Lectures for Marktoberdorf 2010 Summer School

Software and Systems Safety: Specification and Verification

Formal Methods
And Argument-based Safety Cases

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Introduction

Safety and Hazards

- Suppose a village wants to build a soccer field

- But the only space available is at the edge of a cliff

- Clearly, there's a hazard

Dealing With Hazards

- One way to mitigate the hazard is to build a fence at the edge of the cliff
- But then we must look at technical properties of the fence
    - How strong is it?
    - Can it resist collision by n players of weight w at speed s?
    - What are reasonable values of these parameters?
    - And what does resist mean? ... always, probably?
- The fence itself could introduce new hazards
    - Particularly if we don't communicate well to suppliers
    - e.g., suppose it's made of barbed wire
- Perhaps we should eliminate the hazard
    - e.g., by bulldozing the cliff

Safety

- As soon as we postulate the existence of a system, there are likely to be some events or behaviors that are undesirable
- We first sharpen the notion of undesirable

  safety: loss of life, injury, environmental harm,. . .

  security: disclosure of sensitive information, unauthorized modification, . . .

  others: loss of money, goods, reputation
- These critical issues are much more important to larger society than the function of the system (i.e., people falling to their death vs. having a good game of soccer)
  - Though sometimes the two coincide: e.g., failure of the London computer-aided ambulance dispatch system
- Certification, and hence design and assurance, focus on the critical issues

Hazards (again)

- Once we know the undesirable events/behaviors
- We can systematically search for hazards that could bring those undesirable events about
  - This is hazard analysis
- Then we design our system to eliminate or mitigate hazards
  - Mitigate means to lessen their seriousness or frequency
  - So we need metrics on seriousness
  - And frequencies or probabilities of occurrence
- Our design decisions may introduce new hazards
  - Recall discussion of A320 crash in Warsaw
    (LH 2094, 14 Sept 1993)
- So the process iterates

Risk

- Risk is product of severity of bad outcome and its frequency
- Usually want inverse rl'nship between severity and frequency
- e.g., FAA AC 25.1309-1A for civil aircraft: failure conditions

  catastrophic: unable to continue safe flight and landing

  - Not expected to occur in the entire operational life of all airplanes of one type

  severe major: high workload or physical distress so crew cannot perform tasks accurately or completely

  - Not expected to occur in the entire operational life of any one airplane

  severe: significant increase in workload or conditions impairing crew efficiency

  - Expected to occur one or more times during the operational life of each airplane of one type

  minor: ...

Safety and Reliability

- Suppose the hazard is fire in the hold of an airplane
- We could eliminate this by removing oxygen
  - e.g., pressurizing the hold with nitrogen
- Or we could mitigate it with a fire suppression system
  - Then the reliability of that system becomes an issue
- Generally best to eliminate rather than mitigate hazards
- But complex systems usually have some components for mitigation or operation that require extreme reliability
- Reliability is concerned with failure, and there are several types of failure
  - e.g., loss of function, malfunction, unintended function

  The last two are generally the most serious

The 10^{-9} Requirement

- Suppose 1,000 airplanes of one type
- Each flies 3,000 hours per year
- Over a lifetime of 33 years
- That's 10^8 hours
- Suppose 10 software-based systems on board with potentially catastrophic failure conditions
- Then budget for each is a failure rate of 10^{-9} per hour, sustained for 15 hours (length of flight)
- That's where the well-known numbers come from

  catastrophic: 10^{-9} per hour

  severe major: 10^{-7} per hour

  severe: 10^{-5} per hour

Achieving 10^{-9}

- Hardware is subject to random failures at about 10^{-6}/hr
- Often worse at 35,000 feet (SEUs due to cosmic rays)
- Getting worse as transistors get smaller
- So we need fault-tolerant designs
- Fault tolerance is hard: it adds complexity
    - Intuitions of engineers from traditional disciplines (continuous math) are counterproductive, lead to failure-prone homespun designs
- Most failures in flight s/w are due to faults in fault tolerance
- So rational designs for fault-tolerant systems
    - And strong evidence for their correctness

  Are good intellectual investments

Assurance for 10^{-9}

- There are two issues for which assurance is required
  - Fault tolerance correctly deals with random faults
  - There are no design (aka. systematic) faults
- Assurance for fault tolerance, has two sub-issues
  - Given assumptions about the kinds of random faults, and their number (and/or arrival rates)
  - Prove: assumptions satisfied implies faults tolerated
  - Probabilistic analysis that assumptions will be satisfied
- Assurance for complete absence of design/software faults
  - Is unrealistic, and unnecessary (we only need 10^{-9})
  - Hence interest in software reliability

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)

Software Assurance for 10^{-9}

- How can we demonstrate that software (or any complex discrete system) has failure rates around 10^{-9}?
- Down to about 10^{-4}, it is feasible to measure software reliability by statistically valid random testing
- But 10^{-9} would need 114,000 years on test
- What we actually do is a lot of Verif'n and Valid'n (V&V)
    - Good development processes, plenty of reviews etc.
- What V&V, how much, spec'd by standards and guidelines
    - e.g., 57 V&V "objectives" at DO-178B Level C (10^{-5})
    - 65 objectives at DO-178B Level B (10^{-7})
    - 66 objectives at DO-178B Level A (10^{-9})
- How does amount of V&V (a static global concept) connect to reliability (a dynamic execution concept)?

Overview

- Now we've seen the main concepts, I can list the topics I plan to cover
- Formal Methods in support of some aspects of safety analysis
  - SMT solving, infinite bounded model checking, k-induction
  - Assumption synthesis, human factors, real-time, test generation . . .
- The nature of software assurance
- Argument-based safety cases (vs. standards)

Formal Methods

Formal Methods: Analogy with Engineering Mathematics

- Engineers in traditional disciplines build mathematical models of their designs
- And use calculation to establish that the design, in the context of a modeled environment, satisfies its requirements
- Only useful when mechanized (e.g., CFD)
- Used in the design loop (exploration, debugging)
  - Model, calculate, interpret, repeat
- Also used in certification
  - Verify by calculation that the modeled system satisfies certain requirements
- Need to be sure that model faithfully represents the design, design is implemented correctly, environment is modeled faithfully, and calculations are performed without error

Formal Methods: Analogy with Engineering Math (ctd.)

- Formal methods: same idea, applied to computational systems
- The applied math of Computer Science is formal logic
- So the models are formal descriptions in some logical system
    - E.g., a program reinterpreted as a mathematical formula rather than instructions to a machine
- And calculation is mechanized by automated deduction: theorem proving, model checking, static analysis, etc.
- The singular advantage of formal methods over testing, simulation etc., is that formal calculations (can) cover all modeled behaviors
- Because finite formulas can represent infinite sets of states
    - e.g., $x < y$ represents $\{(0,1), (0,2), \ldots (1,2), (1,3) \ldots\}$

Formal Calculations: The Basic Challenge

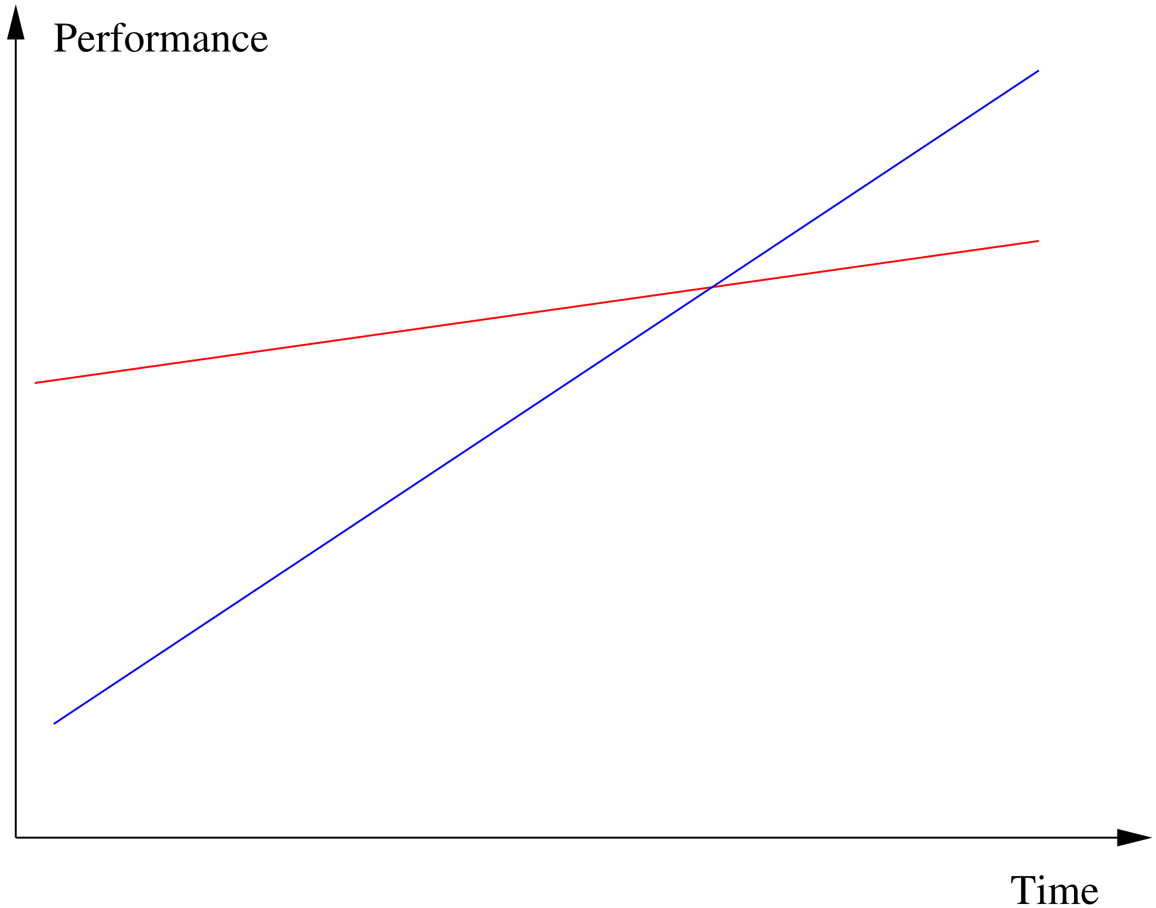
- Build mathematical model of system and deduce properties by calculation
- Calculation is done by automated deduction
- Where all problems are NP-hard, most are superexponential (2^{2^n}), nonelementary ($2^{2^{2^{\cdot^{\cdot^{\cdot}}}}}\}n$), or undecidable
- Why? Have to search a massive space of discrete possibilities
- Which exactly mirrors why it's so hard to provide assurance for computational systems
- But at least we've reduced the problem to a previously unsolved problem!

