CSR/ENCRESS, Bruges, Belgium, Sept 1995

The Role of Formal Methods
In the Certification of Critical Systems

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Overview

- $\boxed{\text{What goes wrong?}}$

- Why is it so hard to get software right, and so hard to provide assurance?

- What are formal methods and why might they help?

- The role of formal methods in certification of critical systems

- But is it practical? The state of the art in formal methods

General Defect Rates—IBM FSD (Mike Dyer)

- Industry average is 50–60 defects per 1000 lines of code prior to test
- Modern design methods reduce this to 20–50
- Final delivery has 2–4 defects per 1000 lines
- Formal reviews can eliminate 50–70% of errors prior to test
- Verification-based reviews can eliminate 95%
- And yield 0.2 defects per 1000 lines at delivery

Defect Rates—Grand Gulf Power Utility

- Pre-1989, more than 5 defects per 1000 lines
- Post-1990, less than 0.5 defects per 1000 lines
- 10k lines of C code, real-time, distributed
- 52% of costs were V&V (57% of duration)
- 27% of V&V cost was review (so 73% testing)
- 4 defects removed at final test cost 21% of all testing
- 1 had survived from design stage
- Fault-removal at detailed design review is 160 times more cost-effective than at final test

Defect Rates—JPL Data

- On a variety of programs, found 1 major defect every 3 pages of requirements (cf. 1 every 21 pages for code); most problems were omissions

- Most hazardous defects found during system integration of Voyager and Galileo were due to deficiencies in requirements and interface specifications
  - Only 3 of 197 mission-critical defects were programming problems
  - Remainder divide 3:1 between "function" faults and "interface" faults

Defect Rates—JPL Data (ctd)

- Two-thirds of function faults were requirements problems (mainly omissions), the rest were design problems (largely due to inherent technical complexity)

- Almost all interface faults due to poor communication between teams, particularly between system engineers and software designers; half of all interface faults concerned hardware behavior

Defect Rates—Hardware (Kurt Keutzer)

- More than half of all designs have design errors detected after first fabrication
- Zero errors from implementations stages (simulation, model-checking, BDDs, CAD tools etc. are very effective)
- Typically 1 design defect per 8000 lines (of VHDL-level code)
- 1 line of code generates 5 transistors
- Therefore, faults per chip > 1 if transistors $> 40{,}000$
- By 1995, 10^6 random logic (not RAM etc.) transistors per chip
  Pentium has 3.1m, P6 has 5.3m (+15m in L2 cache)
- Need to reduce design faults/transistor at same rate as technology raises transistors/chip

Advice (Hardware—Kurt Keutzer)

- Don't waste time trying to model timing, parasitic capacitance, CMOS latch-up

- Need help at the higher *design* levels—help that will *shape* thinking

- Math no problem: prepared to use Büchi automata if worthwhile

- Need help reasoning about interaction of control and data
  - Pipeline hazards, superscalar dispatch, out-of-order cleanup
  - Memory subsystems with caches and bypasses
  - Protocols with many timeouts/priorities
  - Combination of all the above

Summary of General Experience

- Current techniques for quality assurance are fairly adequate at the later stages of implementation
- Mishaps are generally due to design flaws, multiple failures, unanticipated interactions, not coding errors
- Greatest weaknesses are in V&V for early stages of design
    - Inadequate assurance for these stages
    - In particular, lack of early validation
    - Enormous *cost* of late fault detection and removal
    - Potential seriousness of consequences of undetected faults
- And in V&V for systems with complex interactions (e.g., concurrent, fault-tolerant, reactive, real-time systems)

Avionics Applications

- Paradigmatic safety-critical systems
- Digital Flight Control Systems (DFCS): computer systems that interpret pilot commands, sample sensors, evaluate control laws, and command the actuators
- Used on all modern warplanes, increasingly on commercial airplanes (Airbus A320, Boeing 777)
- Aircraft certification requires assurance that catastrophic failures will be "extremely improbable" (FAA AC 25.1309A)
  - Not expected to occur in lifetime of fleet
  - "As an aid to engineering judgment" probability of catastrophic failure should be less than 10^{-9}/hour over a ten hour flight
- Many DFCS failures would be catastrophic

Avionics Applications: Concerns

- Software for aircraft is developed to extremely rigorous standards, and subjected to massive testing
- Evidence is that design and coding bugs in sequential components are eliminated effectively
- Concern centers on
  - Requirements
    (e.g., JPL data for Voyager and Galileo spacecraft: only 3 of 197 mission-critical defects were programming problems; IBM data for Space Shuttle: 400 "user notes" documenting requirements anomalies—cf. 6 potentially life-threatening implementation defects known to have flown)
  - The intrinsically hard problems
    (e.g., coordination of distributed computations, timing, synchronization, fault tolerance)

