Formal Methods Introduction and Demo

A Practical Introduction to Formal Methods

John Rushby

`http://www.csl.sri.com/~rushby/slides/fmtutorial.pdf|ps.gz`

Computer Science Laboratory
SRI International
Menlo Park, California, USA

Agenda

- Philosophy
- Technology overview
- Model checking example: group membership
- Theorem proving example: bus guardian window timing
- Relevance to dependability

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 the 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, and that the design is implemented correctly
  - These concerns are handled separately from verification

Formal Methods: Analogy with Engineering Math (ctd.)

- Same idea
- 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: static analysis, model checking, theorem proving, etc.
- Formal calculations cover all modeled behaviors
- If the model is accurate, this provides verification
- If the model is approximate, still good for debugging (aka. refutation)

Comparison with Simulation, Testing etc.

- Simulation also considers a model of the system (designed for execution rather than analysis)
- Testing considers the real thing
- Both differ from formal methods in that they examine only some of the possible behaviors
- For continuous systems, verification by extrapolation from partial tests is valid, but for discrete systems, it is not
- Can make only statistical projections, and it's expensive
  - 114,000 years on test for 10^{-9}

  Limit to evidence provided by testing is about 10^{-4}
- Heavy duty formal methods can consider all behaviors of an accurate model (but they are expensive)

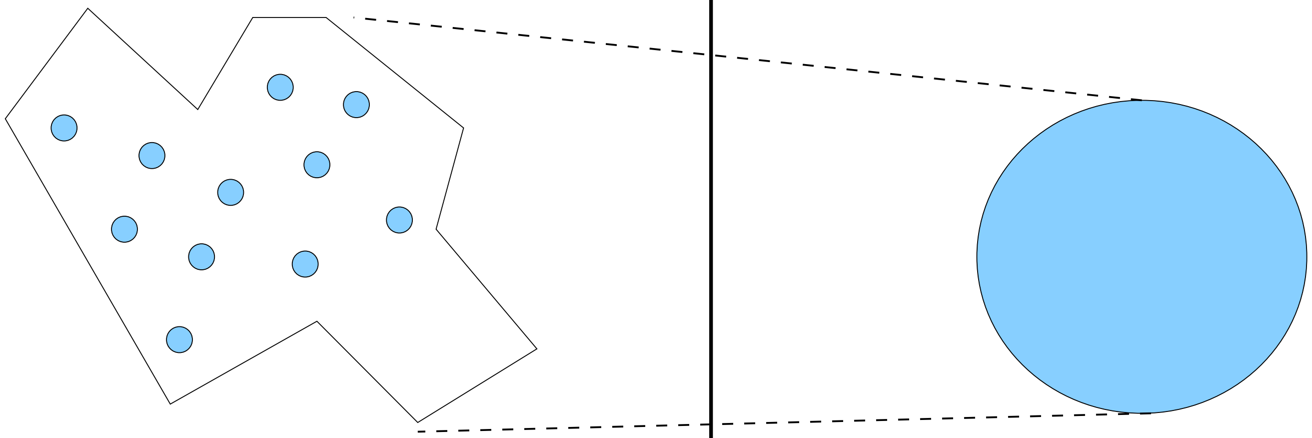
Comparison with Simulation, Testing etc. (ctd)

- Debugging by simulation and testing depend on skill in choosing test cases, luck
- Almost impossible when the environment can introduce huge numbers of different behaviors (fault arrivals, real-time, asynchronous interactions)
- Formal methods consider all behaviors, so certain to find the bugs
  - Provided the model and the properties checked are sufficiently accurate to manifest them

  So depends on skill in modeling, luck
- Experience is that you find more bugs (and more high-value bugs) by exploring all the behaviors of an approximate model than by exploring some of the behaviors of a more accurate one

Formal Methods: In Pictures

Testing/Simulation	Formal Analysis



Real System

Formal Model

- Partial coverage

- Complete coverage (of the modeled system)

  Accurate model: verification

  Approximate model: debugging

Formal Methods Technologies

Static analysis: works directly off programs; can establish weak (but important) properties such as absence of arithmetic overflow, no reference to initialized variables etc. Automatic. Widely used in French aerospace industry (Polyspace, esp. since the loss of Ariane V); mandated for certain UK procurements (Spade/Malpas, esp. since loss of Chinook); best US system (Prefix) bought up by Microsoft

Model checking: usually works off a special modeling language; basically brute force enumeration. Mostly automatic, once the model is built.

