Lecture at Ames Iowa, 7 Mar 2013, based on Dagstuhl talk in January 2103, plus Cambridge University talk in 2011, and “Distinguished Lecture” at Institute of Information Sciences, Academia Sinica, 14 Feb 2011, which was based on LAW 2010 invited talk 7 December 2010

The Challenge of High-Assurance Software

John Rushby
Based on joint work with Bev Littlewood (City University UK)

Computer Science Laboratory
SRI International
Menlo Park CA USA

The Basic Challenge: Systems Level

- Some systems must not fail or go wrong
  - Nuclear power, chemical plants, flight control
  - Phone system, air traffic control
  - Pacemakers, automobile braking and steering
- So, think of everything that could go wrong
  - Those are the hazards

  Design them out, find ways to mitigate them
  - i.e., reduce consequences, frequency

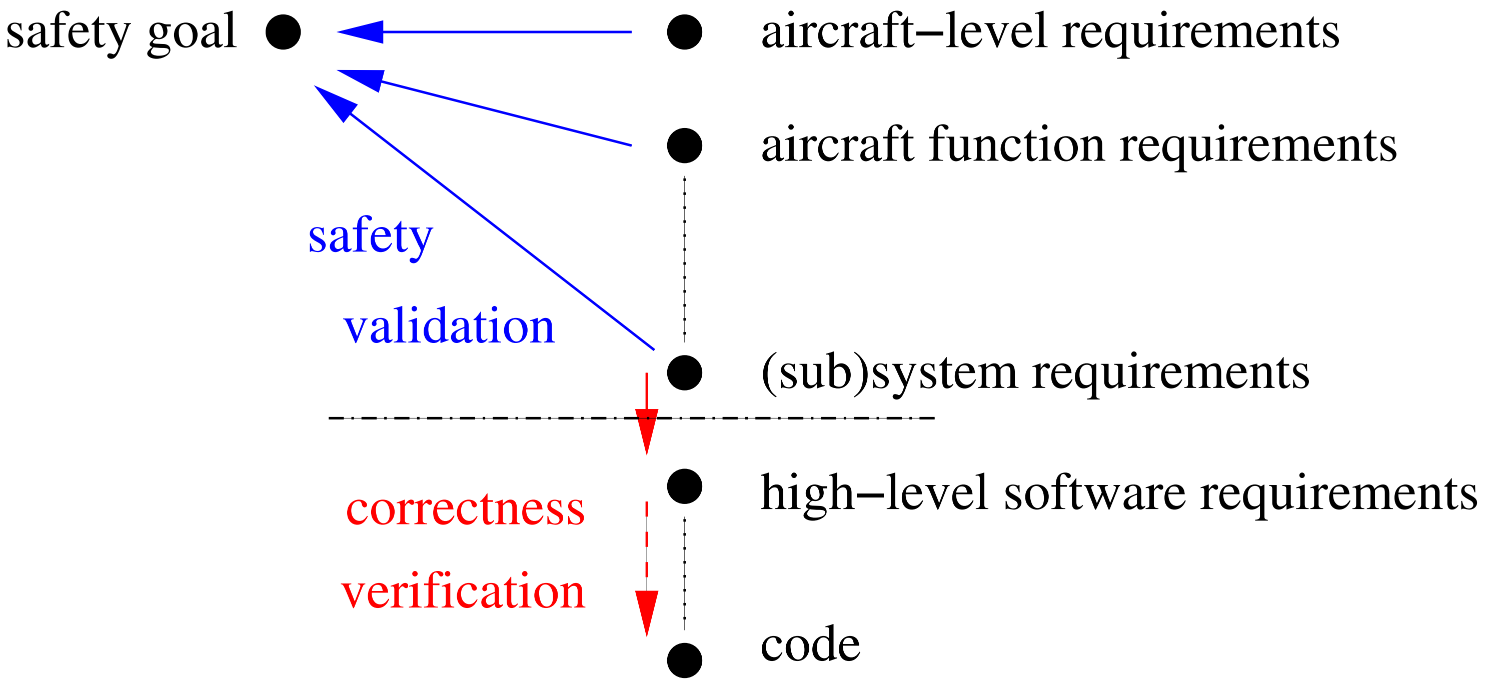
  This may add complexity (a source of hazards)
- Iterate
- And then recurse down through subsystems
- Until you get to widgets
  - Build those correctly
- Provide assurance that you have done all this successfully

