Assessing the Dependability of Softwarebased Systems: A Question of Confidence

Bev Littlewood Centre for Software Reliability, City University, London b.littlewood@csr.city.ac.uk

email me if you want a copy of my presentation



Do you remember 10⁻⁹ and all that?



- Twenty years ago: much controversy about need for 10⁻⁹ probability of failure per hour for flight control software
 - could you achieve it? could you measure it?
 - have things changed since then?



Issues I want to address in this talk

- Why is dependability assessment still an important problem? (why haven't we cracked it by now?)
- What is the present position? (what *can* we do now?)
- Why is 'confidence' in claims so often ignored?
- Where do we go from here?



Why do we need to assess reliability?

Because all software needs to be sufficiently reliable

- This is obvious for some applications e.g. safety-critical ones where failures can result in loss of life
- But it's also true for more 'ordinary' applications
 - e.g. commercial applications such as banking the new Basel II accords impose risk assessment obligations on banks, *and these include IT risks*
 - e.g. what is the cost of failures, world-wide, in MS products such as Office?
- **Gloomy personal view:** where it's obvious we *should* do it (e.g. safety) it's (sometimes) too difficult; where we *can* do it, we don't...



What reliability levels are *required*?

- Most quantitative requirements are from safety-critical systems. Even here the figures vary dramatically.
- Some are extremely stringent
 - e.g. civil aircraft flight control (A320/330/340, B777, etc): 10⁻⁹ probability of failure per hour
 - e.g. railway signalling and control: 10⁻¹² probability of failure per hour!
- Some are quite modest
 - e.g. UK Sizwell nuclear reactor's primary protection system: 10⁻³ probability of failure on demand
 - e.g. surgical robotics: the humans they replace have only modest reliability!
- Seems likely that for *non-safety-critical* systems, the range is just as great



Why uncertainty?

'Software failures are systematic - if it fails in certain circumstances it will always fail in those circumstances'

- True. But it is uncertain when those circumstances will occur
- There is inherent uncertainty about the process of inputs
- 'Systematic' here does not imply 'deterministic'



Why probabilities?

There are other ways of dealing with uncertainty: why not use, e.g. Dempster-Shafer, fuzzy/possibility theory, etc

- Advantages of probability
 - advanced and well-understood formalism
 - widely accepted and used already, so easy to incorporate into existing frameworks, e.g. risk analysis, e.g. wider safety cases, etc
- Problems
 - can be very hard to estimate the numbers (see later comments)
 - but no easier for other formalisms?



What drives software unreliability?

'Why doesn't software work perfectly? it's only logic, after all, why don't you just do it right?'

- Novelty: software is often (usually) used to implement radical new functionality
 - 'if we *can* do it, we *will* do it'
- **Difficulty:** some of the problems that software designers have to solve are intrinsically hard
 - we routinely build things that would be *unthinkable* in any other technology
- **Complexity:** these trends often result in unnecessary complexity in the design solutions
 - e.g. not constrained by the reliability implications of *hardware* complexity



So how bad are things?

If we can't make real programs fault-free, how many faults can we expect?

- What are achieved fault densities?
 - even for safety-critical industries, 1 fault per kLoC is regarded as first class
 - + e.g. study of C130J software by UK MoD estimated 1.4 *safety-critical faults* per kLoC (23 per kLoC for non-critical)
 - for commercial software, studies show around 30 faults per kLoC
 - + Windows XP has 35 MLoC, so >1 million faults!
- Is there no good news here...?!



Many faults means very unreliable?

NOT NECESSARILY!

- Windows reliability has grown from 300 hours MTBF (with 95/98) to about 3000 hours *despite increased size and complexity* (i.e. more faults)
- Operational experience with software in aircraft and automobiles suggest very high reliabilities *can* be achieved
 - After-the-fact estimation of failure rates, based on very extensive usage:
 - + Automobiles: Ellims has estimated that no more than 5 deaths per year (and about 300 injuries) caused by software in the UK - suggests about 0.2 x 10⁻⁶ death/injury failures per hour. Even better *per system* - say 10⁻⁷
 - + Aircraft: *very* few accidents have been attributed to software; Shooman claims, again, about 10⁻⁷ per hour per system



Why can software be so reliable...

...when it contains thousands of faults?

• Because many (most?) faults are 'very small'

- i.e. they occur extremely infrequently during operation

- Adams more than twenty years ago examined occurrence rates of faults on large IBM system software: found that more than 60% were '5000-year' bugs
 - i.e. each such bug only showed itself, on average, every 5000 years (across a world-wide population of many users)
 - the systems he studied had many thousands of these faults, but were acceptably reliable in operation



So what's the problem?

Is there a problem?

- Just because large complex programs *can* be very reliable, it does not mean you can assume that a particular one *will* be
 - even if you have successfully produced reliable software in the past, you can't assume from this that a *new* program will be reliable
 - even if some software engineering processes have been successful in the past, this does not *guarantee* they will produce reliable software next time
- So you need to measure how reliable your software *actually* is
- And this assessment needs to be carried out *before* extensive real-life operational use
 - how else can you make a risk assessment?



So how can we assess dependability?