Formal Calculations: Meeting The Basic Challenge

Ways to cope with the massive computational complexity

- Use human guidance
  - That's interactive theorem proving—e.g., PVS
- Restrict attention to specific kinds of problems
  - E.g., model checking—focuses on state machines
- Use approximate models, incomplete search
  - model checkers are often used this way
- Aim at something other than verification
  - E.g., bug finding, test case generation
- Verify weak properties
  - That's what static analysis typically does
- Give up soundness and/or completeness
  - Again, that's what static analysis typically does
- Schedule a breakthrough: disruptive innovation

Disruptive Innovation



Low-end disruption is when low-end technology overtakes the performance of high-end (Christensen)

Low End Technology: SAT Solving

- Find satisfying assignment to a propositional logic formula
- Formula can be represented as a set of clauses
    - In CNF: conjunction of disjunctions
    - Find an assignment of truth values to variable that makes at least one literal in each clause TRUE
    - Literal: an atomic proposition A or its negation $\bar{A}$
- Example: given following 4 clauses
    - A, B
    - C, D
    - E
    - $\bar{A}, \bar{D}, \bar{E}$

  One solution is $A, C, E, \bar{D}$

  (A, D, E is not and cannot be extended to be one)
- Do this when there are 100,000s of variables and clauses

SAT Solvers

- SAT solving is the quintessential NP-complete problem
- But now amazingly fast in practice (most of the time)
  - Breakthroughs (starting with Chaff) since 2001
    - ⋆ Building on earlier innovations in SATO, GRASP
  - Sustained improvements, honed by competition
- Has become a commodity technology
  - MiniSAT is 700 SLOC
- Can think of it as massively effective search
  - So use it when your problem can be formulated as SAT
- Used in bounded model checking and in AI planning
  - Routine to handle 10^{300} states

SAT Plus Theories

- SAT can encode operations and relations on bounded integers
  - Using bitvector representation
  - With adders etc. represented as Boolean circuits

  And other finite data types and structures

- But cannot do not unbounded types (e.g., reals), or infinite structures (e.g., queues, lists)

- And even bounded arithmetic can be slow when large

- There are fast decision procedures for these theories

Decision Procedures

Many important theories are decidable (usually unquantified)

- Equality with uninterpreted function symbols

  $x = y \wedge f(f(f(x))) = f(x) \supset f(f(f(f(f(y))))) = f(x)$

- Function, record, and tuple updates

  f **with** $[(x) := y](z) \stackrel{\mathrm{def}}{=}$ **if** $z = x$ **then** y **else** $f(z)$

- Linear Arithmetic (over integers and rationals)

  $x \leq y \wedge x \leq 1 - y \wedge 2 \times x \geq 1 \supset 4 \times x = 2$

- It's known how to combine these

  (e.g., Nelson-Oppen method)

Can then decide the combination of theories

$$2 \times car(x) - 3 \times cdr(x) = f(cdr(x)) \supset$$
$$f(cons(4 \times car(x) - 2 \times f(cdr(x)), y)) = f(cons(6 \times cdr(x), y))$$

SMT Solving

- Individual and combined decision procedures usually decide conjunctions of formulas in their decided theories
- SMT allows general propositional structure
  - e.g., $(x \leq y \vee y = 5) \wedge (x < 0 \vee y \leq x) \wedge x \neq y$

    ... possibly continued for 1,000s of terms
- Should exploit search strategies of modern SAT solvers
- So replace the terms by propositional variables
  - i.e., $(A \vee B) \wedge (C \vee D) \wedge E$
- Get a solution from a SAT solver (if none, we are done)
  - e.g., A, D, E
- Restore the interpretation of variables and send the conjunction to the core decision procedure
  - i.e., $x \leq y \wedge y \leq x \wedge x \neq y$

SMT Solving by "Lemmas On Demand"

- If satisfiable, we are done
- If not, ask SAT solver for a new assignment
- But isn't it expensive to keep doing this?
- Yes, so first, do a little bit of work to find fragments that explain the unsatisfiability, and send these back to the SAT solver as additional constraints (i.e., lemmas)
  - $A \wedge D \supset \bar{E}$ (equivalently, $\bar{A} \vee \bar{D} \vee \bar{E}$)
- Iterate to termination
  - e.g., $A, C, E, \bar{D}$
  - i.e., $x \le y, x < 0, x \ne y, y \not\le x$ (simplifies to $x < y, x < 0$)
  - A satisfying assignment is $x = -3, y = 1$
- This is called "lemmas on demand" (de Moura, Ruess, Sorea) or "DPLL(T)"; it yields effective SMT solvers

SMT Solvers: Disruptive Innovation in Theorem Proving

- SMT stands for Satisfiability Modulo Theories
- SMT solvers extend decision procedures with the ability to handle arbitrary propositional structure
    - Traditionally, case analysis is handled heuristically in the theorem prover front end
        - Where must be careful to avoid case explosion
    - SMT solvers use the brute force of modern SAT solving
- Or, dually, they generalize SAT solving by adding the ability to handle arithmetic and other decidable theories
- There is an annual competition for SMT solvers
- Very rapid growth in performance
- Application to verification
    - Via bounded model checking and k-induction
- And to synthesis, by solving exists-forall problems

Bounded Model Checking (BMC)

- Given system specified by initiality predicate I and transition relation T on states S
- Is there a counterexample to property P in k steps or less?
- Can try $k = 1, 2, \ldots$
- Find assignment to states $s_0, \ldots, s_k$ satisfying

  $$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg(P(s_1) \wedge \cdots \wedge P(s_k))$$

- Given a Boolean encoding of I, T, and P (i.e., circuit), this is a propositional satisfiability (SAT) problem
- If I, T, and P are over the theories decided by an SMT solver, then this is an SMT problem
  - Called Infinite Bounded Model Checking (inf-BMC)
- Works for LTL (via Büchi automata), not just invariants

Verification via BMC: k-Induction

- Ordinary inductive invariance (for P):

  Basis: $I(s_0) \supset P(s_0)$

  Step: $P(r_0) \wedge T(r_0, r_1) \supset P(r_1)$

- Extend to induction of depth k:

  Basis: No counterexample of length k or less (i.e., Inf-BMC)

  Step: $P(r_0) \wedge T(r_0, r_1) \wedge P(r_1) \wedge \cdots \wedge P(r_{k-1}) \wedge T(r_{k-1}, r_k) \supset P(r_k)$

  This is a close relative of the BMC formula

- Works for LTL safety properties, not just invariants
- Induction for $k = 2, 3, 4 \ldots$ may succeed where $k = 1$ does not
- Note that counterexamples help debug invariant
- Can easily extend to use lemmas
- Inf-BMC blurs line between model checking and theorem proving: automation, counterexamples, with expressiveness

Applications of Inf-BMC

Hazard Analysis

- We need systematic ways to search for hazards
- In physical systems it is common to look for sources of energy and trace their propagation
- Also look at "pipework" and raise questions prompted by a list of guidewords
    - e.g., too much, not enough, early, late, wrong

  This is called HAZOP, and it can be reinterpreted for software (look at data and control flows)
- Can also suppose there has been a system failure, then ask what could have brought this about
    - This is fault tree analysis (FTA)
- Or suppose some component has failed and ask what the consequences could be
    - This is failure modes and effects analysis (FMEA)