The Basic Challenge: Software

- Software is a widget in this scheme
- We don't analyze it for safety, we build it correctly
- In more detail. . .
  - Systems development yields functional and safety requirements on a subsystem that will be implemented in software
    - ⋆ Often expressed as constraints or goals
  - From these, develop the high level software requirements
    - ⋆ How to achieve those goals
  - Elaborate through more detailed levels of requirements
  - Until you get to code (or something that generates code)
- Provide assurance that you have done all this successfully

Aside: Software is a Mighty Big Widget

The example of aircraft


safety goal
aircraft–level requirements
aircraft function requirements
safety
validation
(sub)system requirements
correctness
verification
high–level software requirements
code

- As more of the system design goes into software
- Maybe the widget boundary should move
- Safety vs. correctness analysis would move with it

Software Safety Assurance: 4+1 Principles (Tim Kelly)

1. Safety elements of high level software requirements must address software contribution to system hazards

2. Intent of these software safety requirements is maintained throughout development

3. New hazards are identified and mitigated

4. Running code must satisfy software safety requirements

Must address the 4 principles in ways that establish confidence commensurate to contribution of software to system risk

Software Safety Assurance: Aircraft Case

DO-178B/C guidelines

1. Safety elements of high level software requirements must address software contribution to system hazards

   This is done at the systems level (ARP 4761, 4754A)

2. Intent of these software safety requirements is maintained

   Lots of requirements documentation, analysis, traceability

3. New hazards are identified and mitigated

   Derived requirements are thrown to the systems level

4. Running code must satisfy software safety requirements

   More documentation, analysis, traceability, testing

Must address the 4 principles in ways that establish confidence commensurate to contribution of software to system risk

Which brings us to. . .

The Conundrum

- Cannot eliminate hazards with certainty (because the environment is uncertain), so top-level claims about the system are stated quantitatively
  - E.g., no catastrophic failure in the lifetime of all airplanes of one type (“in the life of the fleet”)
- And these lead to probabilistic systems-level requirements for software-intensive subsystems
  - E.g., probability of failure in flight control $< 10^{-9}$ per hour
- To assure this, do lots of software assurance
- But this is all about showing correctness
- For stronger subsystem claims, do more software assurance
- How does amount of correctness-based software assurance relate to probability of failure?

The Example of Aircraft

- Aircraft failure conditions are classified in terms of the severity of their consequences
- Catastrophic failure conditions are those that could prevent continued safe flight and landing
- And so on through severe major, major, minor, to no effect
- Severity and probability/frequency must be inversely related
- AC 25.1309: No catastrophic failure conditions in the operational life of all aircraft of one type
- Arithmetic and regulation require the probability of catastrophic failure conditions to be less than 10^{-9} per hour, sustained for many hours
- And 10^{-7}, 10^{-5}, 10^{-3} for the lesser failure conditions

The Example of Aircraft (ctd.)

- DO-178B/C identifies five Design Assurance Levels (DALs)

- And 66 assurance objectives
  - E.g., documentation of requirements, analysis, traceability from requirements to code, test coverage, etc.

- More objectives (plus independence) at higher DALs
  - 28 objectives at DO178B Level D (10^{-3})
  - 57 objectives at DO178B Level C (10^{-5})
  - 65 objectives at DO178B Level B (10^{-7})
  - 66 objectives at DO178B Level A (10^{-9})

- The Conundrum: how does doing more correctness-based objectives relate to lower probability of failure?

