Where is the value in a Program Verifier?

The Systems Assurance Group at QinetiQ in Malvern has used mechanical proof techniques to verify three successive versions of Typhoon's flight control laws implemented in Ada. Approximately 37,000 lines of code were checked against three different Simulink[1] specifications scheduled over 3 processors. Each evolution of the control laws led to a change to about a third of the Ada source code each time. Just over 97% of the verification conditions were automatically proven. The remaining 3% were either manually discharged using a theorem prover, or could not be proven due to limitations in the tools at that time, or very occasionally were false due to mismatches between the specification and the code. Most of theses mismatches were due to the specification not being updated (the code in fact correctly implemented the requirements). Two important points arise from this experience: the first is that these implementations had already been subjected to a mature software verification and validation process when only once did a coding error slip through[2]; the second is that the requirements for the flight control laws have been continually evolving in the light of flight tests and increased aircraft capability.

There is plenty of evidence that poor or immature development processes have led to poor software products. More often commercial pressures lead to trade-offs between the time (and cost) spent developing software and the dependability of the final software product. The evidence from the Typhoon flight control law development suggests that rigorous conventional development is sufficient for most purposes to achieve the desired reliability assuming the software specification is correct. This is supported by further verification work the Systems Assurance Group has performed. More than 80% of the verification conditions for Typhoon's autopilot and auto-throttle have been automatically proven. The remaining verification conditions have been subject to informal "hand proof"[3].

The levels of reliability for Typhoon are so great that the provision of independent formal verification can be justified. Unfortunately the evidence that the formal proof provides is qualitative in nature, it does not empirically demonstrate the quantitative software reliability targets – typically of the order of 10^{-9} faults per operational hour. Of course neither can any other development method scientifically demonstrate such a software reliability target before the software has been operating without change for many years.

The aim of producing software with zero coding errors that a program verifier could achieve is not scientifically demonstrable before the software is deployed. Evidence also suggests that when systems fail it is because of poorly understood requirements or mistaken assumptions in the specification, or design. This suggests that the application of Formal Methods is most powerful in the areas of requirements, specification and design. The Program Verifier challenge would seem to not address a significant problem, however I do not believe that this is the case.

Program code is the ultimate expression of requirements, specification and design. Although conventional software development can prove to be very effective, it has also proven to be expensive. The cost of conventional development and testing also increases considerably as the consequence of failure grows. Consider the following model of software development for a system in Figure 1 below.

---

[1] Simulink is a graphical language for control system specification with simulation and code generation tool support.

2 The coding error was revealed by QinetiQ's verification method, but turned up first during rig testing that had started 6 months beforehand.

3 The automated proof has been less successful because the specifications of these applications are not as mature as the specification for the flight control laws.