Redundancy Management in DFCS

- The intrinsically hard problems mainly arise in redundancy management
- Hardware components of DFCS are far less reliable than the system requirement
- Hence, need redundancy and voting among sensors, actuators, and computing channels—redundancy management is responsible for all this
- Impact of redundancy management is considerable
  - Typically more than 50% of code
  - Intrinsically difficult problems: coordination of distributed systems in presence of faults
  - Can become *primary* source of *un*reliability, and source of single point failure

Historical Experience With DFCS

- Advanced Fighter Technology Integration (AFTI) F16

- Digital Flight Control System (DFCS) to investigate "decoupled" control modes

- Triplex DFCS to provide two-fail operative design

- Analog backup

- Digital computers not synchronized

  "General Dynamics believed synchronization would introduce a single-point failure caused by EMI and lightning effects"

AFTI F16 Flight Test, Flight 36

- Control law problem led to "departure" of three seconds duration

- Sideslip exceeded 20°, normal acceleration exceeded −4g, then +7g, angle of attack went to −10°, then +20°, aircraft rolled 360°, vertical tail exceeded design load, failure indications from canard hydraulics, and air data sensor

- Side air data probe blanked by canard at high AOA

- Wide threshold passed error, different channels took different paths through control laws

- Analysis showed this would cause complete failure of DFCS and reversion to analog backup for several areas of flight envelope

AFTI F16 Flight Test, Flight 44

- Asynchronous operation, skew, and sensor noise led each channel to declare the others failed
- Analog backup not selected (simultaneous failure of two channels not anticipated)
- Aircraft flown home on a single digital channel (not designed for this)
- No hardware failures had occurred

Other AFTI F16 Flight Tests

- Repeated channel failure indication in flight was traced to roll-axis software switch
- Sensor noise and asynchronous operation caused one channel to take a different path through the control laws
- Decided to vote the software switch
- Extensive simulation and testing performed
- Next flight, same problem still there
- Found that although switch value was voted, the unvoted value was used

X29A Flight Test

- Three sources of air data on X29A: nose and two side probes
- If value from nose is within threshold of both side probes, use nose probe value
- Threshold is large due to position errors in certain flight modes
- If nose probe failed to zero at low speed it would still be within threshold of correct readings
- Aircraft would become unstable and “depart”
- Caught in simulation but 162 flights had been at risk

HiMAT Flight Test

- Single failure in redundant uplink hardware
- Software detected this, and continued operation
- But would not allow the landing skids to be deployed
- Aircraft landed with skid retracted, sustained some damage
- Traced to timing change in the software that had undergone extensive testing, but not in presence of failures

Other Examples

- AFTI F16 flight test 15 (interaction with SMS)
- C17 channel dropoffs in flight tests (voter disagreements)
- X31 channel dropoffs in flight tests (air data disagreements due to divide by zero in reused subsystem)
- Failure to launch STS-1 (synchronization of backup computer)
- Loss of data from Voyager at Jupiter (SEUs caused loss of clock synchronization)
- Patriot failure at Dharan (clock drift)
- Saturation of AT&T network on 15 January 1990 (fault in redundancy management)

Analysis: Dale Mackall, NASA Engineer
AFTI F16 Flight Test

- Nearly all failure indications were not due to actual hardware failures, but to design oversights concerning asynchronous computer operation

- Failures due to lack of understanding of interactions among
  - Air data system
  - Redundancy management software
  - Flight control laws (decision points, thumps, ramp-in/out)

An Interpretation of Mackall's Analysis

- Design flaws were due to difficulty of anticipating behavior of distributed, concurrent, real-time execution in the presence of faults
- Space of possible behaviors is vast
- Testing and simulation visit only small fraction of that space, and extrapolation from tested to untested cases is dubious (behavior is not continuous)
- Need a way to predict behavior under *all* circumstances
- (And a rational design to give some structure to the space)

Overview

- What goes wrong?
- $\boxed{\text{Why is it so hard to get software right, and so hard to provide assurance?}}$
- What are formal methods and why might they help?
- The role of formal methods in certification of critical systems
- But is it practical? The state of the art in formal methods

Why is Software so Hard to Get Right? — 1

- All software failures are due to *design faults* (also called systematic faults or generic faults)
- Design faults can occur in any technology
- Usually due to unanticipated interactions among components or between system and its environment
- More likely as number and complexity of interactions increases
- Most of the complexity of modern systems is in the software (VSLI has the same characteristics as software)
- It's also where the changes get made

Why is Software so Hard to Get Right? — 2

- To eliminate design faults, we need complete understanding of all the behaviors and interactions of a system
- Cannot handle many behaviors and interactions with only intuition and tests unless there's some way to group them
- With physical systems, continuity provides this grouping
- But digital systems have *discontinuous* behavior
- Have to separately reason about or test millions of discrete states
- Scale of the task exceeds our (unaided) intellectual grasp

