : To check for faults where possible;

3 : To detect failures, and to devise means of dealing with them.

None of these aims is simple. Just what was the cause of the Ariane explosion, and how would you design a system in which it couldn't occur ?

GUARDING AGAINST ERROR.

In design : careful checks performed on protocols, documentation standards, etc.

In data provision : consistency checks on incoming data.

In execution : fault-tolerant systems, both hardware and software.

CHECKING FOR FAULTS.

If faults are either the consequence of errors or of causes beyond our control, there is little enough that we can do about them directly. What we can do, sometimes, is try to identify them before they cause trouble. We can think of this as following the precedent of the inspection routines developed by engineers to identify faults in machinery before they lead to failures; systems are regularly inspected and tested to check for signs of defective behaviour. In much the same way, we can design computer systems so that they regularly check communications, interfaces, devices, and so on to make sure that they're there and apparently working using whatever tests might be appropriate.

DETECTING FAILURE.

IF YOU CAN'T DETECT IT, IT ISN'T A FAILURE.

– or, at least, there's no point in worrying about it. So the first step is to make sure that you can detect any specific foreseeable failure, and as many of the others as you can manage. (You can often tell that something's wrong even if you don't know what it is because some sensor reading, or combination of readings, has an impossible value.)

A convenient classification of failures results if we think about where the failures are detected. This turns out to identify classes of failure which must be handled in rather different ways.

A failure **detected by the plant**, in the sense that there's something like an alarm wire from plant to controller which means "the furnace is too hot", is usually rather specific. That's because the wire comes from a device which someone has built into the plant specifically to detect some condition which is known to be dangerous. You therefore have a rather good idea of what has gone wrong, and possible corrective actions are likely to be fairly easy to identify. In one sense, the emergency signal is just another input from the plant, and you write the software to cope with that just as you would to cope with any other signal.

A failure **detected by computer software** is often also fairly specific. It can be the result of a consistency check, an assert statement, a timeout condition, or other such planned test. Because of the known context and the nature of the variables concerned, there is quite a bit of context which can be used to diagnose the fault, though as compared with a direct indication from the plant the diagnosis might be less simple – all you get from a consistency check is an indication that at least one of the variables concerned is wrong, and it might not be easy to decide which.

A failure **detected by computer hardware** is typically much less specific. It is likely to be something like an arithmetic overflow, addressing error, or other similar condition, and all we know about it is the code address. In principle that's very useful; if someone gave you the high-level code, pointed to a position therein, and said "the failure happened here", you would have some valuable information. In practice, programming ways to use the information instantly in the code is extraordinarily difficult, and it is not easy to do better than the Ariane code and blow up the rocket.

MANAGING FAILURE.

Once you've detected a failure, what do you do ? Details depend on the sort of failure, how well you can diagnose the fault which caused it, and so on, but there is a general goal which applies to any real-time system which is potentially dangerous :

THE FIRST PRIORITY IS TO REGAIN CONTROL.

That's a brave general goal, but it lacks a correspondingly brave general technique which can be used to achieve the goal. Even so, we can proceed with a fairly general description for a few steps, and the result is a helpful view of the very complicated problem.

To do so, we think of the controlled system as a finite-state machine. In practice, this is almost always possible, even though the machines are often full of continuously varying parameters which suggest that there's little finite about the states. Usually, the precise values of the parameters are not particularly important, and it is more useful to think in terms of what the system is doing at the time. Consider the example of a car; at any moment, it has real-valued parameters such as position, velocity, direction, acceleration, and so on, but from the point of view of controlling the car it is much more useful to think of much broader categories such as turning left, accelerating, going downhill, and so on. This is how we think of the car's motion when driving, and for many purposes it's a much more useful notion of state. (Recall the system states used in Britton's technique for drawing up the system specification document.) In particular, if we use these categories to define the states we find that there are only a finite number.

Given these states, which we can determine by measuring the parameters and classifying them, we can define a state transition diagram, and identify possible state transitions and the control operations which cause the transitions. The effectiveness of this sort of description as the basis for control is demonstrated by the success of fuzzy control methods, which adjust their behaviour according to exactly these state relationships.

In terms of these states, then, how do we describe a failure ? It is an unexpected transition from a known state to some other state which might or might not be known. Recovery from failure is then, in the first instance, a matter of identifying the unexpected state, and then using the available control functions to move from there to a known state, where control can be regained.

Of course, things are not as simple as that. After a failure, you have no guarantee that the parts of the system on which you rely for sensing and control will still be usable, and in any case you have to determine whether or not they are. Nonetheless, the state description is a useful aid to thinking about the problem. For example, it follows from the second sentence of this paragraph that if there are sensors which you need to determine the state, then it would perhaps be sensible to duplicate them and provide code which can determine whether either is reliable.

Alan Creak,
April, 1997.