Hot Issues in Software Safety Standardisation

Peter Bernard Ladkin

University of Bielefeld and Causalis Limited

16 October 2012

Three Issues

The German standards committee for functional safety of E/E/PE systems has a SW "working group". We have been working hard on two issues at a rate of 6-7 meetings per year for the last three years. There is another important general issue which has been raised through the IEC.

- Practical, rigorous software assurance techniques
  - lumped together in IEC 61508:2010 as "formal methods" and treated as if they were one technique
- How to assess previously-used SW?
  - You've got something. It has worked well forever in similar applications. Do you have to throw it away and develop again from scratch?
- The role of human factors in functional safety
  - Not software-specific. I won't address it in this talk.

Requirements and Guidance

- A helpful standard fulfils two roles
  - it requires: it says what people must
    - achieve ("goal-based")
    - do ("process-based")
  - it guides: it indicates how to do something if you don't already know how

References

- On assurance via practical rigorous methods in industrial use
  - PBL, Functional Safety of Software-Based Critical Systems, March 2011
  - *http://www.rvs.uni-bielefeld.de/publications/Papers/LadkinAdaConnection2011.pdf*
- On qualifying SW "black box" through previous successful use
  - DKE, Software elements for safety-related applications, September 2012
  - *http://www.vde.com/en/dke/DKEWork/NewsfromtheCommittees/2012/Pages/automation.aspx*

Progressive directions

- For the Uni BI link
  - Go to *www.rvs.uni-bielefeld.de*
  - From the links left, choose *Publications*
  - Scroll down a bit to *What's New*
  - Scroll down until you find the paper "Functional Safety...."
- For the DKE link
  - Go to *www.dke.de*
  - choose"English" (above right, small print)
  - Look under "News" (column under the banner)
  - Find "software elements for safety-related..."
  - click "more"

Human Requirements Conflicts, Example 1

- From my experience 15 years ago
  - Complex, possibly perfect, specification language and checking/ verification methods
  - "Methods" not described
  - I devised some, with large hints from the developer
  - It worked!
    - including automated proof-checking: Mark Saaltink had the Eves prover check a complex proof in less than one person-day, including debugging the proof
    - but I couldn't have done it
    - using a proof checker remains a singular skill
    - I could see specialist shops doing it
  - It proved difficult to transmit to students
  - I was one of about a half-dozen people who could use it
  - And even I probably can't, any more

Human Requirements Conflicts, Example 2

- "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?
- Business/social question: why does it (still) "cost too much for the benefit"?

Aside: Resolving Example 2

- SW is a mathematical object (Sir Tony)
- But this doesn't always help evaluation (many C compilers)
- Claim: SW behavior can be assured fit-for-purpose in so far as the SW (behavior) can be -is- construed as a mathematical object
- Moral consequence: SW should be written in such a way as to enable its evaluation as a mathematical object
- Some large companies are heavy users of formal methods in this sense
  - Microsoft: methods to evaluate third-party device drivers
  - Airbus: "high-level" state-machine-type SW development with code generation

Appropriately "Safe" Software

- Software whose operation is appropriately free of dangerous behavior
- Two observations –
  - define "appropriate"
  - software execution by itself is not "dangerous"

Software Involvement in Safety

- software behavior causes behavior of kit (hardware)
- this kit behavior might be dangerous
- it follows that dangerous behavior of SW is *derivative*
- the requirements on non-dangerous behavior of SW are derivative
- but that is only part of the story
  - Daniel Jackson claims, on the basis of the above, that all specific safety analysis may be done *before* the software requirements are given to the designer
  - But some appropriate safety-analysis methods don't respect this division of labor (OHA, for example)

Software Behavior

- if it is possible for SW to engage in behavior categorised derivatively as "dangerous"
- then it is a priori conceivable that this behavior might be exhibited at any time
- because "*go to*" is *a priori* possible
- and the data to be operated on/with might be unfortunately apt

Software Assurance

- so 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
- Please note: you establish properties by testing and assuring them directly
- how do you do that? Answer: for many useful properties, it is done – somewhere on this earth