Hazard Analysis as Model Checking

- We can think of many safety analyses as attempts to anticipate all possible scenarios/reachable states to check for safety violations: it's like state exploration/model checking
- But generally applied to very abstract system model
    - Done early in the lifecycle, few details available
    - Analysis done by hand, cannot handle state explosion
- Analysis is approximate (because done by hand)
    - Explore only paths likely to contain violations
    - e.g., those that start from component failure (FMEA)
    - Or backwards from those ending in system failure (FTA)
- Could be improved by automation
- Provided we can stay suitably abstract

Partially Mechanized Hazard Analysis

- Classical model checking (explicit, symbolic, bounded) requires a totally concrete design (equivalent to a program)
- Interactive theorem proving can deal with abstract designs
- Use of uninterpreted functions, predicates, and types is key
  - $f(x), g(p, q)$ etc. where you know nothing about f, g
  - Except what you add via axioms
- Generally eschewed in HOL, Isabelle, ACL2
  - Axioms may introduce inconsistencies
- Welcomed in PVS
  - Soundness guaranteed via theory interpretation and constructive model
  - Has decision procedure for ground equality with uninterpreted functions
- But I'm after pushbutton automation

Mechanized Hazard Analysis/Assumption Synthesis

- Aha! Inf-BMC can do this
- But how to introduce axioms/assumptions on the uninterpreted functions?
- Aha! Can do this via synchronous observers

Synchronous Observers

- Observers are components in a model that “watch” the behavior of other components: a bit like an assert statement
- Observers raise a flag under certain conditions
  - Can encode safety and bounded liveness properties

  Easy for engineers to write (same language as model)
- Verification
  - Flag raised on property violation
  - Model check for `G(flag down)`
- Test generation
  - Flag raised when a good test is recognized
  - Model check for `G(flag down)`, counterexample is test
- Hazard analysis/assumption synthesis
  - Flag raised when assumptions violated
  - Model check for `G(flag down => behavior is correct)`

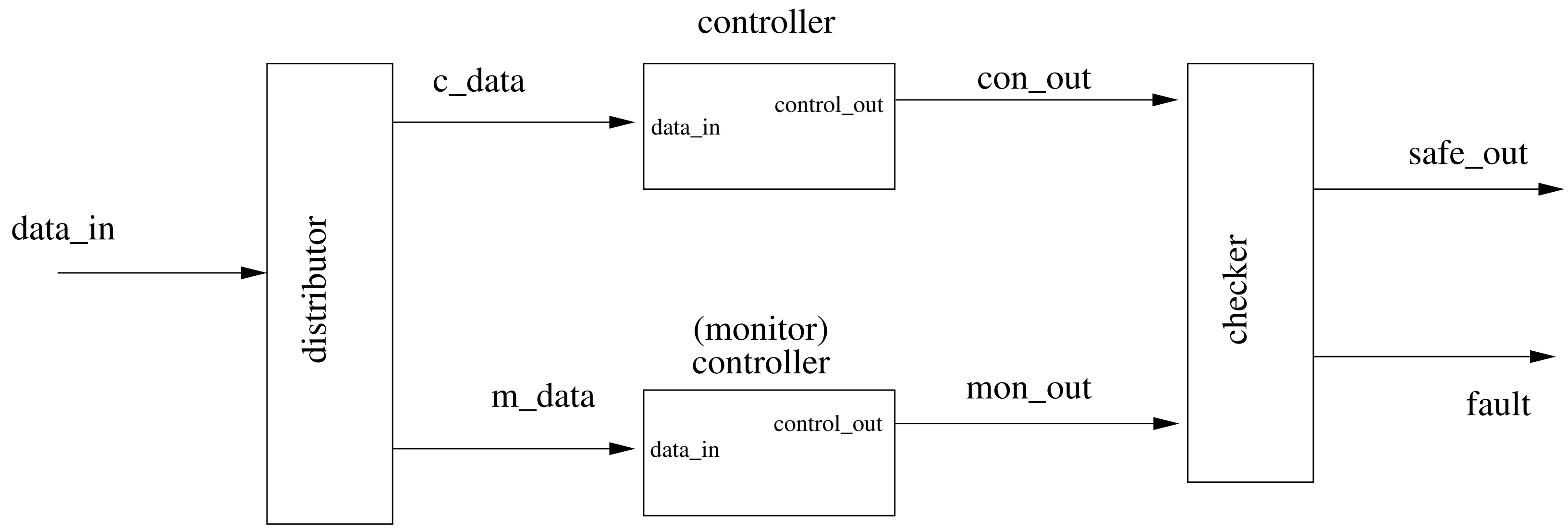
Example: Assumption Synthesis

- This is related to hazard analysis, as we'll see
- Recall for fault tolerance, we need to prove
  - assumptions satisfied implies faults tolerated
- We turn it around and ask under what assumptions does our design work (i.e., tolerate faults)?
- Violations of these assumptions are then the hazards to this design
- We must find all these hazards and consider their probability of occurrence

Example: Self-Checking Pair

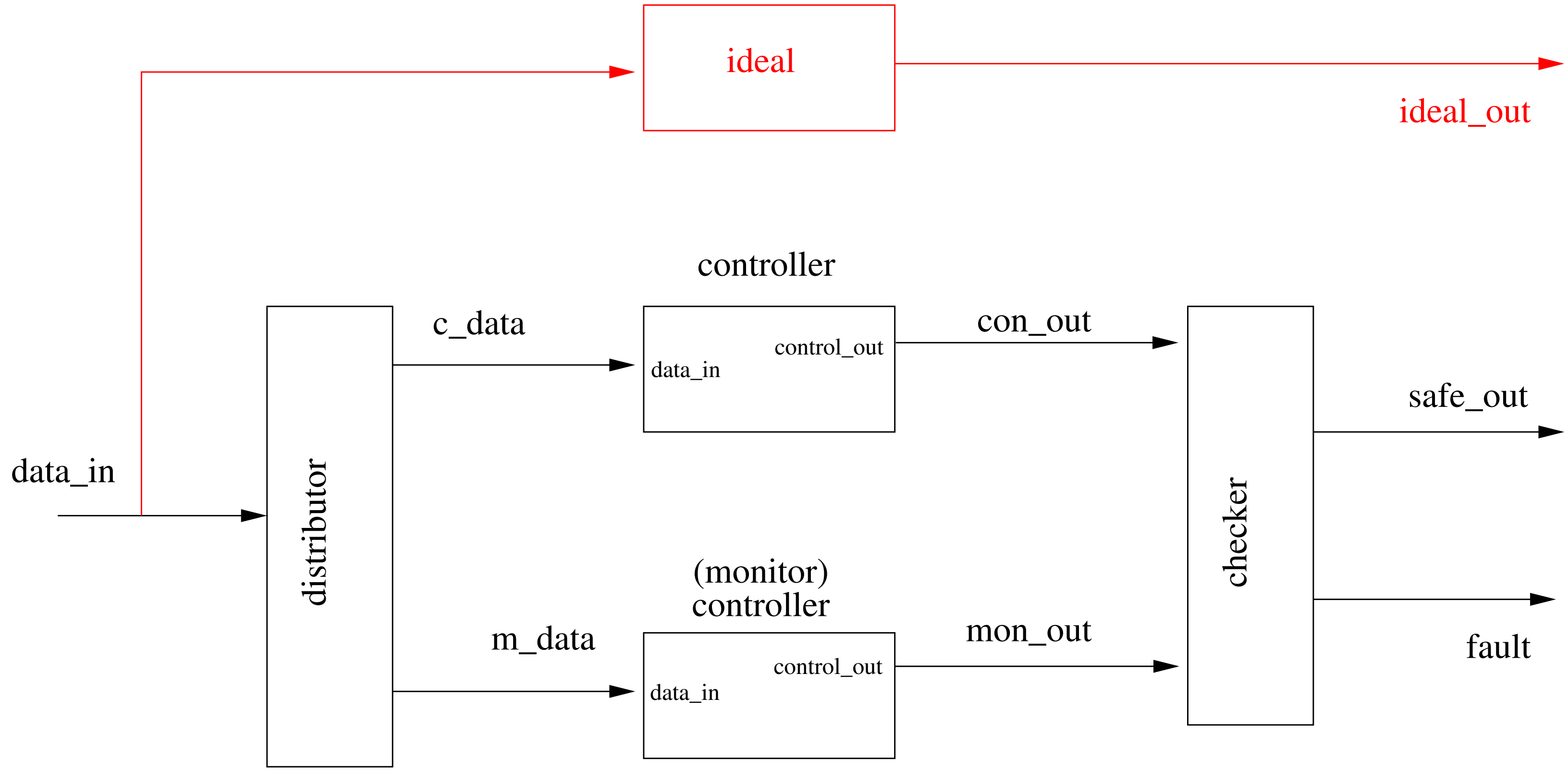
- A component that fails by stopping cleanly is fairly easy to deal with
- The danger is components that do the wrong thing
- We're concerned with random faults, so faults in separate components should be independent
  - Provided they are designed as fault containment units (FCUs) — independent power supplies, locations etc.
  - And ignoring high intensity radiated fields (HIRF) — and other initiators of correlated faults
- So we can duplicate the component and compare the outputs
  - Pass on the output when both agree
  - Signal failure on disagreement
- Under what assumptions does this work?