Why is it so Hard to Provide Assurance for Software? — 1

- Critical systems require very small failure rates (e.g., 10^{-9} per hour) — this is the "ultra-dependable" region
- Require some *evidence* that this has been achieved
- With physical systems, design faults (are assumed to) have been eliminated and failures are due to physical degradation and random events
- Evidence from previous systems, testing, and analysis allows failure rates to be quantified
- Redundant subsystems and backups have independent failure modes (when properly designed), so multiplicative increase in reliability can be achieved and claims of ultra-dependability can be justified

Quantifying Design Faults

- A design fault is either present or not—so what does it mean to talk about probabilities and reliability in the context of design faults?

- All failures can be treated probabilistically: failure due to design fault is associated with probability of encountering the input (sequence) that activates it

- Measuring reliability therefore requires probing the input space according to some "operational profile"

Why is it so Hard to Provide Assurance for Software? — 2

- With software, experimental determination of ultra-reliability requires testing the actual system
  - No credible carryover from previous "similar" designs
- Direct measurement of failure rates on the order of 10^{-9} requires 114,000 years on test—under optimistic assumptions
- Furthermore, very difficult to reproduce the "operational profile" for very rare events (where catastrophic failure often lurks)
- In practice, hard to justify experimental quantification beyond 10^{-4} or 10^{-5}

Assurance Through Diversity

- Can achieve ultra-dependable failure rates by combining lesser components provided correlated failures are nearly zero
- Works for random faults in hardware
- Unfortunately, experiments in low reliability region indicate that for artifacts having significant design complexity (i.e., software), incidence of correlated failures cannot be ignored
- And theoretical studies show uncorrelated faults can lead to correlated errors (and hence failures)
- Therefore, cannot assume independent failures; have to *measure* correlation
- Correlation needs to be in ultra-dependable region
- Where measurement is infeasible

Assurance Through Process

- Suppose had reasonable prior belief that system was dependable
- Then had a reasonable run of failure-free tests
- Would this provide evidence for ultra-dependability?
- Bayesian analysis shows cannot combine two sources of modest assurance to gain assurance of ultra-dependability: prior belief has to be of ultra-dependability

So Where Are We?

- Because of complexity and discontinuity of behavior, cannot provide experimental evidence of ultra-dependability for software systems
- Benefits (if any) of diversity not quantifiable
- Modest numbers of tests add little to assurance
- Essentially all assurance has to derive from (mainly subjective) factors concerning development processes
- In other words, in the ultra-dependable region, cannot provide evidence of *"how well you've done"* (i.e., dependability achieved), instead provide evidence of *"how hard you tried"*
- This is the burden of essentially all certification standards for critical software

Overview

- What goes wrong?
- Why is it so hard to get software right, and so hard to provide assurance?
- $\boxed{\text{What are formal methods and why might they help?}}$
- The role of formal methods in certification of critical systems
- But is it practical? The state of the art in formal methods

Formal Methods (First Approximation)

- Just as some properties of a bridge can be modeled mathematically

- So some properties of requirements, designs, programs, circuits for software and hardware can be modeled mathematically

- *Descriptive formal methods*: mathematical concepts are good ways of thinking about these systems
  - Formal methods contribute useful mental frameworks, notations, and systematic methods to the design, documentation, and analysis of computer systems

- *Analytic formal methods*: you can *calculate* with these mathematical models and predict the properties of a system

Contribution to Certification (First Approximation)

- Descriptive formal methods may be considered "best practice" and can buttress the case for "how hard you tried"

- Analytic formal methods can be used to *calculate* that a *model* of some aspects of a software system has certain properties

- Thus some certification activities that are traditionally performed by *reviews*
  - Processes that depend on human judgment and consensus

  can be replaced or supplemented by *analyses*
  - Processes that can be repeated and checked by others, and potentially so by machine

  (Terminology from RTCA DO-178B/EUROCAE ED-12B)

Representing—and Reasoning about—Designs

- Basic problem with software is huge number of behaviors and interactions, and no way to group them together by looking at the input/output behavior

- Of course not! You have to look at the design!