Some Background and Terminology

Aleatory and Epistemic Uncertainty

- Aleatory or irreducible uncertainty
  - is “uncertainty in the world”
  - e.g., if I have a coin with $P(heads) = p_h$, I cannot predict exactly how many heads will occur in 100 trials because of randomness in the world

  Frequentist interpretation of probability needed here

- Epistemic or reducible uncertainty
  - is “uncertainty about the world”
  - e.g., if I give you the coin, you will not know p_h; you can estimate it, and can try to improve your estimate by doing experiments, learning something about its manufacture, the historical record of similar coins etc.

  Frequentist and subjective interpretations OK here

Aleatory and Epistemic Uncertainty in Models

- In much scientific modeling, the aleatory uncertainty is captured conditionally in a model with parameters

- And the epistemic uncertainty centers upon the values of these parameters

- As in the coin tossing example: p_h is the parameter

Software Reliability

- Software contributes to system failures through faults in its requirements, design, implementation—bugs
- A bug that leads to failure is certain to do so whenever it is encountered in similar circumstances
  - There's nothing probabilistic about it
- Aaah, but the circumstances of the system are a stochastic process
- So there is a probability of encountering the circumstances that activate the bug
- Hence, probabilistic statements about software reliability or failure are perfectly reasonable
- Typically speak of probability of failure on demand (pfd), or failure rate (per hour, say)

Testing and Software Reliability

- The basic way to determine the reliability of given software is by experiment
  - Statistically valid random testing
  - Tests must reproduce the operational profile
  - Requires a lot of tests

- Feasible to get to pfd around 10^{-3}, but not much further
  - 10^{-9} would require 114,000 years on test

- Note that the testing in DO-178B is not of this kind
  - it's coverage-based unit testing: a local correctness check

- So how can we estimate reliability for software?

Back To The Main Thread

Aleatory and Epistemic Uncertainty for Software

- We do correctness-based software assurance
- And do more of it when higher reliability is required
- But the amount of correctness-based software assurance has no obvious relation to reliability
- Maybe it relates better to some other probabilistic property of the software's behavior
- We are interested in a property of its dynamic behavior
  - There is aleatoric uncertainty in this property due to variability in the circumstances of the software's operation
- Assurance examines static attributes of the software to form an epistemic estimate of the dynamic property
  - More examination refines the estimate
- For what kinds of properties could this work?

Perfect Software

- Property cannot be about some executions of the software
  - Like how many fail
  - Because the epistemic examination is static (i.e., global)
  - This is the disconnect with reliability
- Must be a property about all executions, like correctness
- But correctness is relative to software requirements, which themselves may be flawed
  - Actually, the main source of failure in aircraft software
- We want correctness relative to the critical claims in the subsystem requirements
- Call that perfection
- Software that will never experience a critical failure in operation, no matter how much operational exposure it has

Possibly Perfect Software

- You might not believe a given piece of software is perfect
- But you might concede it has a possibility of being perfect
- And the more assurance it has had, the greater that possibility
- So we can speak of a (subjective) probability of perfection
- For a frequentist interpretation: think of all the software that might have been developed by comparable engineering processes to solve the same design problem
    - And that has had the same degree of assurance
    - The probability of perfection is then the probability that any software randomly selected from this class is perfect

Probabilities of Perfection and Failure

- Probability of perfection relates to software assurance
- But it also relates to reliability:

  By the formula for total probability

$$\begin{aligned} & P(\text{s/w fails [on a randomly selected demand]}) \qquad (1) \\ = \; & P(\text{s/w fails} \,|\, \text{s/w perfect}) \times P(\text{s/w perfect}) \\ & + P(\text{s/w fails} \,|\, \text{s/w imperfect}) \times P(\text{s/w imperfect}). \end{aligned}$$

- The first term in this sum is zero, because the software does not fail if it is perfect (other properties won't do)
- Hence, define
  - p_{np} probability the software is imperfect
  - p_{fnp} probability that it fails, if it is imperfect
- Then $P(\text{software fails}) = p_{fnp} \times p_{np}$
- This analysis is aleatoric, with parameters p_{fnp} and p_{np}