Example: Self-Checking Pair (ctd. 1)



- Controllers apply some control law to their input
- Controllers and distributor can fail
  - For simplicity, checker is assumed not to fail
- Need some way to specify requirements and assumptions
- Aha! correctness requirement can be an idealized controller

Example: Self-Checking Pair (ctd. 2)

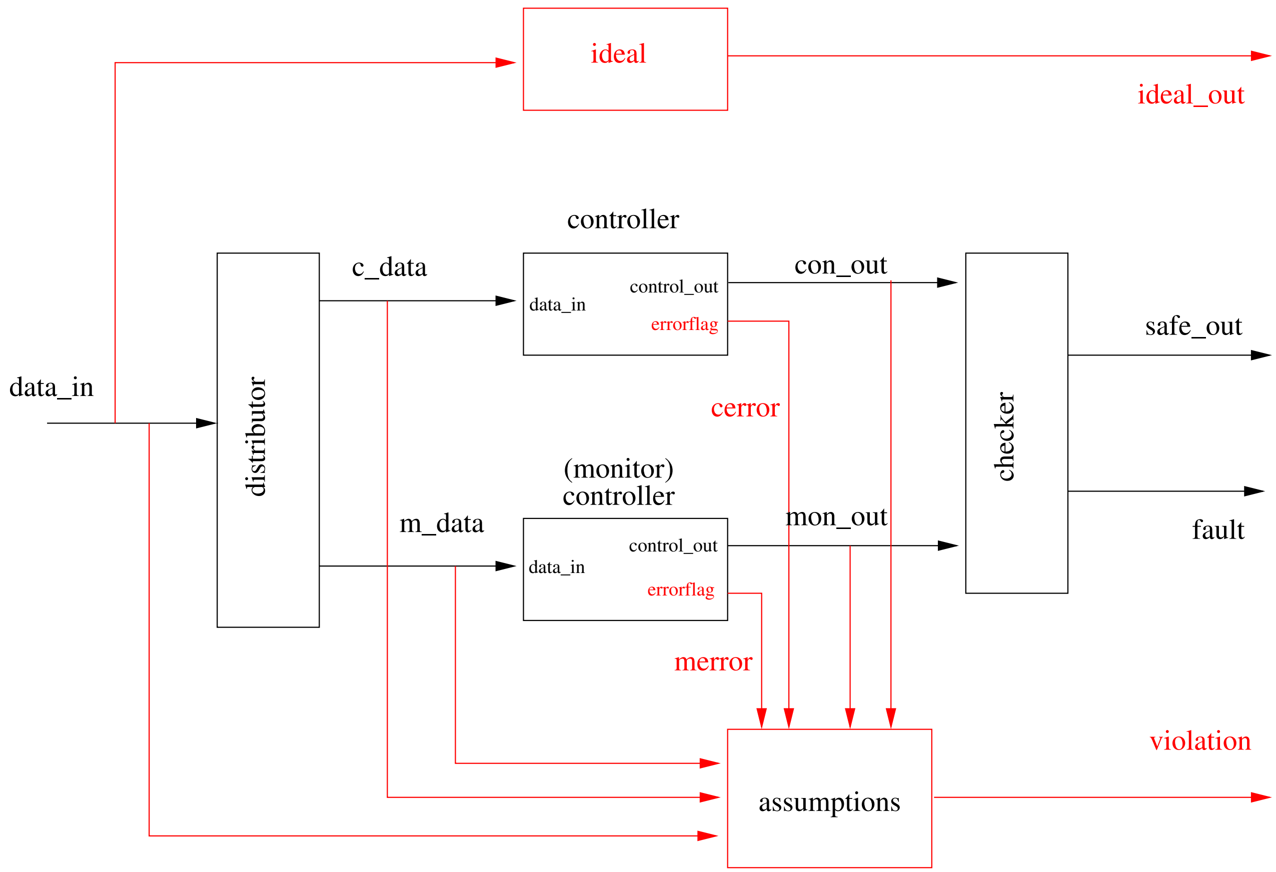


The controllers can fail, the ideal cannot

If no `fault` indicated `safe_out` and `ideal_out` should be the same

Model check for `G((NOT fault => safe_out = ideal_out))`

Example: Self-Checking Pair (ctd. 3)



We need assumptions about the types of fault that can be tolerated: encode these in the assumptions observer

```
G(violation = down => (NOT fault => safe_out = ideal_out))
```

Synthesized Assumptions for Self-Checking Pair

- We will examine this example with the SAL model checker
- Initially, no assumptions
- Counterexamples help us understand what is wrong or missing
- Will discover four assumptions
- Then verify that the design is correct under these assumptions
- Then consider the probability of violating these assumptions and modify our design so that the most likely one is eliminated

selfcheck.sal: Types

```
selfcheck: CONTEXT =
BEGIN

sensor_data: TYPE;

actuator_data: TYPE;
init: actuator_data;

laws(x: sensor_data): actuator_data;

metasignal: TYPE = {up, down};
```

selfcheck.sal: Ideal Controller

```
ideal: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  ideal_out: actuator_data
INITIALIZATION
  ideal_out = init;
TRANSITION
  ideal_out' = laws(data_in)
END;
```

selfcheck.sal: Ordinary Controller

```
controller: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  control_out: actuator_data,    errorflag: metasignal
INITIALIZATION
  control_out = init;    errorflag = down;
TRANSITION
[ normal: TRUE -->
   control_out' = laws(data_in); errorflag' = down;
[] hardware_fault: TRUE -->
   control_out' IN {x: actuator_data | x /= laws(data_in)};
   errorflag' = up;
] END;
```

selfcheck.sal: Distributor

```
distributor: MODULE =
BEGIN
INPUT
  data_in: sensor_data
OUTPUT
  c_data, m_data: sensor_data
INITIALIZATION
  c_data = data_in; m_data = data_in;
TRANSITION
[  distributor_ok:  TRUE -->
    c_data' = data_in'; m_data' = data_in';
[] distributor_bad: TRUE -->
    c_data' IN {x: sensor_data | TRUE};
    m_data' IN {y: sensor_data | TRUE};
] END;
```

selfcheck.sal: Checker

```
checker: MODULE =
BEGIN
INPUT
  con_out: actuator_data,    mon_out: actuator_data
OUTPUT
  safe_out: actuator_data,   fault: boolean
INITIALIZATION
  safe_out = init;   fault = FALSE;
TRANSITION
safe_out' = con_out';
[
disagree:  con_out' /= mon_out' --> fault' = TRUE
[] ELSE -->
]
END;
```

selfcheck.sal: Wiring up the Self-Checking Pair

```
scpair: MODULE = distributor
 || (RENAME
      control_out TO con_out,
      data_in TO c_data,
      errorflag TO cerror
    IN controller)

 || (RENAME
      control_out TO mon_out,
      data_in to m_data,
      errorflag TO merror
    IN controller);

 || checker
```

selfcheck.sal: Assumptions

```
assumptions: MODULE =
BEGIN
OUTPUT
  violation: metasignal
INPUT
  data_in, c_data, m_data: sensor_data,
  cerror, merror: metasignal,
  con_out, mon_out: actuator_data
INITIALIZATION
 violation = down
TRANSITION
[ assumption_violation:
   FALSE % OR your assumption here (actually hazard)
      --> violation' = up;
[] ELSE --> ] END;
```

selfcheck.sal: Testing the Assumptions

```
scpair_ok: LEMMA
 scpair || assumptions || ideal |-
   G(violation = down
      => (NOT fault => safe_out = ideal_out));
```

```
% sal-inf-bmc selfcheck scpair_ok -v 3 -it
```

Assumption Synthesis: First Counterexample

- Both controllers have hardware faults
- And generate same, wrong result
- Derived hazard (assumption is its negation)

  ```
  cerror' = up AND merror' = up AND con_out' = mon_out'
  ```

  Assumption module reads data of different "ticks";
  important to reference correct values (new state here)
- This hazard requires a double failure
  - Any double failure may be considered improbable
- Here, require double failure that gives same result
  - Highly improbable, unless a systematic fault

  Worth thinking about

Assumption Synthesis: Second Counterexample

- Distributor has a fault: sends wrong value to one controller
- The controller that got the good value has a fault, generates same result as correct one that got the bad input
- Derived hazard (assumption is its negation)

```
m_data /= c_data
  AND (merror' = up OR cerror' = up)
   AND mon_out' = con_out'
```

- Double fault, so highly improbable

Assumption Synthesis: Third Counterexample

- Distributor has a fault: sends (different) wrong value(s) to one or both controllers: Byzantine/SOS fault
- It just happens the different inputs produce same outputs
- Very dubious you could find this with a concrete model
  - Such as is needed for conventional model checking
  - Likely to use `laws(x) = x+1` or similar
- Derived hazard (assumption is its negation)

```
m_data /= c_data
  AND (merror' = down AND cerror' = down)
   AND mon_out' = con_out'
```

- Quite plausible
- But fixable: pass inputs to checker
  - This also reduces likelihood of the previous hazard

Assumption Synthesis: Fourth Counterexample

- Distributor has a fault: sends same wrong value to both controllers

- Derived hazard (assumption is its negation)

```
m_data = c_data AND m_data /= data_in
```

