Formal Methods, Testing, and Reuse - towards Reliability Conservation for Software

Pat Hall, Tom Gedeon, Chris Reade
Department of Computer Science,
Brunel University

The Reliability Problem

Safety-critical systems require ultra-high reliability. Figures of demand reliability of 10^{-9} have been cited - but how could this ever be measured? Demand reliability of 10^{-n} requires 10^{n+1} test cases (eg Rook 1990) - where n=9, tests at one per second would take 300 years! Reliability of continuously running processes takes equally long to establish. So how can we ever have software systems with proven reliability of the required levels?

Even if we don't actually measure the reliability, we want assurance that these figures are achieved. Formal methods and verification have been claimed to produce systems of this calibre, as indicated int he ACARD report (1986), and the draft 00-55 (MoD 1989) mandating formal methods. But no evidence has been advanced that these methods do actually achieve these levels of reliability.

How might we obtain assurance that formal methods do indeed deliver the reliability promised? One approach might be to carry our controlled trails, developing a number of systems and then measuring the reliability of these to thereby assess the reliability of the methods. But that just compounds the problem - we have to conduct not just one but many of those 300 year experiments outline above.

What can we?

In one way or another we must exercise our code, both to the connection of the software to the hardware (cf. Fetzer's (1988) argument), and to measure reliability. Could we perhaps mix testing and formal methods, in some way to squeeze more reliability estimation out of each test case? A simple idea might be to use formal reasoning to "spread" each test case throughout its behavioural domain, as in black box testing (eg. Hetzel 1984), but this is not where the problem lies - it lies in the sheer size and complexity of the software that we produce and the formal reasoning processes that are required.

Reliability Conservation through Reuse

So what can we do? Let us turn to traditional engineering technologies: how do the guarantee these ultra-high levels of reliability? There seem to be two ways:
(i) build the system from parts with proven reliability, and then calculate the reliability of the whole from that of the parts and the rules of composition
or
(ii) use complete systems proven through long use, permitting small changes, and using arguments like " we have always done it that way".

This is building new systems from previous systems or parts of them, this is software reuse (eg. Hall and Boldyreff, 1989). For software, we do not have large collections of parts other than in special applications, like libraries of mathematical routines, though this in improving. It is now considered good practice to record the quality and reliability for these parts (eg. Moineau, Abadir and Rames, 1990). The methods of composition can be very complex, as manifest in Module Interconnection Languages (eg. Prieto-Diaz and Neighbors 1986), and methods of calculating the reliability of the whole from that of the parts is not yet possible (Mellor 1987) A purely components approach to the construction of demonstrably reliable systems does not seem promising.

So could we take existing complete software systems and redeploy these? This is re-engineering (eg. Chikovsky and Cross, 1990). And in doing this re-engineering, could we conserve the reliability of these systems? This does seem promising, for consider some base cases:

- we reuse a complete system that has been running without problems for many years, without any changes whatsoever - do we carry over the proven reliability into the new application? Well, not quite, for the operational patterns may vary, and as jaundiced developers of software know, bringing a new user on-stream always throws up new errors. But it is close. And we could view the issue as one of training the users to use the system reliability, the bugs become features.
- we enhance a stable system in small ways for the same set of users, and thoroughly regression test the changes. Clearly if the system was assessed as highly reliable beforehand, then it will be so afterwards. What the precise change in reliability might be needs investigation, but clearly a lot of reliability does carry over.

For new systems, what do we do? Well, most systems are not that new, at some level they rely on generic systems architectures, into which particular generic parts are slotted. These generic parts will have clean interfaces which provide a focus for specification, proof, and reliability measurement. Both these generic architectures and the parts will have been proven in practice - so why can't we just put the system together, and know that it will be reliable? It is not that simple, we inevitably make some changes, and need to understand the effects of these changes:

- what is the reliability of the parts that we use? These may have been extracted from some existing system, and we could instrument the system and discover how the part was used and how it contributed to the reliability of the whole. We need a way of deducing the reliability of the part from the reliability of the whole, turning the earlier desire to calculate reliabilities on its head.
- we abstract (reverse engineer) the parts and the system structure form the original, changing them in the process. We need to use proven (ie. formally verified and thoroughly tested) transformations (eg. Gedeon 1989) - and confront the issue of to what extent can we trust these transformations. Do we go into an infinite regress, or can we produce more reliable systems using less reliable tools?
- when we add new parts to the system, we will need to prove these new parts correct, and test them and the whole system. Test cases may be derived from the formal specifications of the modules, test suites may be controlled by systems extracted from the formal specifications of the system.

For a reliability conservation programme, we need to bring these threads together. Can we blend the tests with formal reasoning and guarantee the conservation of reliability of the original system as we move to the new system? This is how traditional engineering guarantees ultra-high reliability, and it seems the only way that we will be able to do so for software.

References

ACARD, **SOFTWARE. A vital key to UK competitiveness**, Cabinet Office Advisory Council for Applied Research and Development (ACARD), HMSO, 1986.

Chikovsky, E.J. and Cross, J.H.II, *Reverse Engineering and Design Recovery: a Taxonomy.* **IEEE Software.** January 1990

Fetzer J.H., 1988. *.Program Verification: the Very Idea*, **CACM** vol 31, No 9, Sept 1988. pages 1048-1063.

Gedeon, T.D. **IMP: An Interactive Program Transformation System**, PhD Thesis, The University of Western Australia, 1989.

Hall, Patrick and Cornelia Boldyreff, *Software Reuse Overview*, in **Reuse, Maintenance, and Reverse Engineering of Software**, Unicom seminar, Dec. 1989

Hetzel, William. **The Complete Guide to Software Testing.** Collins, 1984.

Mellor, Peter, *Software Reliability Modelling: the state of the art.* **Information and Software Technology Journal**, March April 1987

MoD, Interim Defence Standard 00-55 (draft), **Requirements for the Procurement of Safety Critical Software in Defence Equipment.** 1989.

Prieto-Diaz, Ruben and James Neighbors, *Module Interconnection Languages*, **Journal of Systems Sciences**, vol 6, no 4, November 1986, pages 307-334.

Rook, Paul. **Software Reliability Handbook,** (Editor) Elsevier 1990