Epistemic Estimation

- To apply this result, we need to assess values for p_{fnp} and p_{np}
- These are most likely subjective probabilities
  - i.e., degrees of belief
- Beliefs about p_{fnp} and p_{np} may not be independent
- So will be represented by some joint distribution $F(p_{fnp}, p_{np})$
- Probability of software failure will be given by the Riemann-Stieltjes integral

$$\int\limits_{\substack{0 \le p_{fnp} \le 1 \\ 0 \le p_{np} \le 1}} p_{fnp} \times p_{np} \, dF(p_{fnp}, p_{np}). \qquad (2)$$

- If beliefs can be separated F factorizes as $F(p_{fnp}) \times F(p_{np})$
- And (2) becomes $P_{fnp} \times P_{np}$

  Where these are the means of the posterior distributions representing the assessor's beliefs about the two parameters

Practical Application—Nuclear

- Traditionally, nuclear protection systems are assured by statistically valid random testing
- Very expensive to get to pfd of 10^{-4} this way
- Our analysis says $pfd \leq P_{fnp} \times P_{np}$
- They are essentially setting P_{np} to 1 and doing the work to assess $P_{fnp} < 10^{-4}$
  - Conservative assumption that allows separation of beliefs
- Any software assurance process that could give them $P_{np} < 1$

  Would reduce the amount of testing they need to do
  - e.g., $P_{np} < 10^{-1}$, which seems very plausible
  - Would deliver the the same pfd with $P_{fnp} < 10^{-3}$
- This could reduce the total cost of certification
  - Provided beliefs still factorize

Practical Application—Aircraft, Version 1

- No plane crashes due to software, and enough operational exposure to validate software failure rate $< 10^{-9}$
- Aircraft software is assured by processes such as DO-178B Level A
- They also do a massive amount of all-up testing but do not take assurance credit for this
- Our analysis says software failure rate $\leq P_{fnp} \times P_{np}$
- So they are setting $P_{fnp} = 1$ and $P_{np} < 10^{-9}$
- Littlewood and Povyakalo show (under independence assumption) that large number of failure-free runs shifts assessment from imperfect but reliable toward perfect
- So flight software might indeed have probabilities of imperfection $< 10^{-9}$
- And DO-178B delivers this

Practical Application—Aircraft, Version 2

- Although no crashes due to software, there have been several incidents
- So actual failure rate may be only around 10^{-7}
- Although they don't take credit for all the testing they do, this may be where a lot of the assurance is really coming from
- Our analysis says software failure rate $\leq P_{fnp} \times P_{np}$
- So perhaps testing is implicitly delivering, say, $P_{fnp} < 10^{-3}$
- And DO-178B is delivering only $P_{np} < 10^{-4}$
- I do not know which of Version 1 or 2 is true
- Furthermore, these are *a posteriori* analyses, we need *a priori*
- So what's to be done? . . . hold that question

Practical Application: Two Channel Systems

- Many safety-critical systems have two (or more) diverse "channels" arranged in 1-out-of-2 (1oo2) structure
  - E.g., nuclear shutdown
- A primary protection system is responsible for plant safety
- A simpler secondary channel provides a backup
- Cannot simply multiply the pfds of the two channels to get pfd for the system
  - Failures are unlikely to be independent
  - E.g., failure of one channel suggests this is a difficult case, so failure of the other is more likely
  - Infeasible to measure amount of dependence

  So, traditionally, difficult to assess the reliability delivered

Two Channel Systems and Possible Perfection

- But if the second channel is simple enough to support a plausible claim of possible perfection, then
  - Its imperfection is conditionally independent of failures in the first channel at the aleatory level
  - Hence, system pfd is conservatively bounded by product of pfd of first channel and probability of imperfection of the second
  - $P(\text{system fails on randomly selected demand} \leq pfd_A \times pnp_B$

  This is a theorem

- Epistemic assessment similar to previous case
  - But may be more difficult to separate beliefs
  - Conservative approximations are available