- This one we need to worry about

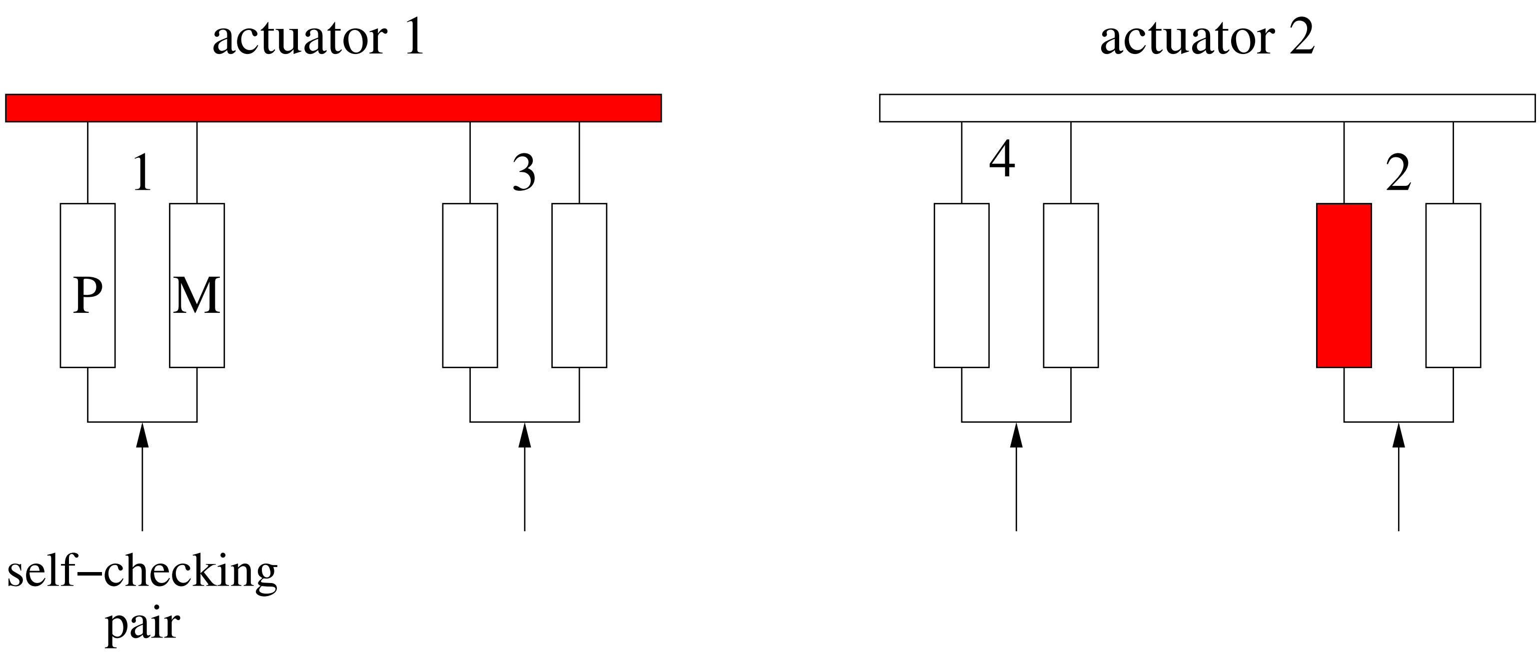
- Byzantine/SOS fault at the distributor is most likely to generate the previous two cases
    - This is an unlikely random fault, but suggests a possible systematic fault

Assumption Synthesis Example: Summary

- We found four assumptions for the self-checking pair
  - When both members of pair are faulty, their outputs differ
  - When the members of the pair receive different inputs, their outputs should differ
    - When neither is faulty: can be eliminated
    - When one or more is faulty
  - When both members of the pair receive the same input, it is the correct input
- Can prove by 1-induction that these are sufficient
  - `sal-inf-bmc selfcheck scpair_ok -v 3 -i -d 1`
- One assumption can be eliminated by redesign, two require double faults
- Attention is directed to the most significant case

Aside: Dealing with Actuator Faults

- One approach, based on self-checking pairs does not attempt to distinguish computer from actuator faults
- Must tolerate one actuator fault and one computer fault simultaneously



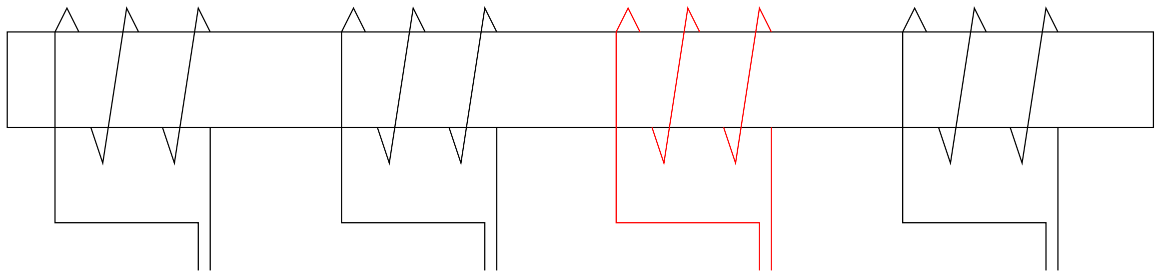
- Can take up to four frames to recover control

Consequences of Slow Recovery

- Must use large, slow moving ailerons rather than small, fast ones
  - Hybrid systems/control theory verification question: why?
- As a result, wing is structurally inferior
- Holds less fuel
- And plane has inferior flying qualities
- All from a choice about how to do fault tolerance

Physical Averaging At The Actuators

- Alternative uses averaging at the actuators
  - E.g., multiple coils on a single solenoid



  - Or multiple pistons in a single hydraulic pot

- Hybrid systems verification question: how well can this work?

Conclusions

- Formal methods, and formal analysis and calculation can be used for several purposes
  - Verification
  - Consistency and completeness checking
  - And also exploration, synthesis, test generation
- Most faults, and most serious faults, are introduced early in the lifecycle
  - Hazard analysis and safety analysis (see Nimrod report)
  - Requirements

  So that's where the biggest payoff for formal methods may be
- Need abstraction and automation: Inf-BMC is a suitable tool

More Examples

If there is interest, we can look at more examples using Inf-BMC and other automated techniques at 13:30 on Wednesday

- Byzantine agreement (cf. assumptions needed for the self-checking pair)
    - This is a good example to compare SMC and BMC, and to see “dial twiddling” in SMC
- Real time systems
- Automated test generation
- Human factors (mental models)
- Doron’s little fairness example
- Relational algebra in PVS

Argument-Based Safety Cases

The Basis For Assurance and Certification

- We have claims or goals that we want to substantiate
  - In our case, they will claims be about safety
  - In other fields, they may be about security, or performance
  - Or some combination

  E.g., no catastrophic failure condition in the life of the fleet

- We produce evidence about the product and its development process to support the claims
  - E.g., analysis and testing of the product and its design
  - And documentation for the process of its development
- And we have an argument that the evidence is sufficient to support the claims
- Surely, this is the intellectual basis for all certification regimes

Standards-Based Approaches to Certification

- Applicant follows a prescribed process
  - Delivers prescribed outputs
    - ⋆ e.g., documented requirements, designs, analyses, tests and outcomes; traceability among these

  These provide evidence

- The goals and argument are largely implicit

- DO-178B (civil aircraft) is like this

- Works well in fields that are stable or change slowly
  - No accidents due to software, but several incidents
  - Can institutionalize lessons learned, best practice
    - ⋆ e.g. evolution of DO-178 from A to B to C

Standards-Based Approaches to Certification (ctd.)

- May be less suitable with novel problems, solutions, methods
- Basis in lessons learned may not anticipate new challenges
  - NextGen (decentralized air traffic control) may be like this
  - Also rapidly moving fields, like medical devices
- Basis in tried-and-true methods can be a brake on innovation
  - Reluctance to use automated verification may be like this
- In the absence of explicit arguments, don't know what alternative evidence might be equally or more effective
  - E.g., what argument does MC/DC testing support?
  - MC/DC is a fairly onerous structural coverage criterion
  - For DO-178B Level A, must generate tests from requirements, achieve MC/DC on the code

Predator Crash near Nogales

- NTSB A-07-65 through 86
- Predator B crashed near Nogales NM, 25 April 2006
- Operated by Customs and Border Protection
- Pilot inadvertently shutdown the engine
  - Numerous operational errors
- No engine, so went to battery power
- Battery bus overload
- Load shedding turned off satcomms and transponder
- Descended out of control through civil airspace with no transponder and crashed 100 yards from a house
- Novel kind of system, or just didn't do a good job?

