Some interdisciplinary observations about getting the "right" specification

Cliff B Jones

Newcastle University
Newcastle, NE1 7RU, UK
cliff.jones@ncl.ac.uk

One can use formal approaches either *post facto* to try to show that a program has desirable properties or one can aim for *verified by construction* (VxC). The former approach tends to focus on specific properties such as avoiding the dereferencing of null pointers; the latter is more likely to address the question of whether the steps of design satisfy some overall specification. I not only prefer the latter but I have also argued that this is the main way to get formal methods to pay off: there is more mileage in getting a clean architecture than in trying to debug a bad design by retrofitting a proof.

I think VxC is also a way to choose an appropriate level of formality perhaps using outline arguments and filling in details if doubt arises (see [Jon96]; Jackson and Wing made a similar point in the same journal; [Jon05] makes a similar point related to proofs).

But we must also face the crucial question "how do we know that the specification is right?". This is not a trivial question especially with the way computers are used today. As computers have become more powerful and less expensive, they have become ever more deeply embedded in the way nearly everyone works. In their short history, computers have moved from batch processors in their own buildings to work tools on every desk (or lap). They are now essential components of administration, retail trade, banking and vehicles; computers in the future will become invisible dust sprinkled on who-knows-what. This has transformed the task of understanding the *requirements* of a system. Above all, the close interaction of people with computer systems makes it essential that designers consider the *whole system* when formulating a specification of the technical parts. This larger system involves people as essential components.

Model-oriented specification techniques like VDM, Z, ASMs and B have an enormous amount in common; among other things shared by this formal methods community is the view that one can start with a formal specification and show that a design/implementation satisfies that specification. It is obvious however that, if a specification does not actually reflect the real need, proving a program correct with respect to it is somewhat pointless. Am I arguing in favour of "XP" or fluid prototyping? Certainly not — at least not for most applications. But one might have to proceed in this way if we were to decide it's impossible to get the right specification.

I strongly believe that, for a crucial set of computer uses, one can –and must– start with a careful process of establishing a good specification. Mine is not a council of despair; I want to see how we can use technical ideas to improve the

process of getting to a specification. In particular, some of the ideas below relate strongly to formal methods.

The point about where effort will have the greatest effect on dependability can be made by looking at accidents: Donald MacKenzie in [Mac94,Mac01] has traced the cause of just over 1100 deaths (up to 1994) where computer systems appear to be implicated. Only three percent of the lives lost appear to be attributed to program bugs! Far more common causes of accidents appear to be situations where humans misunderstand what is going on in a control system or the object being controlled. This is a much deeper issue than the details of HCI; in many cases it is a fundamental question of the allocation of tasks between person and machine. Key questions include the visibility of the "state" of the system being controlled and the extent to which operations which the user can perform are grouped together.

Although accidents are shocking and grab attention, there is also a significant penalty in the deployment of systems which make their users' lives more difficult than they need be. The enormous cost (often to the taxpayer) of systems which are so unusable that they are not even deployed is reported all too often in newspapers.

Of course, we should use formal specification and design techniques for fault avoidance and we still need research to make them more widely usable. (I have contributed to several sets of tools in this area including [JJLM91].) But it would also appear to be worthwhile to see whether there is a *technical* response to the question of how one arrives at a specification which does reflect the needs of the environment in which a system will be embedded. Does the formal methods community have a contribution to make here? I believe so.

This paper sets out some research challenges to which we might be able to offer useful responses. The suggestions have arisen from the six year "Interdisciplinary Research Collaboration on Dependability" (DIRC) — see the WWW pages at [WWW06] for further details. DIRC is focusing its research on how to design *Dependable* computer-based systems. The phrase "computer-based systems" is intended to emphasise that most computer systems today are deeply embedded into an environment which also involves people. For example, the requirement in a hospital is for dependability of the overall system. In such domains, humans will use a computer system to achieve objectives even where they know that it delivers less than perfect information; on other occasions, computers can be programmed to warn when errors appear to be made by humans. People are less good than computers at narrowly specified repetitive tasks but are much better at recognising and reacting to exceptional situations. To achieve overall system dependability, both humans and programs must be properly deployed.