Type 1 and Type 2 Failures in 1oo2 Systems

- So far, considered only failures of omission
  - Type 1 failure: both channels fail to respond to a demand
- Must also consider failures of commission
  - Type 2 failure: either channel responds to a nondemand
- Demands are events at a point in time; nondemands are absence of demands over an interval of time
- So full model must unify these
- Details straightforward but lengthy

Monitored Architectures

- A variant on 1oo2
- One operational channel does the business
- Simpler monitor channel can shut it down if things look bad
- Used in airplanes, avoids malfunction and unintended function
    - Higher level redundancy copes with loss of function
- Analysis is a variant of 1oo2:
    - No Type 2 failures for operational channel
- Monitored architecture risk per unit time

  $\leq c_1 \times (M_1 + F_A \times P_{B1}) + c_2 \times (M_2 + F_{B2|np} \times P_{B2})$

  where the Ms are due to mechanism shared between channels
- May provide justification for some of the architectures suggested in ARP 4754
    - e.g., 10^{-9} system made of Level C operational channel and Level A monitor

Monitors Do Fail

- Fuel emergency on Airbus A340-642, G-VATL, 8 February 2005
  - Type 1 failure
- EFIS Reboot during spin recovery on Airbus A300 (American Airlines Flight 903), 12 May 1997
  - Type 2 failure
- These weren't very good monitors
- So what's to be done? . . . hold that question

Diagnosis and Prescriptions

- First of all we need a framework for talking about the whole process of assurance
- Idea of an assurance case provides this
  - Claims
  - Argument
  - Evidence
  - The argument justifies the claims, based on the evidence
- Some fields require assurance or safety case for certification
  - e.g., FDA requires them for Infusion pumps
- Others use standards and guidelines such as DO-178C
  - The claims are largely established by regulation, guidelines specify the evidence to be produced, and the argument was presumably hashed out in the committee meetings that produced the guidelines
  - In the absence of a documented argument, it's not clear what some of the evidence is for: e.g., MC/DC testing

Safety Cases and Formal Verification

- The argument justifies the claims, based on the evidence
- This is a bit like logic (cf. "argumentation")
    - A proof justifies a conclusion, based on given assumptions and axioms
- So what's the difference between an assurance case and a formal verification?
- Aha! An assurance case also closely examines the interpretation of the formalized conclusion and why we should believe the assumptions and axioms
    - e.g., contemplate my formal verif'n in PVS of Anselm's Ontological Argument (for the existence of God)
- So where do the assumptions and conclusions come from in a formal verification?

Logic And The Real World

- Formal verification is calculation in logic
  - It's difficult because calculations in logic are all NP-Hard
  - But benefits are the same as those for calculation in other engineering fields (can consider all cases)
- Software is logic
- But it interacts with the world
  - Actual semantics of its implementation
  - Sensors, actuators, devices, the environment, people, other systems
- For formal verification we describe these by models, in logic

Logic and Epistemology in Assurance Cases

- We have just two sources of doubt in an assurance case

- Logic doubt: the validity of the argument
  - Can be eliminated by formal verification
  - Subject to caveats on soundness of tools
  - This is Leibniz' Dream: "let us calculate"

- Epistemic doubt: the accuracy and completeness of our knowledge of the world in its interaction with the system
  - As expressed in our models and requirements
  - This is where we need to focus

- Same distinction underlies Verification and Validation (V&V)

Aside: Resilience

- It is often possible to trade epistemic and logic doubts
  - Weaker assumptions, fewer epistemic doubts
  - *vs.* more complex implementations, more logic doubt
- For example, highly specific fault assumptions, vs. Byzantine fault tolerance
- I claim resilience is about favoring weaker assumptions
- Good for security also: the bad guys attack your assumptions
- Formal verification lets us cope with the added logic doubt
  - cf. FAA disallows adaptive control due to logic doubt