Another Recent Incident

- Fuel emergency on Airbus A340-642, G-VATL, on 8 February 2005 (AAIB SPECIAL Bulletin S1/2005)
- Toward the end of a flight from Hong Kong to London: two engines flamed out, crew found certain tanks were critically low on fuel, declared an emergency, landed at Amsterdam
- Two Fuel Control Monitoring Computers (FCMCs) on this type of airplane; they cross-compare and the "healthiest" one drives the outputs to the data bus
- Both FCMCs had fault indications, and one of them was unable to drive the data bus
- Unfortunately, this one was judged the healthiest and was given control of the bus even though it could not exercise it
- Further backup systems were not invoked because the FCMCs indicated they were not both failed

Implicit and Explicit Factors

- See also ATSB incident report for in-flight upset of Boeing 777, 9M-MRG (Malaysian Airlines, near Perth Australia)
- And accident report for violent pitching of A330, VH-QPA (QANTAS, near Perth Australia)
- How could gross errors like these pass through rigorous assurance standards?
- Maybe effectiveness of current certification methods depends on implicit factors such as safety culture, conservatism
- Current business models are leading to a loss of these
  - Outsourcing, COTS, complacency, innovation
- Surely, a credible certification regime should be effective on the basis of its explicit practices
- How else can we cope with challenges of the future?

The Argument-Based Approach to Certification

- E.g., UK air traffic management (CAP670 SW01), defence (DefStan 00-56), Railways (Yellow Book), EU Nuclear, growing interest elsewhere (e.g., FDA, NTSB)
- Applicant develops a safety case
  - Whose outline form may be specified by standards or regulation (e.g., 00-56)
  - Makes an explicit set of goals or claims
  - Provides supporting evidence for the claims
  - And arguments that link the evidence to the claims
    - ⋆ Make clear the underlying assumptions and judgments
    - ⋆ Should allow different viewpoints and levels of detail
- The case is evaluated by independent assessors
- Generalized to security, dependability, assurance cases

Pros and Cons

- The main novelty in safety cases is the explicit argument
- Allows innovation, helps in accident analysis
- Could alleviate some burdens at higher DALs/SILs
- But how credible is the assessment of a novel argument?
    - cf. Nimrod safety case
- Especially when real safety cases are huge
- Need tools to manage/analyze large cases
- Safety cases had origin in UK due to numerous disasters: maybe they should just learn to apply standards
- Standards establish a floor
- Can still employ standards in parts of a case

Standards in Argument-Based Safety Cases

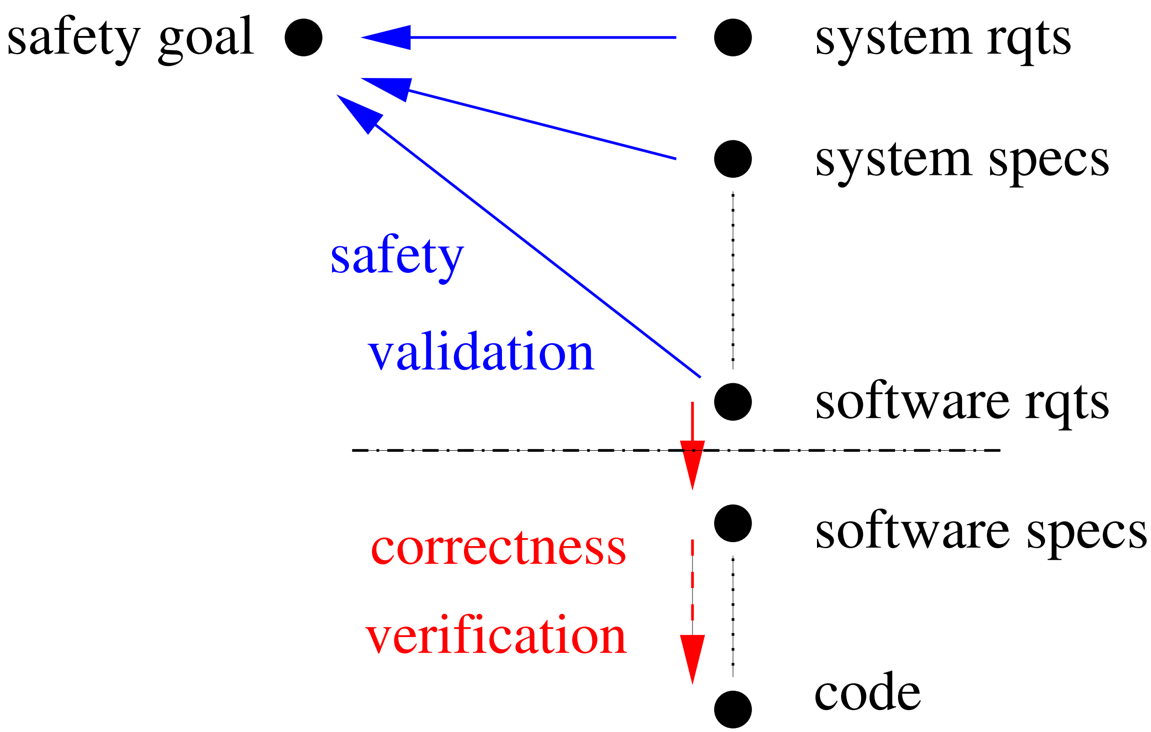
- Cases contain high-level elements such as "all hazards identified and dealt with"
- All hazards? How do we argue this?
- Could appeal to accepted process for hazard identification
  - E.g., ISO 14971 (medical devices)
- So there is a role for standards within safety cases
- Not enough to say "we applied 14971"
- Need to supply evidence from its application to this case

Safety and Correctness

- Currently, we apply safety analysis methods to an informal system description
  - Little automation, but in principle
  - These are abstracted ways to examine all reachable states
  - So the tools of automated verification (a species of formal methods) can assist here

- Then, to be sure the implementation does not introduce new hazards, we require it exactly matches the analyzed description
  - Hence, most implementations standards are about correctness, not safety
  - And are burdensome at higher DALs/SILs

Implementation Standards Focus on Correctness



- As more of the system design goes into software
- The design/implementation boundary should move
- Safety/correctness analysis moves with it

Systems and Components

- The FAA certifies airplanes, engines and propellers
- Components are certified only as part of an airplane or engine
- That's because it's the interactions that matter and it's not known how to certify these compositionally
  - i.e., in a modular manner
- But modern engineering and business practices use massive subcontracting and component-based development that provide little visibility into subsystem designs
- And for systems like NextGen there is no choice but a compositional approach
- And it may in part be an adaptive system
  - i.e., some of its behavior determined at runtime

Compositional and Incremental Certification

- These are immensely difficult
  - Undesired emergent behavior due to interactions
- Safety case may not decompose along architectural lines
  - Important insight (Ibrahim Habli & Tim Kelly)
- But, in some application areas we can insist that it does
  - Goes to the heart of what is an architecture
  - A good one supports and enforces the safety case
- This is what partitioning in IMA is all about
  - IMA is integrated modular avionics
- But also need better understanding and control of failures in intended interactions
  - cf. elementary and composite interfaces (Kopetz)

Two Kinds of Uncertainty In Certification

- One kind concerns failure of a claim, usually stated probabilistically (frequentist interpretation)
  - E.g., 10^{-9} probability of failure per hour, or 10^{-3} probability of failure on demand
- The other kind concerns failure of the assurance process
  - Seldom made explicit
  - But can be stated in terms of subjective probability
    - $\star$ E.g., 95% confident this system achieves 10^{-3} probability of failure on demand
- Demands for multiple sources of evidence are generally aimed at the second of these

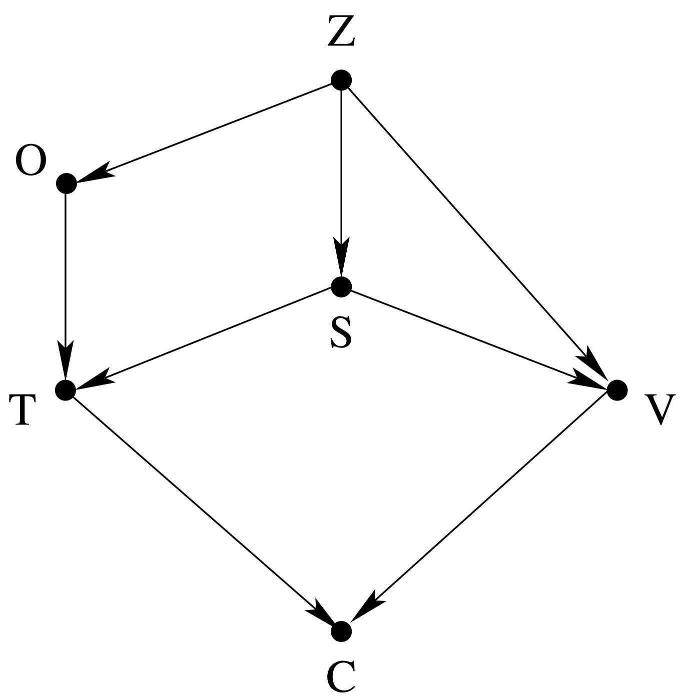
Probabilistic Support for Arguments

- If each assumption or subargument has only a probability of being valid
- How valid is the conclusion?
- Difficult topic, several approaches, none perfect
  - Probabilistic logic (Carnap)
  - Evidential reasoning (Dempster-Shaefer)
  - Bayesian analysis and Bayesian Belief Nets (BBNs)
- Common approach, recently shown sound, is to develop assurance for, say 10^{-5} with 95% confidence, then use within the safety case as if it were 10^{-3} with certainly
- May also employ multi-legged arguments