Theorem proving: usually works off a special modeling language; usually requires skilled human guidance

Model Checking

- Imagine a program with only a fixed number of data locations (i.e., its state is finite)
- Simulator runs the program and saves snapshots of its state
- At each choice point (e.g., program could do this, environment could do that), choose one arbitrarily; when finished, back up and explore different choice
- Terminate exploration along a given path when you encounter a state you've seen before
- At each state, check that specified properties hold
  - On failure, produce a backtrace or counterexample of the path that got you here
- Done when have explored all the paths from all visited states
- This is state exploration, aka. reachability analysis

State Exploration

- Data structure
  - The set of states already seen
  - States in the set are marked as explored or not
  - The set starts with just the initial state(s), marked as unexplored
- Method

```
repeat
  find an unexplored state in the set
  mark it as explored
  calculate all its successors
    add any not seen before to the set,
      marked as unexplored,
      and check the specified properties hold
until done
```

State Exploration: Scaling

- If the state is n bits, there are 2^n different states and we need $n/8 \times 2^n$ bytes to store them all
- And if there are m interacting components, statespace is exponential in this, too—state explosion problem
- Do not need a very large n, m before size of statespace exceeds all the matter in the universe
- Of course, may have relatively few reachable states
- And may detect a bug and stop after exploring relatively few states
  - Breadth first search finds short counterexamples
- But practical limit to explicit state exploration is around 100-200 state bits

State Exploration: Overcoming the Scaling Problem

- Use hashing: sacrifices completeness
- Exploit symmetry, order insensitivity
- Use symbolic representation of state: e.g., $x < y$ represents an infinite number of explicit states: (0,1), (0,2), ... (1,2), (1,3)...
  - Standard method uses BDDs (binary decision diagrams) and variants
  - Recent method (bounded model checking) uses SAT solvers (propositional satisfiability checkers) to find all counterexamples less than specified length; incomplete, but very effective for debugging
- Downscale the model

State Exploration: Properties

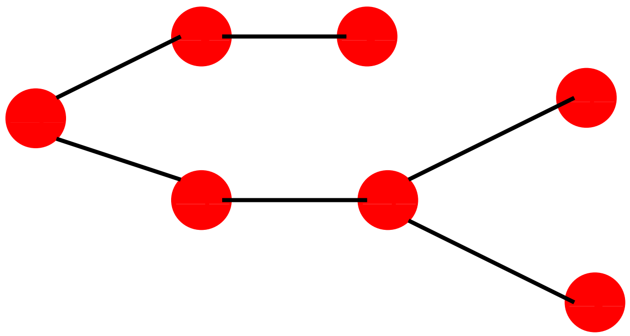
- Checking assertions only allows examination of safety properties
  - Can also check for deadlock
- What about liveness: e.g., "every request is eventually followed by a grant"?
- Need a language to specify these
- Any particular run traces out a linear sequence of states
- Linear Temporal Logic (LTL) is a language for expressing properties of such sequences using always ($\Box$) and eventually ($\Diamond$) modalities

State Exploration: Properties (ctd)

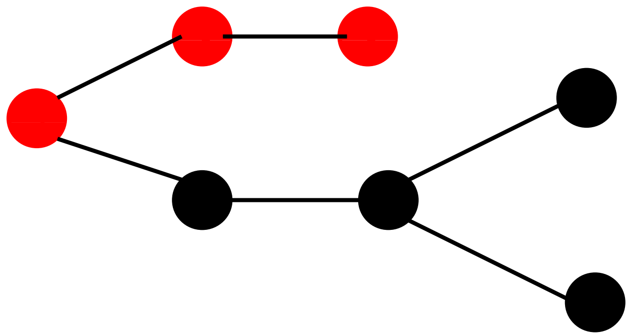
- Evolution of all different behaviors traces out a tree
- Computation Tree Logic (CTL) is a language (a branching time temporal logic) for expressing properties of such trees using four modalities
  - $AG(p)$: p is true everywhere along each path (invariant)
  - $AF(p)$: p is true somewhere along each path
  - $EG(p)$: p is true everywhere along some path
  - $EF(p)$: p is true somewhere along some path

  Also "next" and "until" operators, plus "fair" variants (include only paths where some specified formula is true infinitely often)