- The structure and grouping of similar behaviors is represented in the design and in the understanding *why* the design works

- Traditionally, the understanding is informal and is examined through reviews

- Need a way to represent the design so that we can reason about it scientifically, and thereby predict its overall behavior

Mathematical Modeling

- One way to predict behavior of a system is to construct a mathematical model and *calculate* it

- Model must be reasonably accurate

- And calculation must be performed without error

- For continuous systems, use well-developed mathematical theories (e.g., Navier-Stokes equations for aerodynamics)

- For computer systems, must use discrete mathematics (logic, set theory) and build our own theories

- And proofs of theorems take the place of numerical calculation

Applied Mathematics and Engineering

- Established engineering disciplines use applied mathematics
  - As a *notation* for describing systems
  - As an analytical tool for calculating and *predicting* the behavior of systems
- Computers can provide speed and accuracy for the calculations

Applied Mathematics and Software Engineering

- The applied mathematics of software is *formal logic*
- Formal Logic can provide
  - A notation for describing software designs—*formal specification*
  - A calculus for analyzing and predicting the behavior of systems—*formal verification*
- Computers can provide speed and accuracy for the calculations

Formal Methods

Formal Specification: Use of notations derived from formal logic to describe

- *Assumptions* about the world in which a system will operate
- *Requirements* that the system is to achieve
- A *design* to accomplish those requirements

Formal Verification: Use of methods from formal logic to

- *Analyze* specifications for certain forms of consistency, and completeness
- *Validate* specifications by posing challenges
- *Prove* that the design will satisfy the requirements, given the assumptions
- *Prove* that a more detailed design *implements* a more abstract one

What Makes a Method Formal?

- A *formal* method is equipped with some *rules of deduction*
- These make it possible to *calculate* whether certain conclusions follow from certain premises
- The calculation is called a *proof*
- Because it is a calculation, it can be checked by others
- Or by a machine
- In some cases, can be completely automated (model checking)
- In all cases, formal methods allow *reviews* to be replaced or supplemented by *analysis*

Formal Deduction

- Formal logic supports a form of reasoning that does not rely on the subjective interpretation of the symbols used
  - All $\diamond$s are $\sharp$
  - This $\odot$ is a $\diamond$
  - Therefore, this $\odot$ is $\sharp$

  This reasoning is sound, no matter what the symbols mean

- Logic provides "templates" (logical axioms and rules of deduction) for deducing conclusions from premises

- We provide the subject matter (terms) and assumptions (nonlogical axioms)

- Because they do not depend on intuition, formal proofs can be developed or checked by computer

How Do You Mechanize Formal Methods?

Build models that permit symbolic evaluation and explore the state space (symbolic values capture whole classes of behaviors)

Partial Exploration

Backwards from hazardous states

Forwards from interesting classes of states (FMECA)

Complete Exploration

Brute force (special types of models)

- State exploration: Murϕ, SPIN
- Symbolic model checking: SMV
- Language containment: COSPAN

Heuristics and/or interactive human guidance

- Theorem proving: PVS, Nqthm, HOL, etc.

Claims for Formal Methods

1. Enable faults (of assumptions, requirements, design) to be detected *earlier* than otherwise
   - Due to greater precision and explicitness early in the lifecycle
2. Enable faults to be detected with greater *certainty* than otherwise
   - Because they replace reviews (consensus) by analyses (calculations)
   - And can provide *total* coverage (of selected, modeled properties)
3. *Guarantee* absence of specified faults (subject to accuracy of modeling employed)
   - Because the calculations (proofs) can be checked mechanically (by a theorem prover)

Other Benefits of Formal Specifications

- Force early attention to (relevant) detail
- Allow precision without irrelevant detail
- Support notions of completeness
- Allow accurate interface descriptions
- Support unambiguous communication

Other Benefits of Formal Verification

Proofs are a tool of discovery, not just of certification

- *Discovering*, *pruning* and *sharpening* assumptions
- Challenging requirements to *validate* them
- *Debugging* designs
- *Deriving* test cases
- *Exploring* alternative designs
  - Like CFD, formal methods can allow the design space to be explored more completely and more cheaply than prototyping
- Providing assurance for *negative* properties
- *Adjusting* designs to changed requirements

What can go Wrong?

- Fallibility of the whole paradigm
  - Proof and truth, models and reality
  - Incorrect or incomplete assumptions
  - Improper requirements
  - Inconsistent design specifications
  - Invalid proofs
- Self-inflicted wounds
- Bad economics
- Poor psychology

Proof and Truth

- Logic provides rules of calculation that enable valid conclusions to be deduced from premises
- The calculation is called a *proof*
- Assumptions about the world are made *explicit* in the premises, separated from rules of deduction
- If the premises are true statements about the world
- Then the soundness theorems of logic guarantee that the conclusion is also a true statement about the world
    - But not necessarily the one you wanted

