Practically Applicable Formal Methods

Peter Bernard Ladkin

University of Bielefeld and Causalis Limited

13 March 2013
The Task

The German standards committee GK914 for functional safety of E/E/PE systems has a SW ”working group”, AK 914.0.3, working hard on two issues at a rate of 6-7 meetings per year for the last three years. One of them is

- Practical, rigorous software assurance techniques
  - lumped together in IEC 61508:2010 as “formal methods” and treated as if they were one technique
Why?

Assessment over the SW industry performed by an experienced, well-known colleague, under the caveat

...non-scientific, non-reproduceable collection of data over the years. Some .. real firm”, i.e. project databases ..; some .. semi-firm, i.e. [indirect]; some .... anecdotal

gives the following figures

- commercial software: 10-30 flaws/kLoC
- good critical software 0.1-1 flaws/kLoC
- Praxis/Altran UK: 0.04 flaws/kLoC - a “bit of an outlier”.
Both Hatton (mainly C, 1995) and Altran (SPARK-Ada, Barnes 2003)

- use of a “language subset” with clear, unambiguous semantics
- use of mathematically rigorous program-analytical methods (“static analysis”)

Praxis/Altran:

- SPARK is a programming “environment” (tool) with built-in analytical methods

Hatton:

...there is currently little relationship between the quality of the software process and the intrinsic quality of the resulting code (Safer C, 1995)
Requirements Specification and Analysis

- Robyn Lutz, study of approx. 200 mission failures at NASA, early 1990’s
  - all but two: discrepancies between actual environment and requirements
- many observers: vast majority (over 70%) of SW failures are such “requirements anomalies”
- Praxis/Altran: rigorous (mathematical) specification and analytical techniques demonstrate their worth continually
Human Conflicts, Example 1

- From my experience 15 years ago
  - Complex, possibly perfect, specification language, along with checking/verification methods, for *algorithms*
  - “Methods” not described; I devised some, with hints from the developer
  - It worked!
    - Hand proof - weeks of work. Not commercially feasible
    - Automated proof-checking: Mark Saaltink had the Eves prover check the proof in less than one person-day, including debugging the proof
    - But I couldn’t have done it
    - Using a proof checker of this power remains a singular skill
    - I could see specialist shops doing it
  - This technique proved difficult to transmit to students
  - I was one of about a half-dozen people who could use it

That is not the kind of experience which translates into practical industrial methods.
Human Conflicts, Example 2

..... but it is the kind of thing which people focus on when the term “rigorous/formal/mathematical methods” is uttered.

- “Formal methods don’t work!” (reputed: B. Boehm, 1980’s)
- Some of us: “they do, you know!” (Sir Tony Hoare, Martyn Thomas, AdaCore, me, my pals, my cat)
  - “But we can’t learn them”
  - “We’ll develop some you can learn”
  - “It costs too much (people, time) for the benefit”
- Moral question: should we any longer be building systems which we don’t guarantee are fit for purpose?
Resolving Example 2

- SW is a mathematical object (Sir Tony)
- Claim: SW behavior can be assured fit for purpose in so far as the SW (behavior) can be -is- construed as a mathematical object and its relevant properties assured
- Some large companies doing this:
  - Microsoft: methods to evaluate third-party device drivers
  - Airbus: “high-level” state-machine-type SW development with code generation
Software Assurance

- Ruling out dangerous behavior caused by SW means being fairly convinced that the operation of the SW has certain properties which prevent such situations from arising inappropriately often.
- Assurance consists in part in establishing such properties.
Software Assurance Example

- there are general things that SW can do that you don’t want
- for example, run-time errors that cause a processor exception
- mathematical ("formal") methods are used for decades
- namely high-level languages supported by compilers and linkers
- more recently, incorporating intermittent assertions and correctness checks
  - Microsoft’s VCC
  - Ada 2012
How Good Are We?