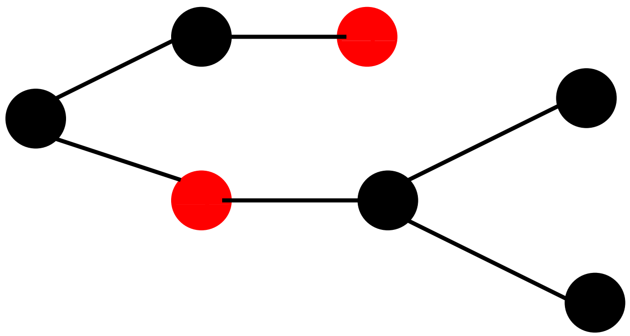
Computation Tree Logic



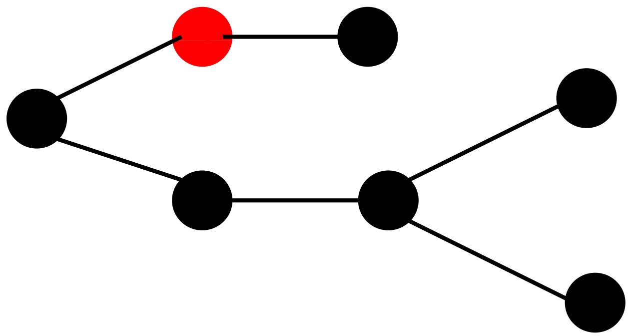
AG



EG



AF



EG

State Exploration: Properties (ctd)

- Can modify explicit-state and symbolic state exploration systems to check for LTL or CTL properties
- Technically, we are checking whether the system description is a (Kripke) model for the property expressed in a temporal logic
- That's why it's called model checking
- Variants described in terms of ω-regular language inclusion, bisimulation, etc.
- Can also do model checking for non finite-state (e.g., timed, hybrid) systems: Kronos, HiTech, Uppaal, LCS

Model Checking Tools

Quite widely used in hardware and protocol verification

- **Spin** (Bell Labs): explicit state, depth-first search, LTL, partial-order reduction
- **SMV** (CMU): symbolic, CTL
- **Murϕ** (Stanford): explicit state, breadth-first search
- **Formal Check** (Bell Labs): commercial version of Cospan; does ω-regular language inclusion
- **Concurrency Workbench** (SUNY SB): does bisimulation in process algebras
- **FDR** (Formal Systems): commercial, does refinement in CSP
- **Aldebaran** (Verimag): explicit state, does static analysis to prune state space

Limitations of Model Checking

- Usually have to downscale the model to make model checking tractable
- Often good at finding bugs, but what if no bugs detected?
- Have we achieved verification, or just got an inadequate model or property?
- Sometimes it's possible to prove that a small model is a property-preserving abstraction of a large, accurate one
- Then not detecting a bug is equivalent to verification
- But in general, have to resort to theorem proving if you want to verify a fully accurate (large or infinite state) model

Theorem Proving

- Using symbolic representations, can establish an infinite number of cases at one go
  - cf. `5*5-3*3 = (5-3)*(5+3)` and $x^2 - y^2 = (x - y)(x + y)$
- Also, we can abstract away irrelevant details
  - E.g., in model checking, simulation, or testing, we have to ascribe a specific incorrect behavior to faulty components
  - With theorem proving, we can just say we know nothing about faulty components
- Use mechanized deduction (theorem proving) to establish that a formula specifying a required property follows from formulas specifying behavior of the system and its environment

However. . .