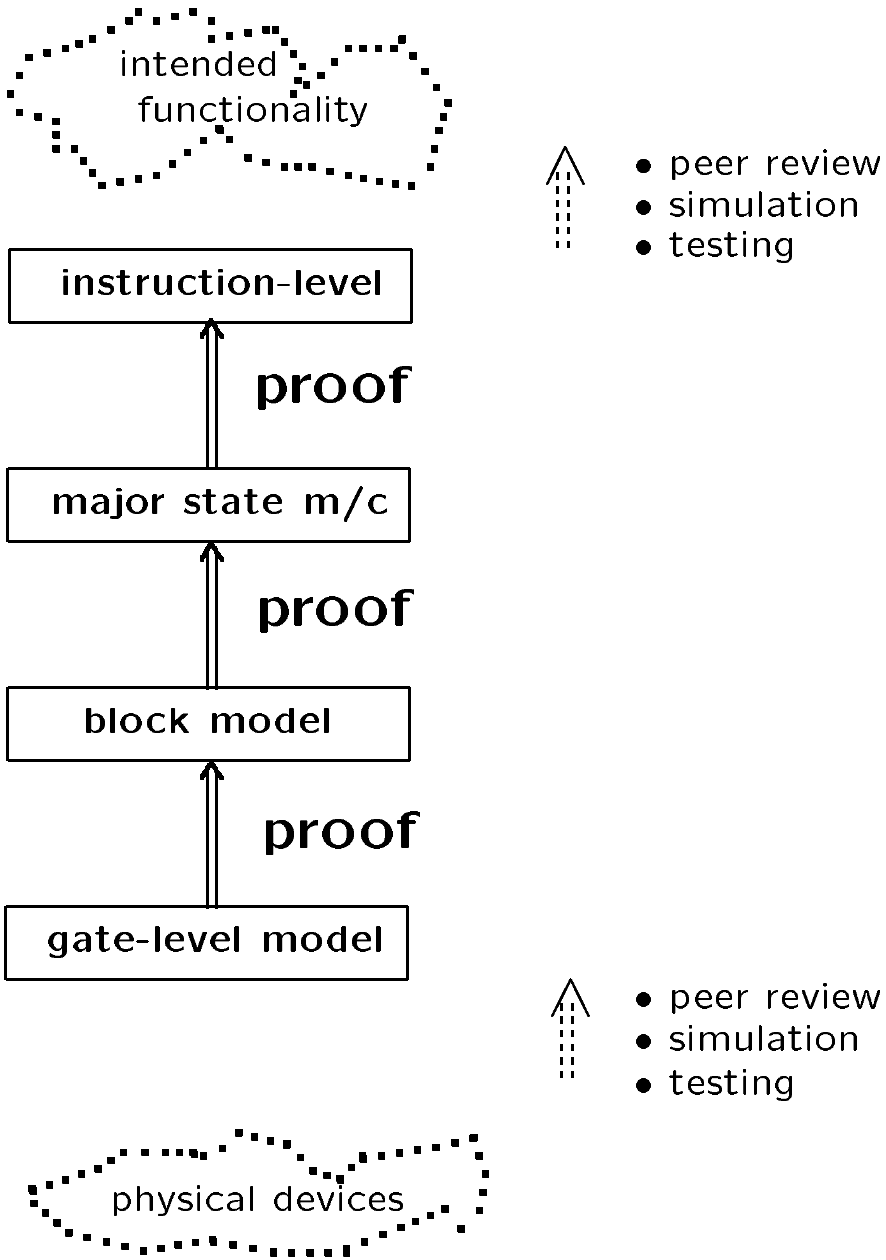
Limitations of Formal Methods

- Correctness is a real-world property
- Formal methods are a modeling activity: they deal with mathematical *models* of reality
- Just like applied math in all other engineering disciplines
- To be useful, we must *validate* that our assumptions are true statements about the real world

  And that our formalized requirements accurately capture the real requirements

  And that our interpretation of the conclusion is accurate
- These validation activities are necessarily informal, empirical and imperfect
- Furthermore, safety isn't the same as correctness

Illustration of Limitations



Incorrect or Incomplete Specification of Assumptions

- Formal methods cannot solve this problem, but
- Formal verification forces a complete enumeration of assumptions
  - Documents them precisely and makes them available for scrutiny
  - Helps in pruning them, and in sharpening their statements
  - Identifies those properties that must be validated in the real world
  - Provides requirement specifications for lower-level implementations

Improper Specification of Requirements

- Formal methods cannot solve this problem, but
- Formal specification provides a precise description for scrutiny
- Formal verification can allow *testing* of requirements
  - Suggests putative theorems (challenges)
  - "If I've got it right, the following ought to follow"
  - "If I've got it right, the following ought to be impossible"

Inconsistent Design Specifications

- Inconsistent specifications cannot be implemented
  - But are easy to create by accident
  - And can be used to prove anything
- Constructive specifications are always consistent
  - But can be overly prescriptive
  - And of limited applicability
- Need to enrich the class of specifications that provide *conservative extension*
- *And* provide ability to exhibit a model (non-prescriptive "implementation") that demonstrates consistency of axiomatic specifications

Validation of Specifications

- How do we gain confidence that our specifications are right?