Reducing Epistemic Doubt: Validity

- We have a model and we want to know if it is valid
- One way is to run experiments against it
- That's why simulation models are popular
  - To be executable, have to include a lot of detail
  - But detail is not necessarily a good thing in a model
  - Epistemic doubt whether real world matches all that detail
- Instead we should favor descriptions in terms of constraints
  - Our task is to describe the world, not to implement it
  - Less is more!
- Calculation on constraint-based models is now feasible
  - Recent advances in fully automated verification
  - Infinite bounded model checking (Inf-BMC), enabled by solvers for satisfiability modulo theories (SMT)

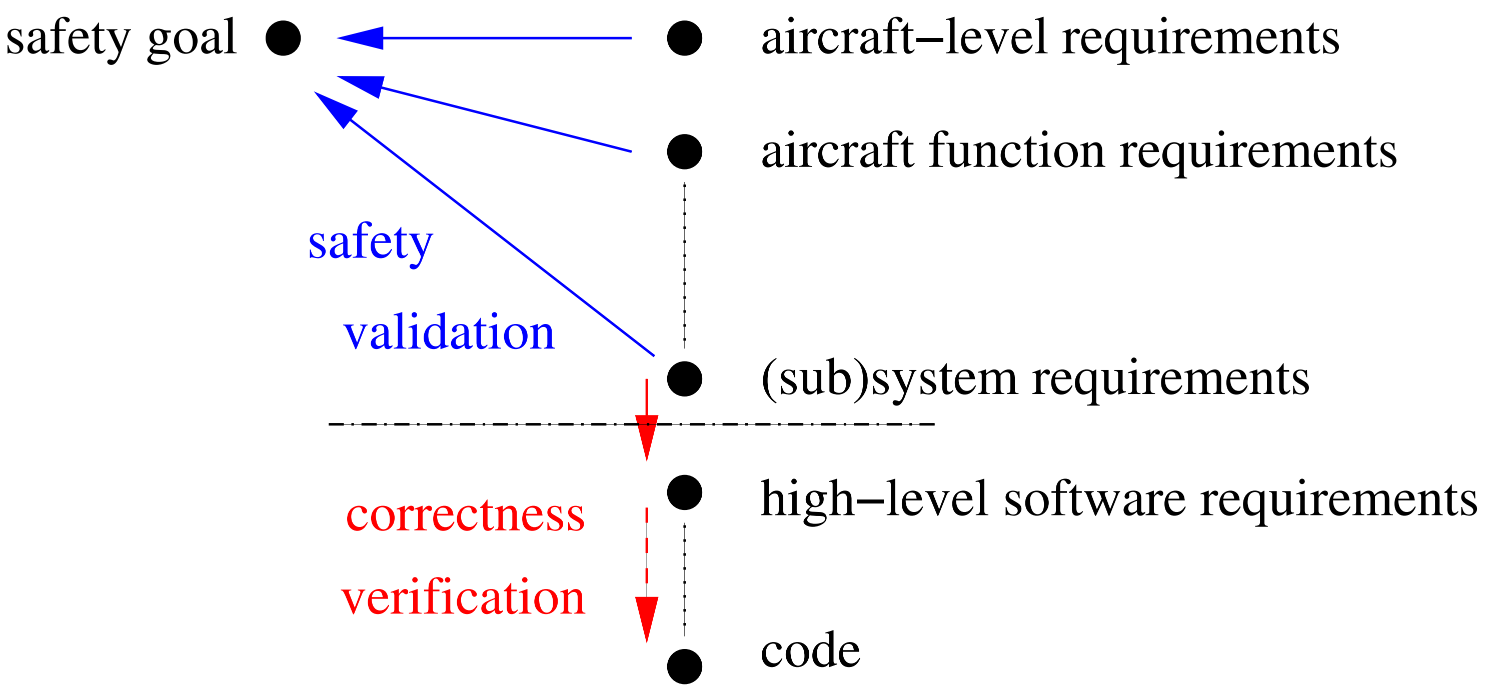
Reducing Epistemic Doubt: Validity (ctd. 1)