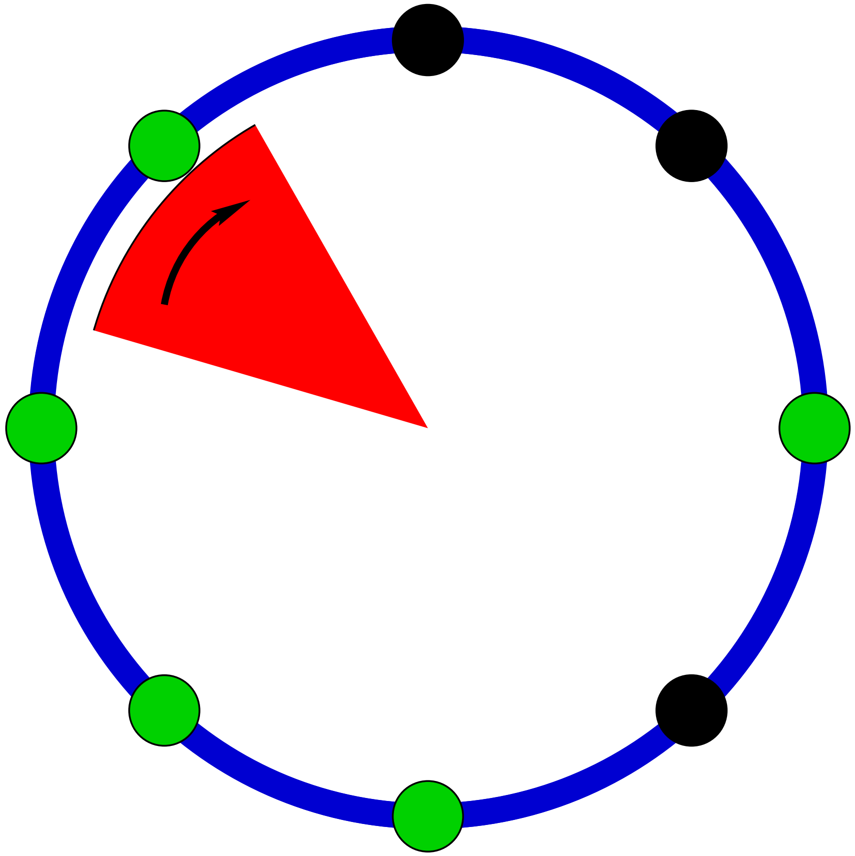
- Theorem proving
  - Is undecidable in general
  - And even decidable problems have much greater computational complexity than mechanizations of continuous mathematics
- So full automation is impossible in general
- Must rely on heuristics (guesses) which will sometimes fail
  - Heuristic theorem proving
- Or rely on human guidance
  - Interactive theorem proving
- Or trade off accuracy or completeness of the model for tractability and automation of calculation
  - Model checking, "lite" theorem proving

Model Checking Example: Group Membership

- This example is based on group membership in TTA
- It's a different algorithm, however (simpler, weaker properties, smaller statespace)

Background: The Time-Triggered Architecture (TTA)

Creates a synchronous, TDMA ring on a broadcast bus



Background: The Time-Triggered Protocol (TTP/C)

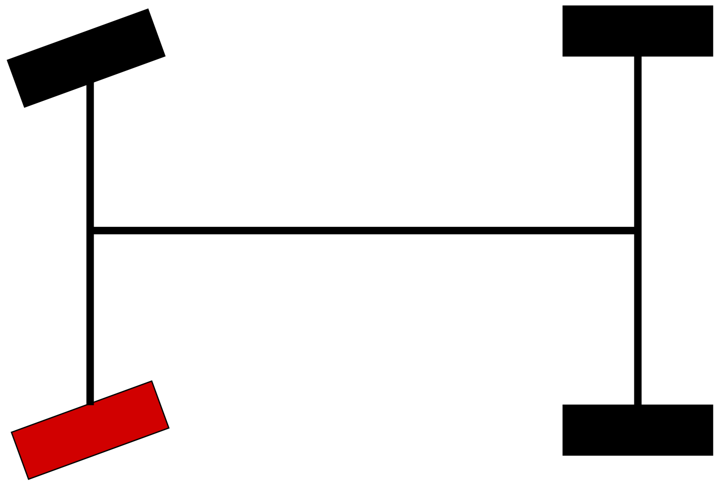
TTP/C is the heart of TTA; it provides several services in a tightly integrated manner

- Clock synchronization
- Time-triggered implementation of synchronous algorithms
- Time Division Multiple Access (TDMA)
  - Deterministic mutual exclusion
  - Data communication
- Group membership (the example considered here)

Assumes (and enforces) fail silence

The Need for Group Membership

- Consider a brake-by-wire application
- Separate computers at each wheel adjust braking force according to inputs from brake pedal, accelerometers, steering angle, wheel-spin sensors etc.
- Suppose one of these computers fails



- The others need to redistribute the braking force
- So must have consistent opinion about who has failed

Requirements For Group Membership

Each processor maintains a membership set

Validity: the membership sets of nonfaulty processors contain all the nonfaulty processors

- And, ideally, nothing else—but this is not possible because it takes some time to diagnose a faulty processor
- So allow at most one faulty processor in the membership

Agreement: all nonfaulty processors have the same membership sets

Self-Diagnosis: faulty processors eventually remove themselves from their own membership sets (and fail silently)