.........after all these years? Let’s go back a decade and look

- Major military airplane, SW developed according to civil standards (DO-178B)
- SW developed according to DA Level A and B
- No significant quality difference found between Levels A and B
- “Module” quality generally very poor
  - Let me call the pieces of SW “modules”, not a technical term here
  - The worst had a defect rate of 1 in 10 lines of executable code (LOC)
  - The best had a defect rate of 1 in 250 LOC
  - Errors found are a litany of run-time-type problems, including some that should count as solved since the late 1960’s but apparently aren’t
Types of Errors 1

(With thanks to Martyn Thomas, Andy German, Dewi Daniels)
The following defects were among those reported in the software after certification:

* Erroneous signal de-activation.
* Data not sent or lost
* Inadequate defensive programming with respected to untrusted input data
* Warnings not sent
* Display of misleading data
* Stale values inconsistently treated
* Undefined array, local data and output parameters
Types of Errors 2

- Incorrect data message formats
- Ambiguous variable process update
- Incorrect initialisation of variables
- Inadequate RAM test
- Indefinite timeouts after test failure
- RAM corruption
- Timing issues - system runs backwards
- Process does not disengage when required
Types of Errors 3

- Switches not operated when required
- System does not close down after failure
- Safety check not conducted within a suitable time frame
- Use of exception handling and continuous resets
- Invalid aircraft transition states used
- Incorrect aircraft direction data
- Incorrect Magic numbers used
- Reliance on a single bit to prevent erroneous operation
What To Do

- Eliminate the possibility of run-time exceptions!
  - for example, by using SPARK
  - actually, if the client had insisted on using a strongly-typed language with adequate compiler, most of them could not have occurred

Are there other things we can do that are technically as mature?

Yes! Lots!
Two More Illustrations

- Complex (multimillion LOC) critical C&C software
  - data-integrity is an issue
  - there exists an algorithm to ensure the requisite data integrity
  - SW developed evolutionarily, so the system requirement was not addressed at system level: algorithm not used
    - lots of clever and effective part-checks for subsystems
  - with a formal system-level requirement, and assurance, that approach could not have been taken
- Small amount of control SW in a small, cheap box
  - sold in millions, sometimes in safety-relevant use
  - lots of functionality testing (requirements - top of the “V”)
  - code-level specifications derived through automatic generation from simulation “model” incorporating “HW in the loop” subsystems
  - hand-coded with heavy use of static analysis
  - the anomalies happen below the level of object code
Bringing in Architecture

- SW has four general life stages
  - Requirements development and specification
  - Design specification
  - “Source” code: more generally, the intermediate constructed object
  - The object code (linked)

- Apply the generic method to these four stages

- Hint: you pretty much have to use formal refinement
Assurance Via Architecture

So, for example, you need

- To assess requirements
- Compare design against requirements
- Compare source code against design
- Compare object code against source code
- Consider run-time monitoring

There are 26 industrially-mature steps and techniques which can be applied.
1. Rigorous functional requirements specification (FRS)
2. Formal FRS analysis (consistency; completeness)
3. Rigorous safety requirements specification (FSRS)
4. Formal FSRS analysis
5. Automated proving/proof checking of critical FRS, FSRS properties
6. Formal modelling FRS, FSRS, model checking, model exploration
7. Rigorous design specification (FDS)
8. Rigorous FDS analysis
9. Automated proving/proof checking that FDS fulfils FRS/ FSRS
10. Formal modelling, model checking, model exploration of FDS
11. Deterministic static analysis of FDS (information flow, data flow, possibilities of run-time error)
12. Codevelopment of FDS with Executable Source Code (ESC)
13. Automated source-code generation from FDS or intermediate specification (IS)
14. Automated proving/proof checking of fulfilment of FDS by IS
15. Automated verification-condition generation from/with ESC
16. Rigorous semantics of ESC
17. Automated ESCL-level proving /proof checking of properties (such as freedom from susceptibility to certain kinds of run-time error)
18. Automated proving/proof checking of fulfilment of FDS by ESC
19. Formal test generation from FRS
20. Formal test generation from FSRS
21. Formal test generation from FDS
26 Methods, Numbers 22-26

- 22. Formal test generation from IS
- 23. Formal test generation from ESC
- 24. Formal coding-standards analysis (SPARK, MISRA C, etc)
- 25. Worst-Case Execution Time (WCET) analysis
- 26. Monitor synthesis/runtime verification

These all in www.rvs.uni-bielefeld.de → Publications → What’s New → Functional Safety of Software-Based Critical Systems (PBL)

See also (same place) Hot Issues in Software Safety Standardisation (also PBL, slide set)
The Task

- Encourage and ensure widespread use of these state-of-the-practice mature methods
- State-of-the-practice is embodied in standards
- So that is where they belong
Thanks for listening!