General Software Assurance

- there are general things that SW can do that you don't want
- for example, run-time errors that cause a processor exception
- then there are specific things, derived from the safety requirements of the kit operated by the SW
- I think it is practically worthwhile to treat these two aspects differently
    - dependability1
    - dependability2

Two other aspects

- Dependable SW *does what we want it to do*
- But in safety we also need that
  it *doesn't do what we don't want it to do*
  - How do we know what we don't want?
    - HazAn
  - How do we assure ourselves that we know?
    - Assured properties of HazAn
  - How do we tell that the SW doesn't do any of that?
    - assured properties of the SW
- As Dijkstra pointed out, it is very hard to establish a negative
- This isn't the whole story: we must also consider fail-operational systems, such as building emergency functions.

Dependability1

- formal methods are used here for decades
- namely high-level languages supported by compilers and linkers
- Example: my C compiler *does what I want it to do*
  - Do I know what I want, exactly? Yes.
  - How do we tell the SW does it? Ummm. (BMW's experience was recounted at SAFECOMP 2007)
- But it also *doesn't do what we don't want it to do*
  - How do we know what we don't want? Trickier.
  - How do we assure ourselves that we know? I don't think we do
  - How do we tell that the SW doesn't do any of that? Ten years ago, it did. 80% of internet exploits were buffer overflow exploits
  - We have recently got a lot better (Microsoft's VCC)

How Good Are We?

........after all these years? Let's go back a decade and more

- 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

How Good We Are, cont'd

- One airplane: 1 in 250 LOC or worse
- Rumored industry standard for safety-critical SW: // 1 in 1000 LOC to 1 in 10,000 LOC
- Best documented quality: 1 in 25,000 LOC (guess who!)

I say we aren't very good. Still.

This is a public example. I have *recent* private examples.

What To Do About It

- It must be a people problem
  - There is no other explanation for why mistakes are still being made whose technical solution has been known for four decades
- People problems are notoriously intractable
- Address the memes and mantras

Memes and Mantras

- Actually, I prefer the term "trope"
- A meme is an idea that promulgates (Dawkins and Dennett)
- A mantra is a short statement or belief (see below)
- A trope is a mantra with reasoning
  - "Formal methods don't work". Depends.
    - Dependability1: eliminating run-time errors is practical
    - Dependability2: Rigorous verification that the design spec fulfils the requirements spec is mostly impractical
  - "Programming language Q is as good as programming language S if you take care".
    - Dependability1 issue
    - Taking care didn't help with all those errors in the aircraft SW.
    - If the client had insisted on using a strongly-typed language with adequate compiler, most of them could not have occurred

Memes and Mantras, continued

- " Our SW has been proven reliable in use"
  - Can you show us the statistics?
    - Mostly not. Or very poor. Incomplete, etc.
  - How do we know you have reliably detected all erroneous behavior?
    - Mostly, we don't
    - Because you likely haven't
    - Only those failures were noted which happened to result in something noteworthy
  - Is your statistical reasoning valid?
    - Most people don't distinguish assertion from confidence
    - they are inverse in strength on the same data
    - rigorous testing:

      high confidence that reliable to one failure in 10,000 ophours

      very low confidence .... in 1,000,000 ophours
    - no guidance in IEC 61508:2010 as to acceptable confidence ranges
  - Involves both Dependability1 and Dependability2

What Not To Do About It

- Write a generic software safety standard that is 50pp long
  - That is in part incoherent - experts disagree on whether safety requirements traceability to SW is assured
  - Based on a set of concept definitions that are sophomoric (if you happen to study a subject for which analysing definitions is essential)
- Then, thirteen years later, extend it to 110pp!
  - Not my fault! I came in later
- Some prominent SW specialists think the standards process is broken
- How can the standard be fixed?
  - Shorten it
  - Make it readable and coherent
  - Address dependability1 issues
  - Address dependability2 issues as far as possible
  - Enumerate the state of the art where one exists

A Generic Safety Standard

- Determine what we don't want the system to do (Accidents)
  - As completely as possible
  - Provide the assurance that you have everything
- Determine how it could happen (Hazards)
  - As completely as possible
  - If you go too far, that's OK
  - Provide the assurance of coverage (completeness)
- "Apportion" the hazard behavior to the system components (including SW)
  - Show the apportionment covers the hazards
- For the SW, indeed any component: repeat the above

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*

Bringing in 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.