- All aircraft incidents due to software had their root cause in flawed requirements
  - Either the system level requirements were wrong
  - Or the high level software requirements did not correctly reproduce their intent
- None were due to implementation defects
  - Might not be so in other application areas
- One problem is that descriptions at the system level are (rightly) very abstract
  - Typically box and arrow pictures, supplemented with math
  - Little support for automated exploration and analysis
- And these descriptions are getting more complex, because there are more cases to deal with (i.e., more like software)

Reducing Epistemic Doubt: Validity (ctd. 2)

- Traditional ways to explore system-level models, such as failure modes and effects analysis (FMEA) and fault tree analysis (FTA) can be seen as manual ways to do incomplete state exploration with some heuristic focus that directs attention to the paths most likely to be informative
- Increasingly many cases to consider, like software. so it makes sense to apply methods from software to the specification and analysis of these designs
- But must keep things abstract
- Aha! Inf-BMC can do this
- Inf-BMC allows use of uninterpreted functions, e.g., $f(x)$
- Constraints can be encoded as synchronous observers
- With comparable models Inf-BMC can do automated model checking and cover the entire modeled space

Recall This Picture


safety goal
aircraft–level requirements
aircraft function requirements
safety
validation
(sub)system requirements
correctness
verification
high–level software requirements
code

- As more of the system design goes into software
- Software analysis methods should be applied to system

Reducing Epistemic Doubt: Completeness

- Quintessential completeness problem is hazard analysis
  - Have I thought of all the things that could go wrong?
- There are systematic techniques that help suggest possible hazards: FMEA, HAZOP etc.
  - These can be partially automated
  - cf. notion in Epistemology that knowledge is belief justified by a generally reliable method
- But there seems no way to prove we do have all the hazards
- So surely need some measure of our confidence that we do
- Same for all the other reasons (called defeaters) why our safety argument might be flawed

Eliminative Induction, Baconian Probability

- Some take inspiration from scientific method
- Many candidate theories, design experiments to test them, eliminate those shown to be wrong (Francis Bacon, roughly)
- “Once you eliminate the impossible, whatever remains, no matter how improbable, must be the truth” (Holmes)
- Substitute defeaters for theories
  - Have many reasons why safety argument could be flawed, eliminate them one by one
- Baconian Probability is a measure for this: number eliminated ÷ number considered
- More complex form advocated in Philosophy of Law (Cohen)
  - “Beyond reasonable doubt,” “balance of probabilities”
- Doesn’t behave like a probability
- Easily gamed (invent ridiculous defeaters)

Bayesian Induction

- An intellectually justifiable method should allow us to quantify
  - Confidence that we have identified all hazards
  - Confidence that we have eliminated or mitigated any given hazard
  - A way to apportion effort: confidence required in the elimination of any given hazard should depend on the risk (i.e., likelihood and consequence) that it poses
- Surely the right way to do this is to use genuine probabilities
  - Subjective prior probabilities updated (via Bayes rule) as evidence becomes available
- "Bayesian Induction is Eliminative Induction" (Hawthorne)
- Making this practical would be a significant research agenda

Conclusion

- Probability of perfection is a radical and valuable idea
  - It's due to Bev Littlewood, and Lorenzo Strigini
- Provides the bridge between correctness-based verification activities and probabilistic claims needed at the system level
- Explains what software assurance is
- Relieves formal verification, and its tools, of the burden of infallibility
- Explains the merit of monitors
- Distinguishing logic and epistemic doubts allows different methods to be focused on each
- Possibly explains resilience
- Suggests approaches for reducing epistemic doubts
- And for quantifying confidence in total case

Proposals: Practical and Speculative

- Use monitors formally verified or synthesized against the system safety requirements

- Use formal methods in analysis of system-level designs and requirements

- Develop a priori estimates of probability of perfection based on assurance performed

- Combine testing and correctness-based software assurance in estimating reliability

- Develop an intellectually justifiable approach to certification

- But note that none of this is compositional: fix that!