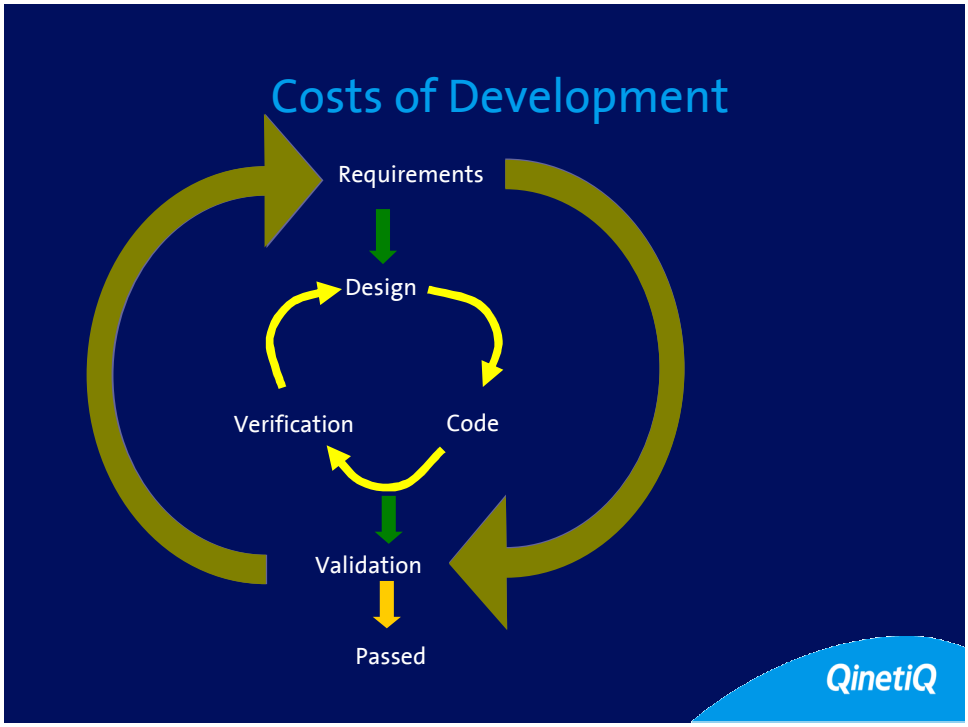

Figure 1

The model starts with the requirements, which might be captured by a model that can be simulated. The simulations would help explore the requirements space and lead to changes in the model, a different articulation of the requirements, or actual changes to the requirement in the light of insights gained. Moving around the loop in the model of the development is relatively inexpensive. At some point, after N iterations of the outer loop, a development will enter the design phase leading to the inner loop in the model shown in Figure 1. Code is developed from the design, verified by some means (testing, Fagan inspections, or whatever) and then probably corrected.

After M iterations of the inner loop of the development, a judgement will be made that no more is needed and it will move back to the outer loop. At this point requirements testing will take place. Requirements testing at this stage in the development can be a very expensive affair possibly employing continual operation of test rigs and test flights (in the case of aircraft). An error[4] found at this stage will mean that a transition will be made from the outer loop back into the inner loop to correct the identified error. Unfortunately more inner loop iterations may be necessary before moving back to the outer loop to repeat requirements testing. After P iterations of the outer loop a judgement will be made that it has passed and the product can be deployed. The cost of a residual error not detected within the inner loop is greatly magnified by the requirements testing in the final outer loop and it leads to further iteration of the inner loop, possibly introducing more errors.

Two observations arise from this discussion. The first observation is that the elimination of implementation errors would mean that any errors detected in the final outer loop must be requirements errors. If a Program Verifier were used, then only one iteration within the inner loop would be required; further there would be a significant reduction in costs due to the elimination of residual implementation errors that otherwise would be detected within the outer loop of requirements testing. The second observation is that the elimination of requirements errors in general has a larger impact than the elimination of implementation errors.

[4] whether it is due to requirements or has not been detected during verification in the inner loop

What does this mean for the Program Verifier grand challenge? First the use of formal techniques supported by tools is important for the reason given above. However how is a requirement error determined? Ultimately it is a subjective judgement (supported by a rational argument) that is made because real requirements are rooted within a human organisation. Even the requirements for an advanced aircraft like Typhoon are set by socio-economic and political conditions. Technical requirements based on capabilities derived from an operational analysis still require human judgement about for example acceptable loss rates or cost-benefit trade-offs. This means that people with special expertise will be central in judging requirements errors. Formal techniques and tools have an important role to make them more effective.

A third observation arises from the previous discussion, it is that nearly all requirements change due to the changes in the business/commercial, political or social environment. It is a significant problem for many of our systems today that soon after the system is developed and deployed it suffers from changes in the original environment it was developed for. This means that software development has to take evolution of the requirements into account during the system development, especially for complex systems.

The problem that the Program Verifier challenge addresses can be reduced to a mechanical procedure. When an error is detected then human intervention is still important in order to eliminate the class of errors detected, however it is systematic in nature and the rate of errors detected should diminish over time.

A key observation is that there is a trend towards the automation of code generation from simulation (and similar) models. The rise of automatic code generation is taking place for reasons quite separate from the Program Verifier grand challenge, but it could play a key role to the importance of the challenge. The grand challenge is then to provide Program Verifiers for commercial automated code generators from modelling languages into commercial languages. If source code generation and its formal verification can be automated then the problem of evolving requirements is mitigated if the cost of re-developing software and verifying it is significantly reduced. However the most important contribution of the Program Verifier challenge is that it could be scalable in a way that the elimination of requirements errors cannot be.

Consider the following simplistic, but useful, illustration. Suppose the elimination of requirement errors through the use formal techniques and human expertise resulted in a 50% saving in a development. Now consider the limited expertise available. For a number of development projects totalling £100, 000, 000, 000 (one hundred billion pounds), it would not be unreasonable to consider that only 1% could be addressed by the scarce human expertise available. This means that savings of 50% of one billion pounds could be achieved, a not inconsiderable sum of five hundred million pounds.