Direct evaluation: Operational testing

- Statistical methods based upon operational testing are the only means of *direct* evaluation of reliability
 - allow estimation and prediction of such measures as *mtbf*, *reliability function* (i.e. probability of surviving failure-free for time t), *failure rate*, etc
- They require
 - means of generating test data that is *statistically representative of operational use*
 - an *oracle* to allow unacceptable ('failed') output to be detected
- The resulting process of 'acceptable' and 'unacceptable' outputs is used to estimate and predict reliability



Reliability growth modelling

- Huge literature developed over the past 25 years
- Idea here is that faults are fixed as they are identified in operational testing
 - so reliability grows
- Many sophisticated probability models have been developed
 - aim is to predict future failure behaviour from observation of the past
 - none of the models can be trusted to be accurate *a priori*
 - but ways of telling whether a model is accurate *in a particular context*
 - also ways of 'learning from past errors'
- Sophisticated tools for doing this
 - e.g. our 'PETERS' tool, and several others
 - the bottom line here is that you can generally obtain trustworthy results from this approach, *and know that the results are trustworthy*



Example of real software failure data



CITY City University London

Cumulative plot of same data



CITY City University London

Successive median predictions



CITY City University London

What sort of claims can you make?

It turns out that this kind of evidence of reliability growth from testing only allows quite weak claims

- e.g. if you want to claim *mtbf* of x hours, you will typically need to see 10s or 100s times x hours on test
- even worse, there is a very strong *law of diminishing returns* operating
 - you may end up with very many very small faults, each of which is very hard to find
- what about the 'best possible' case where no failures at all are seen in *x* hours of test?
 - even here, you can only make modest claims for future behaviour
 - + e.g. only about a 50:50 chance of seeing further *x* hours failure-free operation before failure.



So what can we do?

In fact, of course, there is always lots of other information available, in addition to operational test data

- e.g. quality of software engineering process used
- e.g. information from static analysis
- e.g. expert judgement, etc
- we need ways of combining such disparate evidence in formally reasoned *dependability cases* to support (*probabilistic*) dependability claims



Dependability 'case'

Informally, a dependability case is an **argument**, based on **assumptions** and **evidence**, that supports a dependability claim at a particular level of confidence

- For a particular claim (e.g. the probability of failure on demand of this system is better than 10⁻³), your confidence in the truth of the claim depends on:
 - strength/weakness of evidence (e.g. the extensiveness of the testing)
 - confidence/doubt in truth of assumptions
- Conjecture: assumption doubt is a harder problem to handle than evidence weakness



But evidence in SE is weak

- E.g. software engineering 'process' evidence is often used to support product dependability claims
- But it is *very* weak
- We have some evidence to support inference of the kind process->product (e.g. fault count)
- We don't have much evidence to support inference of the kind process->product->product_reliability (e.g. failure rate)



A promising formalism for 'cases': BBNs



- manageable visual description, automated Bayesian inference
- describe prior knowledge about
 - development quality
- modify with inference from observed failures
- propagate to prediction about lifetime reliability



Test

BBNs - advantages and disadvantages

- Advantages
 - powerful aid to reasoning
 - computational algorithms now make 'proper' analysis feasible
 - allow expert knowledge to be incorporated (i.e. combine judgement and empirical evidence) - this seems vital for software-based systems
- Limitations/disadvantages
 - permit over-enthusiastic use of expert knowledge!
 - humans are *very* poor at expressing probabilistically their beliefs about uncertainty
 - even the *topologies* of BBNs are not easy to construct
- Current position: use with caution and humility
 - e.g. be more willing to trust simple BBNs than complex ones



Analogy between arguments and systems

There's an analogy between arguments and systems. Think of a protection system, and an argument supporting a dependability claim:

- Both can 'fail'
 - argument: accept claim when it's false (or reject when true)
 - system: fails to trip when it should (or trips when it shouldn't)
- Confidence in argument <-> reliability of system
 - both concern probability of not failing

You would not want to push the analogy too far, but can we exploit 'system tricks' to improve arguments? E.g.....



Dependability case 'fault tolerance'

Can we borrow ideas from *system* fault tolerance? 'Argument diversity' as analogy of 'system diversity'?

- Multi-legged arguments to increase confidence in claim(s)
 - leg *B* could overcome evidence weakness and/or assumption doubt in leg *A*
 - legs need to be *diverse*
 - advocated in some existing standards (but only informal justification)
 - we are trying to formalise this via special BBN structure
 - + same 'independence' issues as for systems



Summary; and where do we go now?

- The *need* for trustworthy dependability assessment continues even grows
- For many situations essentially those with modest requirements trustworthy evaluation of software reliability is possible
- Great difficulties when required levels are very high
 - the 10⁻⁹ problem remains unsolved, and is likely to remain so
- We need
 - more and better evidence
 - better, more formal, ways of reasoning about disparate evidence in 'cases' to support dependability claims
 - in particular, formal treatment of 'confidence' in claims



Some new, harder problems loom

We need a much more holistic approach

- Beyond 'reliability and safety', to incorporate security
 - very little work has been done on problem of (probabilistic) security assessment
 - but some of the reliability techniques probably apply
 - need to be able to understand trade-offs
- Beyond 'software and computers'
 - it's very rare for systems to be purely 'technical' there are almost always humans and organisations involved, and the whole system needs to be addressed
 - interactions here can be complex and counter-intuitive
 - require collaboration with psychologists, sociologists, etc



THE END

(with no apologies for being so gloomy)

