Feasibility of Using Formal Methods in the Development of e-Voting Systems

Anthony Hall
Praxis Critical Systems
anthony.hall@praxis-cs.co.uk
19th November 2003

Management Summary

We strongly agree with you that sensible deployment of formal methods would greatly increase the confidence in eVoting software. At the very least there should be a formal specification of the system and static analysis of the code to show that there are no undesirable flows of information. The formal specification would support a rigorous testing programme.
Formal Methods are a practical approach to developing systems of any size and complexity. We have built systems as big as the voting machine and IES, and far more complex, using formal methods.
The practical use of formal methods depends on making a sensible choice of what activities should be carried out formally and integrating these activities into the normal software lifecycle.

What should be done formally?

"Formal Methods" covers a wide range of activities. Not all of these are necessary or cost-effective on every project (contrary to the purist opinions expressed by some academics). Any or all of the following may be useful. The items marked * can be done automatically. Items marked (*) can be done automatically in part, but may need human intervention.

1) Formalising the requirements. In secure systems, for example, there is the concept of a "Formal Security Policy Model" which captures exactly what security properties the system should have. In the context of a voting system it would capture requirements like "the vote cannot be removed or altered once it has been recorded".
2) Formalising the system specification.
   a) This is typically done by giving an abstract model of the system state and an abstract model of the operations of the system in terms of their inputs, outputs, and effect on the state.
   b) Sometimes it is also useful to formalise the concurrent behaviour. This is typically done by specifying in a process algebra like CCS or CSP how different behaviours of the system (and its environment) can occur concurrently.
3) Formalising the design. This typically means writing specifications of the modules within the system. Again, it may also involve describing their interactions in process algebras.
4) Developing formally annotated code. Each operation in the code may have pre and post-conditions attached to it.
5) Defining the required flow relations on the code. In SPARK, for example, the information flow properties of the code are specified and checked as part of the detailed design process, preferably before the code itself is written.

6) Writing down the refinement relation between the formal design and the formal specification
7) *Generating proof obligations to show the code is free from run time errors.
8) (*)Proving that the system specification is satisfactory – for example, that it is consistent and that all operations preserve the state invariant.
9) (*)Proving or model checking that the system specification satisfies the formal requirements.
10) (*)Proving or model checking that the refinement relation is satisfied
11) (*)Proving that the pre and post conditions in the code satisfy the design
12) (*)Discharging the code proof obligations.
13) *Showing that the code satisfies the information flow conditions in 5)
14) (*)Deriving test conditions from the formal specification and design.

In our experience, the most cost-effective step is usually step 2, formalising the system specification. This specification is the pivotal point of the whole development – everything else really depends on it.

In addition, any automatic steps are worthwhile. For example, static analysis of the code is essentially free and is a good measure of the integrity of your module design.

The value of formalising the requirements depends on whether they are at all obscure and whether there is perceived to be any difficulty proving whether they are met or not. It would be interesting to see how important this might be in the eVoting context.

Formalising the design is useful if there is a complex relationship between the design and the specification, for example if a black box specification is implemented on a distributed or highly parallelised system. Otherwise, it may be perfectly feasible to go straight from a formal specification plus (informal) module architecture to well structured, traceable code.

Pre and post conditions in the code itself tend to be used at the highest levels of criticality. I don't know whether they would give significant benefits in an e-voting system. It is important to realise, though, that this is not an all or nothing choice. It may well be sensible to put pre and post conditions on particularly critical or difficult pieces of code, forming only a small percentage of the whole.

The cost-effectiveness of the various kinds of proof and analysis is again something that is flexible. We have done major projects in which there has been no proof at all, and others in which we have done significant proofs at both the specification and code level. In the latter case we have found that proof is an effective and efficient method of finding defects.

Practical Experience

Electronic voting is well within the scope of good industrial practice in formal methods. I understand that the voting machine contains about 25,000 lines of code and the Integrated Election Software about 200,000 lines. We have developed safety critical systems of about the size of the voting machine using the full rigour of DefStan 00-55, including extensive proofs [1]. We have developed an air traffic information system [2] of about 200,000 lines of code using formal specification and formal design, with almost no proof. We have developed a highly secure system of about 100,000 lines to the standards of ITSEC E6 with a formal security policy model, a formal specification, selective formal design and static analysis of the code [3,4]. It is noteworthy that, like the IES, this system was based on a COTS database product –yet it meets extremely stringent security requirements.

References

1) "Is Proof More Cost Effective Than Testing?" (PDF — 564Kb)
   Steve King, Jonathan Hammond, Rod Chapman and Andy Pryor,
   IEEE Transactions on Software Engineering, Volume 26 Number 8.
2) *Using Formal Methods to Develop an ATC Information System*, Anthony Hall, IEEE Software, March 1996, pp 66-76
3) "Correctness By Construction: Developing a Commercial Secure System" (PDF 174kb)
   Anthony Hall and Roderick Chapman, IEEE Software, Jan/Feb 2002, pp18-25
4) *Correctness By Construction: Integrating Formality into a Commercial Development Process*, Anthony Hall, Proceedings of FME '02, pp 224-233.