Bayesian Belief Nets

- Bayes Theorem is the principal tool for analyzing subjective probabilities
- Allows a prior assessment of probability to be updated by new evidence to yield a rational posterior probability
    - E.g., P(C) vs. P(C | E)
- Math gets difficult when the models are complex
    - i.e., when we have many conditional probabilities of the form p(A | B and C or D)
- BBNs provide a graphical representation for hierarchical models, and tools to automate the calculations
- Can allow principled construction of multi-legged arguments

BBN Example (Multi-Legged Argument)



Z: System Specification

O: Test Oracle

S: System's true quality

T: Test results

V: Verification outcome

C: Conclusion

Example joint probability table: successful test outcome

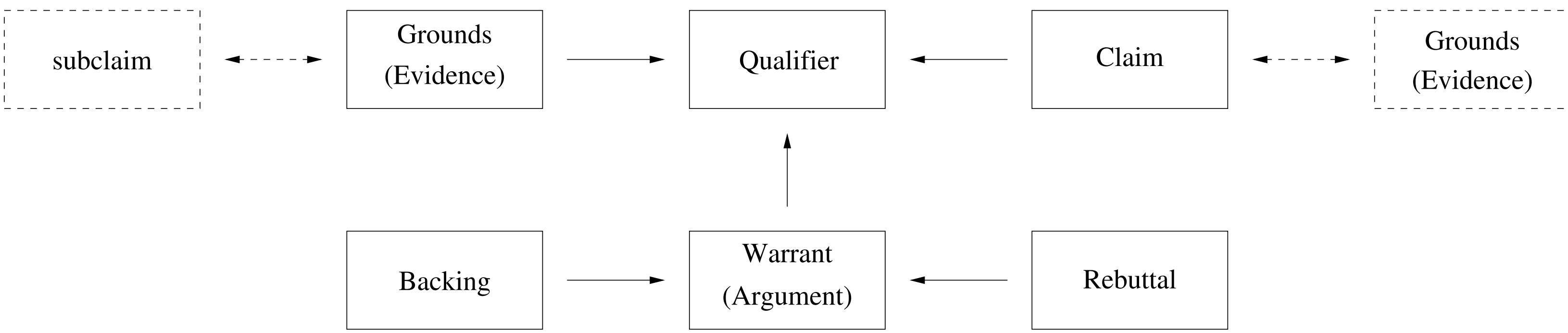
Correct System		Incorrect System	
Correct Oracle	Bad Oracle	Correct Oracle	Bad Oracle
100%	50%	5%	30%

Connections to Philosophy

- Philosophy of science touches on similar topics
- Theories cannot be confirmed, only refuted (Popper)
- Yes, but some theories have more confirmation than others
- Studied by confirmation theory
  (part of Bayesian Epistemology)
- Confirmation for claim C given evidence E:
  $c(C, E) = P(E \mid C) - P(E \mid \text{not } C)$
  Or logarithm of this
- Argumentation is also a topic of study, distinct from formal proof (there are journals devoted to this)

Argumentation

- Certification is ultimately a judgement
- So classical formal reasoning may not be entirely appropriate
- Advocates of assurance cases often look to Toulmin's model of argument



GSN, CAE are based on this

Toulmin's Model of Argument (ctd.)

Claim: This is the expressed opinion or conclusion that the arguer wants accepted by the audience

Grounds: This is the evidence or data for the claim

Qualifier: An adverbial phrase indicating the strength of the claim (e.g., certainly, presumably, probably, possibly, etc.)

Warrant: The reasoning or argument (e.g., rules or principles) for connecting the data to the claim

Backing: Further facts or reasoning used to support or legitimate the warrant

Rebuttal: Circumstances or conditions that cast doubt on the argument; it represents any reservations or "exceptions to the rule" that undermine the reasoning expressed in the warrant or the backing for it

Formal Methods Support for Arguments

- Formal logic focuses on inference whereas in safety cases we're interested in justification and persuasion
- Toulmin stresses these
- Makes sense if we're arguing about aesthetics or morality, where reasonable people may have different views
  - But we're arguing about properties of designed artifacts
- Furthermore, he had only the logic technology of 1950
- I suspect we can now do better using formal verification technology to represent and analyze cases
  - Make cases "active" so you can explore them like a spreadsheet: use Inf-BMC or other automation to allow interactive examination
  - But no illusions that you can "verify" each subclaim

Argument Support for Formal Methods

- Formal verification typically proves
  - assumptions + design => requirements

  And we think of the proof as absolute, but
- How do we know these are the right assumptions?
- How do we know these are the right requirements?
- How do we know the design is implemented correctly?
  - All the way down
- Is this really the whole design?
- Safety cases provide a framework for addressing these
  - Provides the useful notion of assurance deficit

Conclusion

- Whether done by following standards or by developing an argument-based safety case, assurance for a safety-critical system is a huge amount of work, all of it important
- Human judgement and experience are vital
- I believe we should use formal methods to the greatest extent possible, and in such a way that human talent is liberated to focus on the topics that really need it
- So I adovcate using formal methods for exploration and synthesis, in addition to verification
- And also to tie the whole argument (including its nonformal parts) together
- And I advocate ecelecticism in methods and tools
  - Loose federations interacting via a toolbus
- What do you think?

What Does V&V Achieve?

This is joint work with Bev Littlewood (City Univ, London)

Measuring/Predicting Software Reliability

- For pfds down to about 10^{-4}, it is feasible to measure software reliability by statistically valid random testing
- But 10^{-9} would need 114,000 years on test
- So how do we establish that a piece of software is adequately reliable for a system that requires extreme reliability?
- Most standards for system safety (e.g., IEC 61508, DO178B) require you to show that you did a lot of V&V
  - e.g., 57 V&V "objectives" at DO178B Level C (10^{-5})
- And you have to do more for higher levels
  - 65 objectives at DO178B Level B (10^{-7})
  - 66 objectives at DO178B Level A (10^{-9})
- What's the connection between amount of V&V and degree of software reliability?

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

Aleatory and Epistemic Uncertainty for Software

- We have some probabilistic property of the software's dynamic behavior
    - There is aleatoric uncertainty due to variability in the circumstances of the software's operation

- We examine the static attributes of the software to form an epistemic estimate of the property
    - More examination refines the estimate

- For what kinds of properties does this work?

Perfect Software

- Property cannot be about individual executions of the software
  - Because the epistemic examination is static (i.e., global)
  - This is the difficulty with reliability
- Must be a global property, like correctness
- But correctness is relative to specifications, which themselves may be flawed
- We want correctness relative to the critical claims
- Call that perfection
- Software that will never experience a 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 V&V it has had, the greater that possibility
- So we can speak of a probability of perfection
  - A subjective probability
- For a frequentist interpretation, think of all the software that might have been developed by comparable engineering processes to solve the same design problem as the software at hand
  - And that has had the same degree of V&V
  - 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 correctness-based V&V
- And it also relates to reliability:

  By the formula for total probability

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

- 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 system failure will be given by the Riemann-Stieltjes integral

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

- 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

Crude Epistemic Estimation

- If beliefs cannot be separated, we can make conservative approximations to assess $P(\text{software fails}) < p_{fnp} \times p_{np}$
- Assume software always fails if it is imperfect (i.e., $p_{fnp} = 1$)
- Then, very crudely, and very conservatively,

$$P(\text{software fails}) < P(\text{software imperfect})$$

  Dually, probability of perfection is a lower bound on reliability

- Alternatively, can assume software is imperfect (i.e., $p_{np} = 1$)
  - This is the conventional assumption
  - Estimate of p_{fnp} is then taken as system failure rate
  - Any value $p_{np} < 1$ would improve this

Less Crude Epistemic Estimation

- Littlewood and Povyakalo show that if we have
  - $p_{np} < a$ with doubt A (i.e., confidence $1 - A$)
  - $p_{fnp} < b$ with doubt B (i.e., $P(p_{fnp} < b) > 1 - B$)

  Then system failure rate is less than $a \times b$ with doubt $A + B$

- e.g., p_{np}, p_{fnp} both 10^{-3} at 95% confidence,
  gives 10^{-6} for system at 90% confidence

- They also show (under independence assumption) that large number of failure-free runs shifts assessment from imperfect but reliable toward perfect

- Also some evidence for perfection can come from other comparable software

Two Channel Systems

- We saw a self-checking pair earlier
- Components were identical; threat was random faults
- Diverse components could protect against software faults
- Many safety-critical systems have two (or more) diverse "channels" like this: e.g., nuclear shutdown, flight control
- One operational channel does the business
- A simpler channel provides a backup or monitor
- 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 reliabilty delivered

Two Channel Systems and Possible Perfection

- But if the second channel is simple enough to support a plausible claim of possible perfection
  - 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
  - $P(\text{system fails on randomly selected demand} \leq pfd_A \times pnp_B$
- Epistemic assessment raises same issues as before
- 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