- Internal consistency (means something)
  - Strong typechecking
  - Definitional principle
  - Exhibition of models
  - Modules and parameters

- External fidelity (means what you intended)
  - Review
  - Analysis ("challenges")
  - Systematic (but imperfect) completeness checks

Invalid Proofs

- Formal verification allows proofs to be checked mechanically
- But doesn't that mean the tools need to be certified?
- Trouble is, certifiable proof checkers many orders of magnitude slower than those that can deal with real problems
  - It's no use having a certified proof checker if it doesn't have the power to tackle theorems of real interest
  - And the focus on provers is too narrow: it can blow at *any* seam—version management, parsing, typechecking
- In reality, trustworthiness of provers is not a weak link
  - Redo verification using a second system if worried
  - Final proof should be carefully argued *informal* proof (derived from formal one) for *human* review and certification

Self-Inflicted Wounds: "The Fangs of the Viper"

- British (Government developed) microprocessor for safety critical applications
- Formal methods (Level 1,2) used extensively in its design, some Level 3 verification
- Described as "the first commercially available microprocessor with a formal specification and a proof that the chip conforms to it"
- Report commissioned from CLI described it as "not formally verified" (i.e., not Level 3 from top to bottom)
- Licensee sued UK Government; High Court asked to decide "what is a proof?"
- Overblown claims + narrow-minded criticism
  $\leadsto$ good work thrown away

Bad Economics

- Need to choose carefully where and how to apply formal methods
- Apply with degree of rigor appropriate to the application and in *conjunction* with other methods
- Use effective tools
- But need *methods* as well as tools
- Apply where other methods are absent, ineffective, expensive
- Apply selectively
  - To abstraction of whole system
  - To selected stages of the lifecycle
  - To critical *components*
  - To critical *properties*

Bad Psychology (for both Engineers and Managers)

- Poorly chosen formal methods can be psychologically unsatisfactory
  - Spend effort wrestling with difficulties of formalism rather than substance of the problem
  - Can't *do* anything with unmechanized specifications
- Formal Methods need careful introduction and thoughtful management
  - Front-loads development costs
  - Mismatched to skills of current workforce
- Must not oversell

Overview

- What goes wrong?
- Why is it so hard to get software right, and so hard to provide assurance?
- What are formal methods and why might they help?
- $\boxed{\text{The role of formal methods in certification of critical systems}}$
- But is it practical? The state of the art in formal methods

DO-178B (ED-12B)

- "Software Considerations in Airborne Systems and Equipment Certification," December 1992
- Major revision to DO-178A
- Produced by RTCA (Requirements and Technical Concepts for Aviation) and EUROCAE WG-12
- Industry-accepted guidelines for meeting certification requirements
- Based on very formalized software development process; extensive documentation, reviews and analyses of the products of each lifecycle process
- Formal methods included among "alternative methods"

DO-178B and Formal Methods

- DO-178B allows formal methods as "alternative means of compliance"
- "The goal of applying formal methods is to prevent and eliminate specification, design and implementation errors, throughout software development"
- "Formal methods are complementary to testing. . . can be used to show that anomalous behavior will not occur for inputs that are out of range or unlikely to occur" (?)
- "Application of formal methods can start and stop with (any) consecutive levels in the design structure"

DO-178B and Formal Methods (ctd)

- "Requirements suitable for the application of formal methods are those that
  - "Are safety-related"
  - "Can be defined by discrete mathematics"
  - "Involve complex behavior, such as, concurrency, distributed processing, redundancy management, and synchronization"

- "Formal methods include these increasingly rigorous levels:
  - "Formal specification with no proofs"
  - "Formal specifications with manual proofs"
  - "Formal specifications with automatically checked or generated proofs"