Some of the insights from the DIRC project include:

Determining specifications An approach being worked on with Ian Hayes and Michael Jackson [HJJ03,JHJ06] looks at determining the specification of, say, a control system by first specifying a wider system including the phenomena of the physical world which are to be influenced. To avoid having to build a model of the behaviour of all physical components, assumptions about their behaviour

are recorded using *rely conditions*. This leaves a clear record of assumptions which need to be considered before the control system is deployed. Development from the derived specification of the control system is conducted in the standard (formal) way. (Dines Bjørner's books [Bjø05] tackle "domain modelling".)

Limiting failure propagation The design of boundaries that limit the *propagation of failures* is better articulated for technical systems than for the human part of computer-based systems. This is odd because the intuition about limiting, say, accounting errors by auditors is long established. Many examples can be cited to suggest that most human systems are "debugged" rather than designed. The motivation for where to place containment boundaries ought to come from an analysis of the frequency of minor faults and the danger of their affecting a wider system. This analysis ought to precede the allocation of tasks to computers which, in turn of course, must be done prior to their specifications being "signed off".

Cognitive mismatch A major cause of near or actual accidents is a "cognitive mismatch"[1] between an operator's view of what is going on and the actual state of affairs in the system the operator is trying to control. This was a significant factor in the Three Mile Island reactor incident. John Rushby [Rus99] has looked at pilot errors on the MD-88: in simulators, they frequently breach the required altitude ceiling. Rushby's careful formal analysis builds a state model of the pilot's understanding of the system and explores its interaction with a model of the aircraft systems. (It would be informative to compare this approach with rely conditions.)

The role of procedures The general way in which *processes* (or procedures) are used in the human parts of computer-based systems is interesting. If one contrasts a traditional car production line with the depiction in the film *Apollo 13* of the search for a solution to the need to improvise CO_2 scrubbers in the damaged capsule, one sees that processes both limit action and reduce the need for information. Designing processes which cope with all exceptions is in many cases impossible and one argument for relying on humans in computer-based systems is precisely that they notice when it is safer to violate a procedure than slavishly to follow one that does not cover an exceptional case. Clearly, either following an inappropriate process or deviating from a correct process can lead to system failure. But it is absolutely mandatory that thought is given to processes in the design of a computer-based system. Interestingly, one can spot errors in legislation where an algorithmic rule is frozen into law: there have been several cases in financial legislation where a well-intentioned trigger has had (or nearly had) counter-productive effects. A recent DIRC book [Mac06] addresses financial markets from this perspective.

[1] Both of James Reason's books [Rea90,Rea97] look at relevant issues: the earlier reference looks at a division of the sort of errors that humans make; the second has insightful analyses of many system failures. Perrow in [Per99] talks of "Normal accidents".

Advisory systems Within DIRC, the role of *advisory systems* has received particular attention: [SPA03] studies a prompter used in the analysis of mammogram images. Surprising conclusions include statistically significant evidence that, under the tested conditions, the most accurate operators offered *less accurate* conclusions with the help of the advisory system than without its use. It is clear that the role of such advisory systems has to be considered far more widely than just by looking at their technical specifications. In fact, even pure safety limiters (where one would believe they can only increase safety) have been used by operators in a way which supplants their normal judgement.

Creating dependable systems Systems can create other things whose dependability is the goal. In the simplest case, a production line might manufacture silicon chips and faults in the manufacturing process might result in faulty components for computers. A software example is a compiler that, if faulty, could translate a perfect program into machine code which does not respect the formal semantics of the source language. In many cases, the creation process is human and, for example, a designer of a bridge which fails to withstand expected forces is at fault. The creation of computer software is just such a process and is not always fault free. DIRC has provided an opportunity to look at Gerry Weinberg's conjectures in [Wei71] that different psychological types might be more or less adept at different sub-tasks within the broad area known as programming [DG06]. The implications of this research for building dependable systems might include steering people toward the tasks at which they are likely to perform best (and probably be most content).