Type 1 and Type 2 Backup/Monitor Failures

- Fuel emergency on Airbus A340-642, G-VATL, 8 February 2005
  - Type 1 failure: monitor did not work
- EFIS Reboot during spin recovery on Airbus A300 (American Airlines Flight 903), 12 May 1997
  - Type 2 failure: monitor triggered false alarm
- These were the wrong way round on preprints
- Full treatment derives risk of these kinds of failures given possibly perfect backups or monitors
- Current proposals are for formally synthesized/verified monitors
- So estimates of perfection for formal monitors are of interest

Formal Verification and the Probability of Perfection

- We want to assess p_{np}
- Context is likely a safety case in which claims about a system are justified by an argument based on evidence about the system and its development
- Suppose part of the evidence is formal verification
- ◦ i.e., what is the probability of perfection of formally verified software?
- Let's consider where formal verification can go wrong

The Basic Requirements For The Software Are Wrong

- This error is made before any formalization
- It seems to be the dominant source of errors in flight software
- But monitoring and backup software is built to requirements taken directly from the safety case
    - If these are wrong, we have big problems
- In any case, it's not specific to formal verification
- So we'll discount this concern

The Requirements etc. are Formalized Incorrectly

- Could also be the assumptions, or the design
- Formalization may be inconsistent
  - i.e., meaningless

  Can be eliminated using constructive specifications
  - In a tool-supported framework
  - That guarantees conservative extension

  But that's not always appropriate
  - Prefer to state assumptions as axioms
  - Consistency can then be guaranteed by exhibiting a constructive model (interpretation)
  - PVS can do this
- So we can eliminate concern about inconsistency

The Requirements etc. are Formalized Incorrectly (ctd.)

- Formalization may be consistent, but wrong
- Formal specifications that have not been subjected to analysis are no more likely to be correct than programs that have never been run
  - In fact, less so: engineers have better intuitions about programs than specifications
- Should challenge formal specifications
  - Prove putative theorems
  - Get counterexamples for deliberately false conjectures
  - Directly execute them on test cases
- Social process operates on widely used theories
- In my experience, incorrect formalization is the dominant source of errors in formal verification
  - There are papers on errors in my specifications

The Requirements etc. are Formalized Incorrectly (ctd. 2)

- Even if a theory or specification is formalized incorrectly, it does not necessarily invalidate all theorems that use it
- Only if the verification actually exploits the incorrectness will the validity of the theorem be in doubt
    - Even then, it could still be true, but unproven
- Some verification systems identify all the axioms and definitions on which a formally verified conclusion depends
    - PVS does this

  If these are correct, then logical validity of the verified conclusion follows by soundness of the verification system
    - Can apply special scrutiny to them
- So concern about incorrect formalization can be managed

The Formal Specification and Verification is Discontinuous or Incomplete

- Discontinuities arise when several analysis tools are applied in the same specification
  - e.g., static analyzer, model checker, timing analyzer

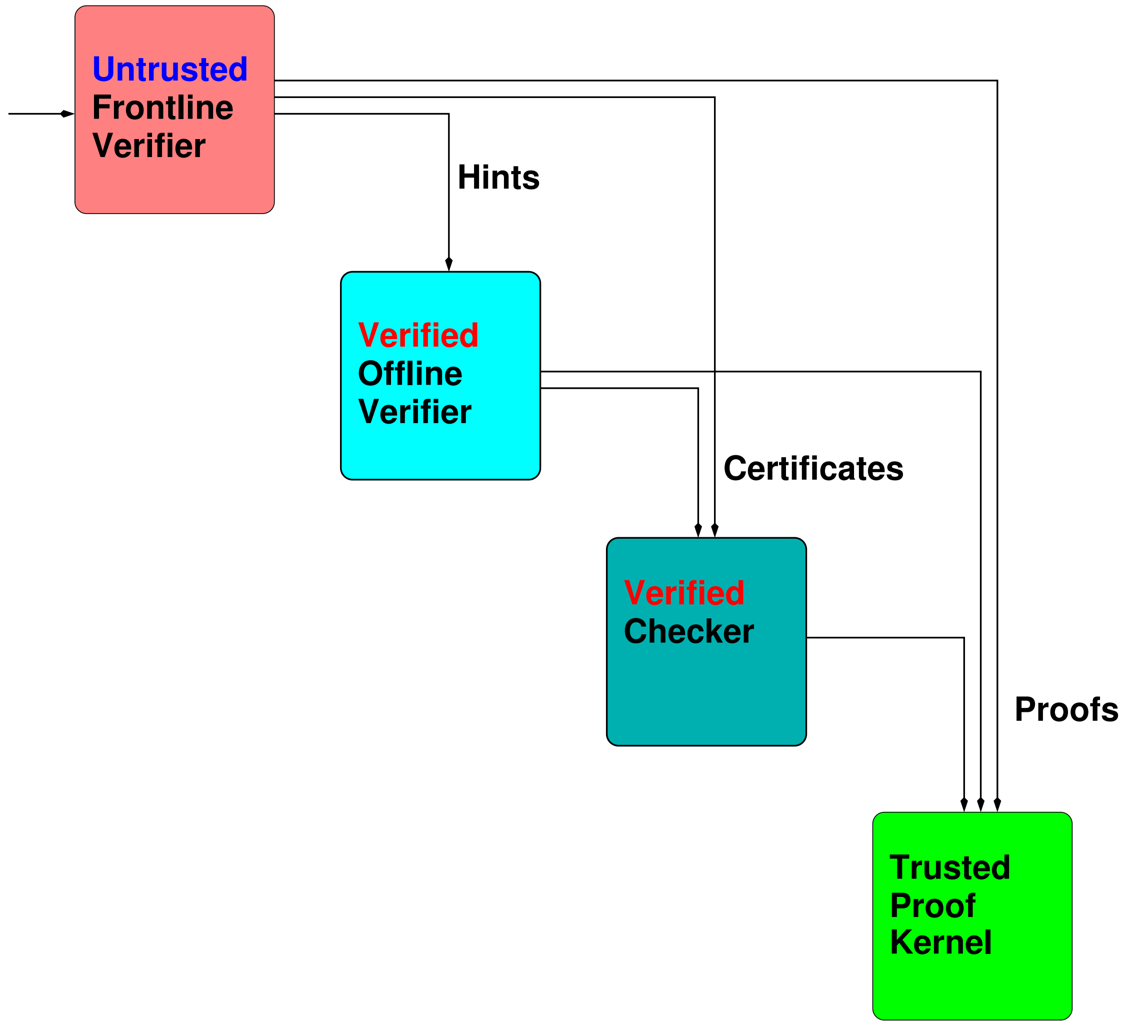
  Concern is that different tools ascribe different semantics
- Increasing issue as specialized tools outstrip monolithic ones
  - Need integrating frameworks such as a tool bus
- Most significant incompleteness is generally the gap between the most detailed model and the real thing
  - Algorithms vs. code, libraries, OS calls

  That's one reason why we still need testing
  - Driven from the formal specification
  - Cf. penetration tests for security: probe the assumptions
- Concerns about incompletness need to be managed

Unsoundness In the Verification System

- All verification systems have had soundness bugs
- But none have been exploited to prove a false theorem
- Many efforts to guarantee soundness are costly
  - e.g., reduction to elementary steps, proof objects
  - What does soundness matter if you cannot do the proof?
- A better approach is KOT: the Kernel Of Truth (Shankar)
  - A ladder of increasingly powerful verified checkers
  - Untrusted prover leaves a trail, blessed by verified checker
  - More powerful checkers guaranteed by one-time check of its verification by the one below
  - The more powerful the verified checker, the more economical the trail can be (little more than hints)
- So concern about unsoundness can be reduced

KOT: A Ladder of Verified Checkers



Shankar and Marc Vaucher have verified a modern SAT solver that is executable (modulo lacunae in the PVS evaluator)

Application

- Suppose our goal is p_{np} of 10^{-4}
- Bulk of this "budget" should be divided between incorrect formalization and incompleteness of the formal analysis, with small fraction allocated to unsoundness of verification system
- Through sufficiently careful and comprehensive formal challenges, it is plausible an assessor can assign a subjective posterior probability of imperfection on the order of 10^{-4} to the formal statements on which a formal verification depends
- Through testing and other scrutiny, a similar figure can be assigned to the probability of imperfection due to discontinuities and incompleteness in the formal analysis
- By use of a verification system with a trusted or verified kernel, or trusted, verified, or diverse checkers, assessor can assign probability of 10^{-5} or smaller that the theorem prover incorrectly verified the theorems that attest to perfection

Discussion

- Probability of perfection is a radical and valuable idea
- Provides the bridge between correctness-based verification activities and probabilistic claims needed at the system level
- Relieves formal verification, and its tools, of the burden of absolute perfection
- But perfection is a strong claim, is $pnp < 10^{-4}$ credible?
    - Why 10^{-4} and not 10^{-3} or 10^{-5}?
    - We need to develop a basis for numerical estimates
    - But if you believe my analysis, historical record suggests DO-178B Level A does justify very strong estimates

The End

- Safety-critical systems are among the most interesting topics in computer science
- Raise interesting challenges in design
- And in assurance and certification
  - Correctness vs. reliability
  - Formalization vs. argument
- These provide intellectual and practical challenges of great interest and value