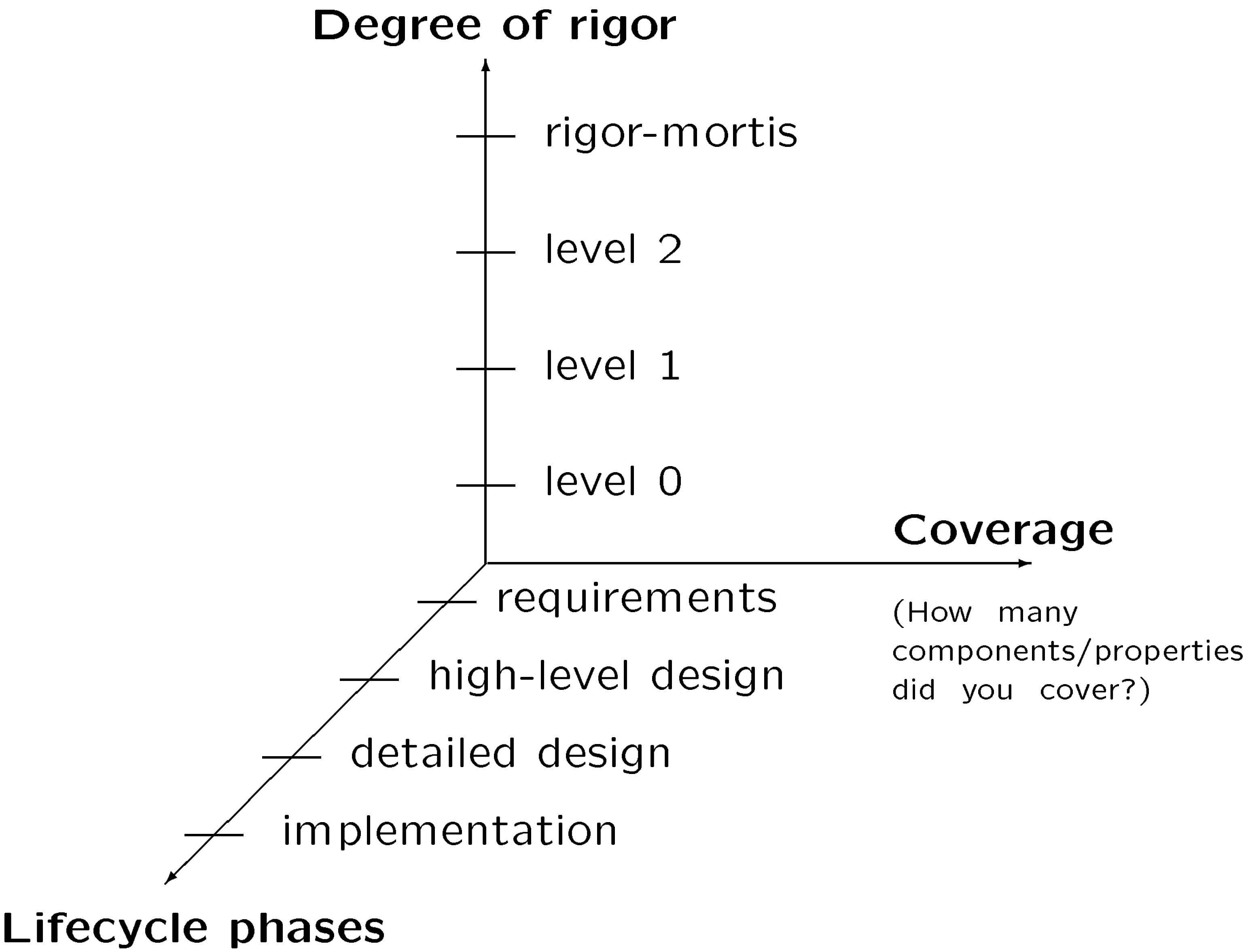
Where, How, and Why to Apply Formal Methods

- Need not (and probably cannot) apply formal methods everywhere
    - What is the basis for selection?

- There are many formal methods, and many tools to support them
    - Again, what is the basis for selection?

- What good reasons are there for using formal methods?

Extent of Application of Formal Methods



Selected Application of Formal Methods

- Cannot achieve total coverage; there will always be gaps that must be covered informally
- Must choose where and how "big" those gaps should be
- There are four axes of selectivity:
  - Lifecycle phases
  - Components
  - Properties
  - Degree of rigor
- Guide: use where traditional methods are absent, ineffective, expensive
  - Usually the hardest problems (where many interactions)
  - Benefits come from calculation (automation essential)

Lifecycle Phases

Pros and cons of applying formal methods in early and late lifecycle phases

- Late lifecycle

  Pro: That's what runs

  Con: Size of description is large; must often leave purely functional world of ordinary logic (i.e., need VCGs, Hoare sentences); *traditional methods are very effective*

- Early lifecycle

  Pro: That's where the serious errors are; that's where the concern is; few other rigorous techniques available

  Con: front-loads development time and cost

You *do* have to choose: going from requirements down to gates is usually inappropriate

Selected System Components and Properties

- Most disciplines concerned with critical systems assign criticality levels to software components based on hazard analysis and potential consequences of failure (must consider malfunction and unintended function as well as loss of function)
  - Higher criticality levels seem good place to focus use of formal methods
- May not need to verify all functional properties
  - May be enough to guarantee a "safe" action, not necessarily an optimal one
  - In other cases, may want to verify properties other than function (e.g., security, or fault containment)
  - Often, absence of specific malfunctions is most important property

Selected System Components and Properties (DO-178B)