Rejoin: Repaired processors can get back in

Subject to fault hypothesis about possible fault modes, fault arrival rate, and maximum number of faults

Here's a Group membership Algorithm (not TTA)

Broadcaster: Let $b = i \bmod n$.

$$
\begin{array}{llcll}
(a) & b \in \texttt{mem}(b) & \rightarrow & \texttt{mem}(b)' = \texttt{mem}(b), & \texttt{ack}(b)' = \mathit{true} \\
 & \text{otherwise} & \rightarrow & \text{no change.} &
\end{array}
$$

Receiver: Consider an arbitrary processor $p \neq b$. If $b \in \texttt{mem}(p)$ and $p \in \texttt{mem}(p)$, the appropriate guarded command from the following list is executed:

$$
\begin{array}{llcll}
(b) & \texttt{ack}(p) \wedge \text{no msg} & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{false} \\
(c) & \texttt{ack}(p) \wedge \texttt{ack}(b) & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true} \\
(d) & \texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{b\}, & \texttt{ack}(p)' = \mathit{true} \\
(e) & \neg\texttt{ack}(p) \wedge \text{no msg} & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\
(f) & \neg\texttt{ack}(p) \wedge \neg\texttt{ack}(b) & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p), & \texttt{ack}(p)' = \mathit{true} \\
(g) & \neg\texttt{ack}(p) \wedge \texttt{ack}(b) & \rightarrow & \texttt{mem}(p)' = \texttt{mem}(p) - \{p\} & \\
 & \text{otherwise} & \rightarrow & \text{no change.} &
\end{array}
$$

Fault Hypothesis for Membership

Fault modes: very restrictive assumptions

Send Fault

- A faulty processor might not broadcast (in some rounds)
  - Broadcasts are received either by all or none of the nonfaulty processors (weaker in TTA)

Receive Fault

- A faulty processor might not receive some broadcasts

Fault arrivals: faults must be “rare”

- Roughly, no more than one per round
- More precisely, may arrive only when system is “stable”
  - And after a fault it returns to a stable state very quickly

Maximum faults: no restriction (different in TTA)

Model Checking Example: Group Membership

Do it

Theorem Proving Example: Bus Guardian Window Timing

- TTP/C operates according to a global schedule
- All controllers have approximately synchronized clocks
- So each knows when it's its turn to transmit
- And there will be no collisions on the bus
- But a faulty controller might misbehave (or simply lose clock sync) and transmit out of turn—or all the time!—babbling idiot failure mode
- So interpose a bus guardian that fails independently (has its own clock and copy of the schedule)
- Want the bus guardian window to be as narrow as possible
- But still pass all messages from nonfaulty controllers