Evolution If the above list were not daunting enough (and it is far from complete even with respect to DIRC's findings) there is another overriding concern. The sort of computer-based system we have been studying will always *evolve*. Designing a system which can be modified in reaction to a reasonable class of evolutions in the environment is extremely challenging. One class of system which has been studied within the DIRC project is *generic systems*. The justification of this sort of system is that it can be instantiated for a range of applications: characterising this range is itself a technical problem (and a further challenges is trying to maximise the range). It is clear that issues around evolution will have a long-term impact on dependability. There are related questions about how data survives such evolution which are equally challenging.

DIRC has identified far more than the above set of issues; the selection here has been based on the ease with which this one member of a project (involving more than fifty researchers) could pull together the information.

One key experience from the project is the invaluable role of interdisciplinarity. Looking at experiments on psychological type and debugging performance required wholehearted collaboration of psychologists and computer scientists; tackling the mammography advisory system involved interaction between statisticians, sociologists and psychologists. DIRC researchers could list many more examples of how our combination of psychologists, statisticians, sociologists and

computer scientists has made real progress that no one of these disciplines could have accomplished.

My own inclination is to seek technical approaches to problems and I hope that the list above indicates that this is a viable challenge. But the DIRC project has been a superb example of collaboration and if faced with a complex application area, I would now know how to call on the expertise of other disciplines. In particular, the painstaking gathering of observational data needs sociologists.

One key message from our experience is to tackle application problems together as a team. With an "Operations Research" (OR) like team representing several disciplines terminology problems disappear, contributions become understood and something is achieved which no single discipline could have envisaged.

Acknowledgements

My research acknowledgement is to the many colleagues involved in DIRC; it is a privilege to lead such an exciting project.

We are all grateful to EPSRC for the six year funding window which we feel was essential to foster such a wide interdisciplinary span.

References

[Bjø05] D. Bjørner. *Software Engineering (3 vols.)*. Springer-Verlag, 2005.

[DG06] A Devito Da Cunha and David Greathead. Does personality matter? an analysis of code-review ability. *In press, Communications of the ACM*, 2006.

[HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[JHJ06] Cliff Jones, Ian Hayes, and Michael Jackson. Specifying systems that connect to the physical world. *Acta Informatica*, 2006. submitted.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[Jon96] C. B. Jones. A rigorous approach to formal methods. *IEEE, Computer*, 29(4):20–21, 1996.

[Jon05] C. B. Jones. Reasoning about the design of programs. *Royal Soc, Phil Trans R Soc A*, 363(1835):2395–2396, 2005.

[Mac94] Donald MacKenzie. Computer-related accidental death: an empirical exploration. *Science and Public Policy*, 21:233–248, 1994.

[Mac01] D. MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, Cambridge, Mass., 2001.

[Mac06] D. MacKenzie. *An Engine, Not a Camera: How Financial Models Shape Markets*. MIT Press, Cambridge, Mass., 2006.

[Per99] Charles Perrow. *Normal Accidents*. Princeton University Press, 1999.

[Rea90] James Reason. *Human Error*. Cambridge University Press, 1990.

[Rea97] James Reason. *Managing the Risks of Organisational Accidents*. Ashgate Publishing Limited, 1997.

[Rus99] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In *Proceedings of 3rd Workshop on Human Error*, pages 1–18. HESSD'99, 1999.

[SPA03] L Strigini, A. Povyakalo, and E. Alberdi. Human machine diversity in the use of computerised advisory systems: A case study. In *DSN 2003-IEEE International Conference on Dependable Systems and Networks*, pages 249–258, San Francisco, USA, 2003.

[Wei71] Gerald M. Weinberg. *The Psychology of Computer Programming*. Van Norstrand, 1971.

[WWW06] WWW. www.dirc.org.uk, 2006.