- DO-178B assigns software criticality levels A through E according to maximum failure severity (determined through hazard analysis)

- FAA failure effect severities: catastrophic (10^{-9} per hour), hazardous/severe-major, major, minor, no effect

- Must consider malfunction and unintended function as well as loss of function

- May be able to lower criticality levels of large bodies of software by providing components that limit consequences of failure
  - Partitioning (fault containment)
  - Monitoring
  - Centralized redundancy management (not in DO-178B)

Degrees of Rigor in Formal Methods

- Can apply formal methods at many levels of rigor

  0. No use of formal methods
  1. Use the *ideas* of formal methods, but ad-hoc notation, proofs based on informal argument, tools are pencil and eraser (the way conventional mathematics is done)
  2. Formalize and maybe mechanize specification language and methodology, retain pencil and eraser for proofs
  3. Full mechanization with automated theorem proving or checking

- Roughly speaking, Europeans have focussed on Level 1 ($\leadsto$ 2), US on Level 3

- Higher levels are not necessarily better, but...

Degrees of Rigor in Formal Methods (ctd)

- Experience with Level 3 methods shows that most unchecked specifications are flawed (often seriously, sometimes they are meaningless)

- Writing correct formal specifications is no easier than writing correct programs (for most people it's harder)

- May be difficult to believe this, until you've experienced the personal shock of finding errors in your own specifications

- Tools can help you find errors early and cheaply

Finding Errors In Formal Specifications

Tools for formal methods should provide a collection of graduated techniques to help in the detection of errors

- Syntax: parser
- Formal consistency: typechecker
- Sanity check: direct execution ("animation")
- Deeper sanity check: formal challenges
- Reviews: prettyprinter, cross-reference generator
- Dark corners: state exploration/model checking
- Hazard Analysis/Safety: backwards execution
- Correctness: formal verification

What Can Formal Methods Add to Assurance?

- Documentation
  - Formal specifications can provide precise, unambiguous documentation of interfaces (including assumptions) and functions
  - And can provide traceability of design decisions
  - And support reuse
- Design discipline
  - Formal methods can help you think straight
  - And discourage over-complex solutions
  - And facilitate communication and walkthroughs
- These are good practice, but I wouldn't give certification credit for them

What Can Formal Methods Add to Assurance? (ctd)

- For certification credit, need some *objective* evidence of superior assurance

- For process-based certification, assurance hinges on reviews and analyses:
  - "One distinction between reviews and analyses is that analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness." (Draft version said "reviews provide a group consensus")

- My opinion is that formal methods contribute most to assurance when they supplement or replace reviews with analyses

- That means use of formal "calculation"

When to Use Formal Methods?

- When existing methods are inadequate, expensive, absent
- Existing methods are adequate for *most* late-lifecycle processes
  - Very disciplined development and coding methods are used
  - Fagan-style inspections, systematic unit-testing methods, standard, reused components
    (hardware below RTL level is similar)
  - No serious failures in flight software traced to coding bug
  - Formal methods are at their most expensive
    (though methods are very well understood)
  - But there is a case for applying formal methods to especially novel, difficult and/or crucial implementations
    (concurrent, distributed, single-points of failure)

When to Use Formal Methods? (ctd)

- Existing methods are barely adequate for *many* early-lifecycle processes
  - Accidents are generally the result of multiple failures, unanticipated interactions, improperly understood requirements
  - Few other systematic or analytic methods are available early in the lifecycle
  - Formal methods are at their most effective and economical (though developing effective approaches to specific problems is often a research problem)
  - "Lite" methods can go a long way, can be completely automated, and are acceptable to practitioners (e.g., "Tablewise" checks well-formedness and internal consistency of tables for mode switching logic)

When to Use Formal Methods? (ctd)

- Existing methods largely inadequate for design issues involving concurrency, complex interactions, fault-tolerance
- Requires heavyweight tools, and great skill and experience in their application
- However, these hard and critical issues are usually relatively localized and few in number
- And the tools are at their best in these domains
- Can accomplish a lot with a few highly skilled people

FAA Digital Systems Validation Handbook

- FAA Technical Center, Atlantic City, sponsored a chapter on formal methods for the Handbook (a guide to certifiers)
  `http://www.csl.sri.com/csl-95-1.html`

- Discusses many of these issues in detail

- But is not an introduction to formal methods

- Reasonably nonpartisan

- Should be of use to developers as well as certifiers, and to domains other than aircraft

- A much longer and more technical treatment is
  `http://www.csl.sri.com/csl-93-7.html`

Overview

- What goes wrong?
- Whys is it so hard to get software right, and so hard to provide assurance?
- What are formal methods and why might they help?
- The role of formal methods in certification of critical systems
- $\boxed{\text{But is it practical? The state of the art in formal methods}}$