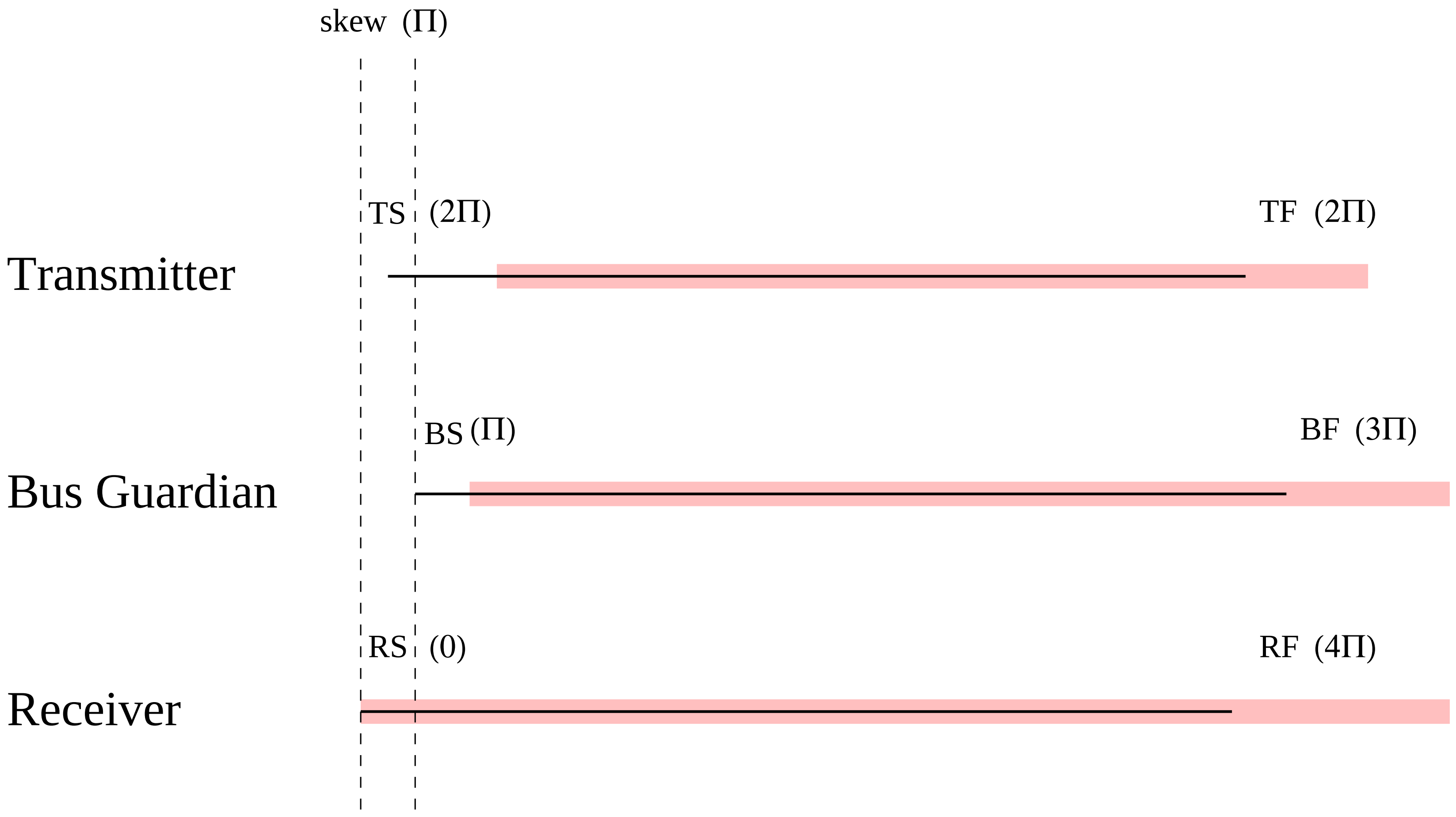
Window Timing: Requirements

- Need to consider windows of three (classes of) components
  - A transmitter
  - Its bus guardian
  - The receivers
- Requirements

  Validity: If any nonfaulty node transmits a message, then all nonfaulty nodes will accept the transmission.

  Agreement: If any nonfaulty node accepts a transmission, then all nonfaulty nodes do
- Given that clocks are synchronized only within some parameter Π

Window Timing: In Pictures



Window Timing: Design Rules

- Each slot has a start time and a maximum duration recorded in the schedule
- 1. Transmission begins $2\,\Pi$ units after the beginning of the slot and should last no longer than the allotted duration.
  2. The bus guardian for a transmitter opens its window Π units after the beginning of the slot and closes it $3\,\Pi$ beyond its allotted duration.
  3. The receive window extends from the beginning of the slot to $4\,\Pi$ beyond its allotted duration.

  These are new: not what is documented in the TTP/C specification
- TTTech has an informal argument that this works

Window Timing: Verification

- Cannot model check this because it's infinite state—relative clock skews can be any three real numbers less than Π—and they can change during transmission
- Need to model it in logic

Theorem Proving Example: Bus Guardian Window Timing

Do it

Summary: Good Points of Model Checking

- Model checking is good for finding bugs
- Particularly in highly concurrent systems
- And in fault-tolerant systems
  - Can "inject" all possible fault scenarios in given class
  - Though it can be hard to specify all possible fault classes
- Counterexamples are extremely helpful
- It's automatic
- And similar to simulation and testing
- Acceptable to engineers and useful in the design loop

Summary: Weak Points of Model Checking

- The models (and properties) have to be simplified to make them tractable to fully automated analysis
- But simplified models may not be fully accurate with respect to the property of interest
  - And that's why they cannot be used for verification
- So when to stop?
  - Lack of refutation is not the same as verification

Summary: Good Points of Theorem Proving

- Can be used for verification
- Can model arbitrary faults (because can remain silent about what faults do)
- And can also leave irrelevant details unstated
- In some domains, can develop highly efficient specialized automatic procedures (e.g., processor pipeline logic—can verify Itanium in fractions of a second)
- But usually requires interactive guidance