The scalability of the result of the Program Verifier challenge means that potentially all the projects totalling one hundred billion pounds could be addressed. Even a 10% saving would result in a saving of ten billion pounds. An experiment conducted by QinetiQ based on the Typhoon development indicates a saving of 30% over the system development lifecycle [1]. An important aspect of the experiment was to compare costs to achieve the same outcome, to pass the same set of acceptance tests. This means that the objective of software verification was not to achieve a failure rate better than that of the conventional development, rather it was to achieve at least the same quantifiable failure rate but at significantly less cost.

In order to achieve the potential benefits of the Program Verifier grand challenge a number of research issues must be addressed, the following constitute a minimum.

- A validated and rigorous measurement framework for comparing costs and benefits for development processes.

- The development of Program Verifiers for commercial language subsets with respect to commercially accepted modelling languages, this implies two supporting objectives:
    - language semantics for subsets of commercial languages (such as C and C++) that are mechanically checkable and sympathetic to program verification needs;
    - model language semantics that are mechanically checkable and sympathetic to program verification needs.
- Automated proof maintenance to support the evolution of the software.

Much work has already been done in these areas, but the products of these areas needs to be brought together and put into a coherent framework to support the Program Verifier challenge. A repository to collate results in this area would be an important step. The repository should also be used to make a judgement on the fitness for purpose of products in these areas. For example there are many semantic models for languages, there are rather fewer that are mechanically verifiable and directly support software verification. Many aspects of the judgement will be scientifically based, however some, such as usefulness to industry, will be partly subjective. The repository would also provide a means of conducting scientific experiments to measure the effectiveness (through testing) of a program verifier and compare its cost against historical data in a meaningful way [2] as was done for the Typhoon experiment [1].

Conclusions

The popular prejudice is that formal methods add cost to add reliability, I challenge this view. There is little evidence to show that a program verifier will make a scientifically quantifiable improvement in reliability before deployment of a system (the evidence that the use of a program verifier led to greater reliability 10 or 30 years later is not useful). Scientific experiments to determine the cost of achieving a quantifiable reliability target, implicit in an acceptance test criteria, can be conducted with an appropriate measurement framework. A programme of work to drive down the costs of verification and validation for commercial models and languages is required. The trend is towards a set of automatic code generators from commercial modelling languages that lend themselves to automated verification i.e. a set of program verifiers.

Systems are becoming more complex increasing the chances that errors will occur at higher levels, but in order to address this we need secure foundations. Software verification is important because it provides a solid foundation for Formal Methods to be applied in the design and requirements phase to gain further benefits.

The Systems Assurance Group has demonstrated the benefits of a program verifier in a narrow area for a particular modelling language. Requirements analysis of control law models by developing a Hoare logic for Simulink was based on the original verification work conducted by QinetiQ [3, 4].

QinetiQ is investigating the use of a mechanically validated semantics for an industrially relevant subset of C to specify a program verifier. Work to develop a mechanically checked semantics for an industrially relevant subset of C++ is due to start in August 2005. Formalisation of “The Mathworks” Stateflow modelling language for the purposes of software verification is nearing completion. An alternative semantic treatment for checking critical properties of Stateflow (and Statemate Statecharts) models is also underway. All this new work needs to be validated through use on real projects and is not the final answer. Richer semantic models to enable more

sophisticated implementations will be required. More modelling languages, such as versions of UML, need to be addressed. The development of cost modelling tools and comparison techniques grounded in a sound theory is required. All of these are worthy of being part of a Program Verifier grand challenge.

References

[1] N. Tudor, M. Adams, P. Clayton, C. O'Halloran, *Auto-coding/Auto-proving flight control software*, Digital Avionics Systems Conference, Salt Lake City, 2004.

[2] Clark, G. D., Caseley, P. R., Powell, A. L., Murdoch, J. , *Measurement of Safety Processes*. Incose 2003, Washington USA.

[3] R. J. Boulton, H. Gottliebsen, R. Hardy, T. Kelsy, and U. Martin. *Design Verification for Control Engineering*. In E. A. Boiten, J. Derrick, and G. Smith, editors, IFM 2004: Integrated Formal Methods, volume 2999 of LNCS, pages 21 – 35. Springer-Verlag, 2004. Invited paper.

[4] R. J. Boulton, R. Hardy, and U. Martin. 6th International Workshop on Hybrid Systems: Computation and Control. *A Hoare-Logic for Single-Input Single-Output Continuous-Time Control Systems*, volume 2623 of LNCS, pages 113 – 125. Springer-Verlag, 2003.