26 Methods

- 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. Determininistic static analysis of FDS
  (information flow, data flow, possibilities of run-time error)

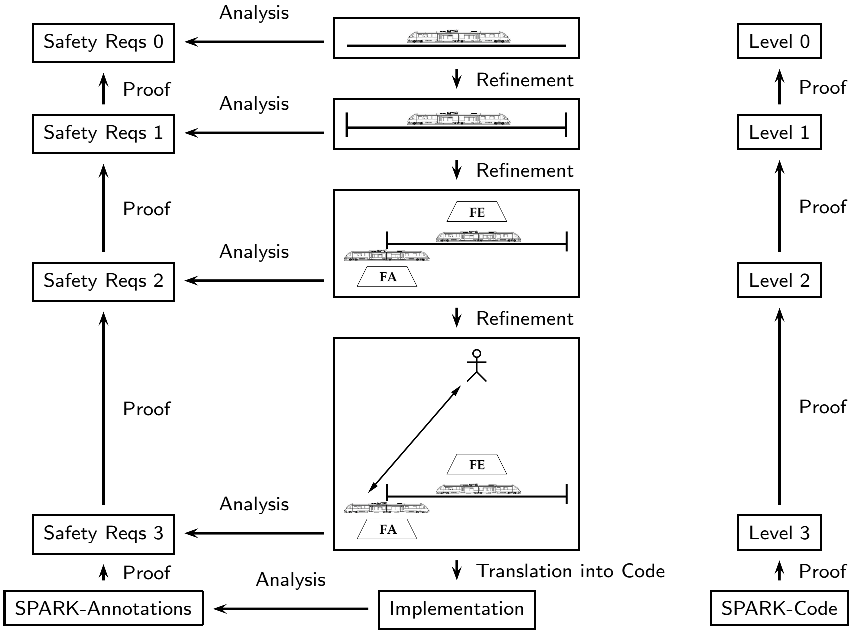
26 Methods, continued

- 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
- 22. Formal test generation from IS
- 23. Formal test generation from ESC

26 Methods, continued

- 24. Formal coding-standards analysis (SPARK, MISRA C, etc)
- 25. Worst-Case Execution Time (WCET) analysis
- 26. Monitor synthesis/runtime verification

Example — Ontological Hazard Analysis



“Black Box” Assurance

- Using SW which has been successfully used before
- No access to the logic of the code
  - you must interface to other proprietary SW
  - you want to use the interface that has been successfully used before
  - the manufacturer gives you the interface specification
- The raw statistics are daunting
  - At the 95% confidence level, 3 billion hours of fault-free operation

More on the Statistical Inference

- At the 95% confidence level, 3 billion hours of fault-free operation
- At the 99% level, 4.6 billion hours
- comes from the exponential model of SW faults
- "the **exact** operational profile... must be used, and used **exactly**" (Bev Littlewood) for valid inference from previous to future use
- that involves (IEC 61508:2010) "*clearly restricted and specified functionality*" and operational conditions "*sufficiently close to*" those of previous use
- it involves rather more than that! (German committee)

Statistical Inference, continued

- It involves (redaction of German committee observations)
  - all future combinations of input data having been proven-in-use for long enough
  - all future sequences of function calls .....
  - temporal relations of those sequences ....
  - identical environment, configuration, SW interfaces, libraries, OS and compiler
  - plus written assurance: complete description of conditions of use, rigorous specification, etc
- this seems to be a hugely high hurdle
- very few companies, if any, have this level of detail about their previous SW operations
- the big, unanswered, question: What to do?

Finis

- Though sketchy, this talk is long enough
- The technical material is or will be available on the WWW
- The important thing is that people who are interested and concerned know it is there, and how to get to it
- ...... and are motivated to engage in the process of improving matters

Thanks for listening!