Summary: Weak Points of Theorem Proving

- Usually requires interactive human guidance
  - Focuses on proof, and idiosyncrasies of the prover, not on the design
  - Difficult to interpret failure (bug, or bad proof?)
  - Can prove any true property given enough time, skill, patience
  - Or find subtle bugs
  - But often requires too much time, skill, patience

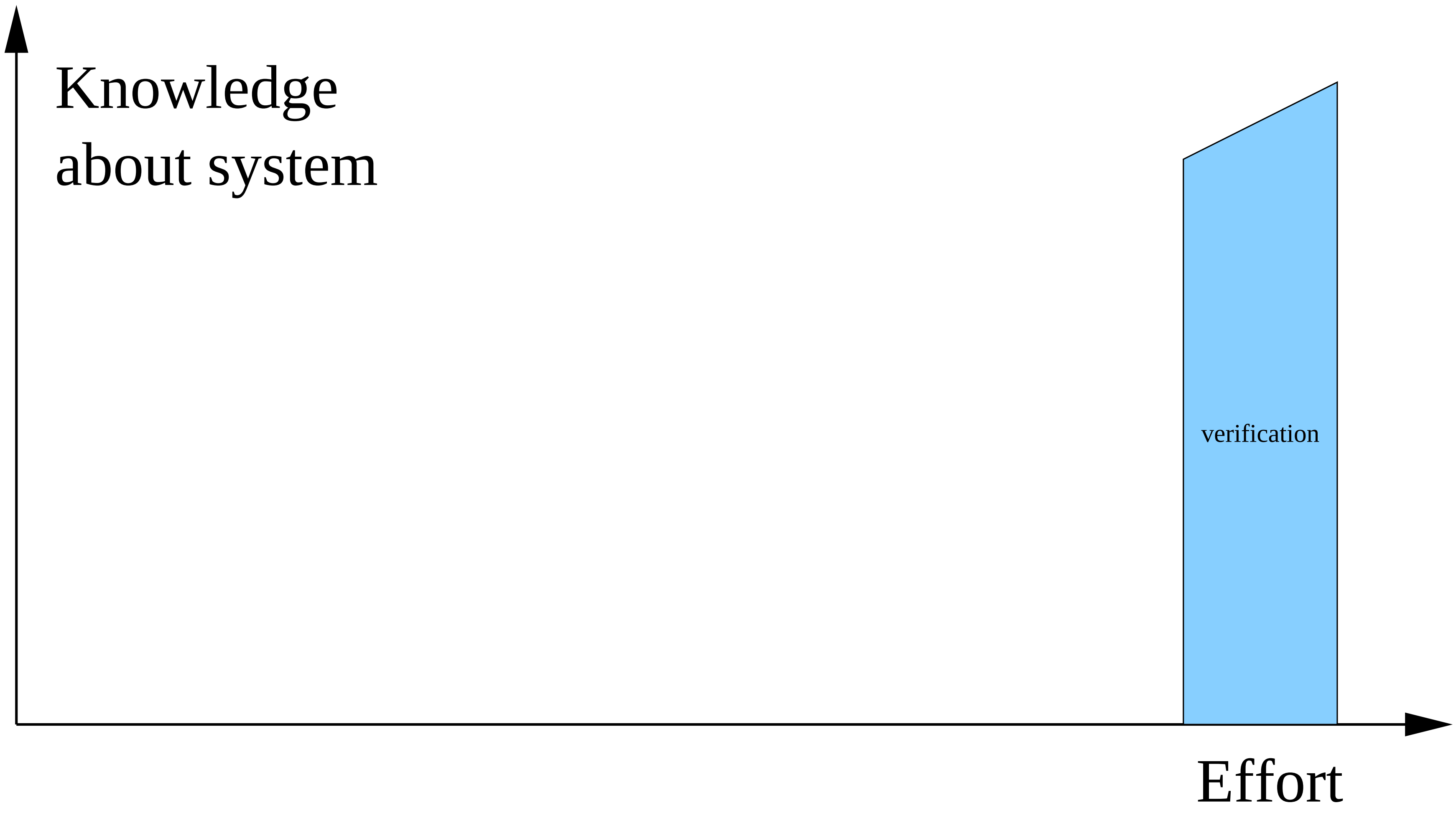
  "Interactive theorem proving is a waste of human talent"
- Also, must strengthen invariants to make them inductive
- And it's all or nothing
- Probably not acceptable to engineers and best for verification of truly critical properties

Inductive Invariants

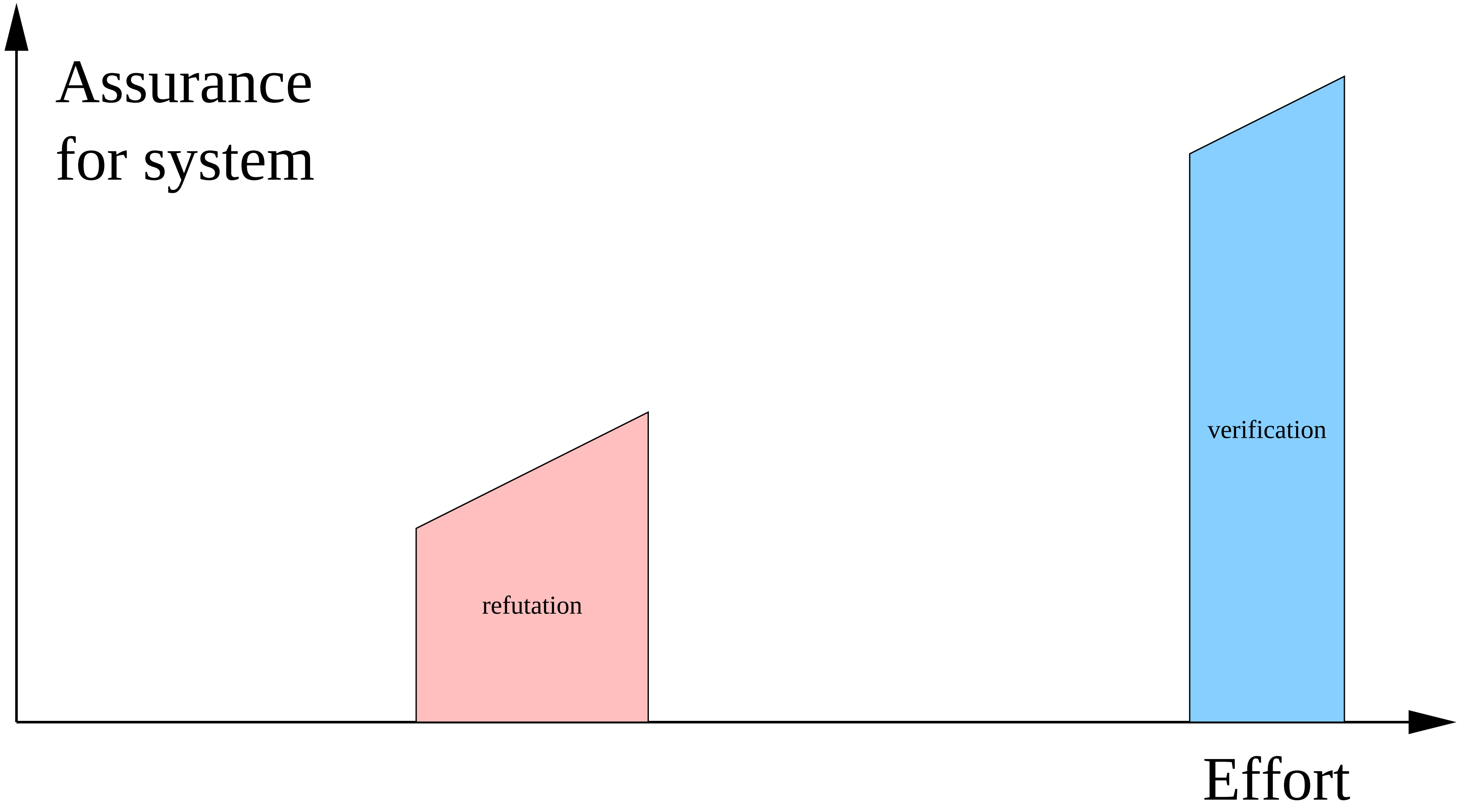
- To establish an invariant or safety property (one true of all reachable states) by theorem proving, we invent another property that implies the one of interest and that is inductive
  - Includes all the initial states
  - Is closed on the transitions

  The reachable states are the smallest set that is inductive
- Trouble is, naturally stated invariants are seldom inductive
  - The second condition is violated
- Postulate a new invariant that excludes the states (so far discovered) that take you outside the desired invariant
- Iterate until success or exasperation
- Bounded retransmission protocol required 57 such iterations

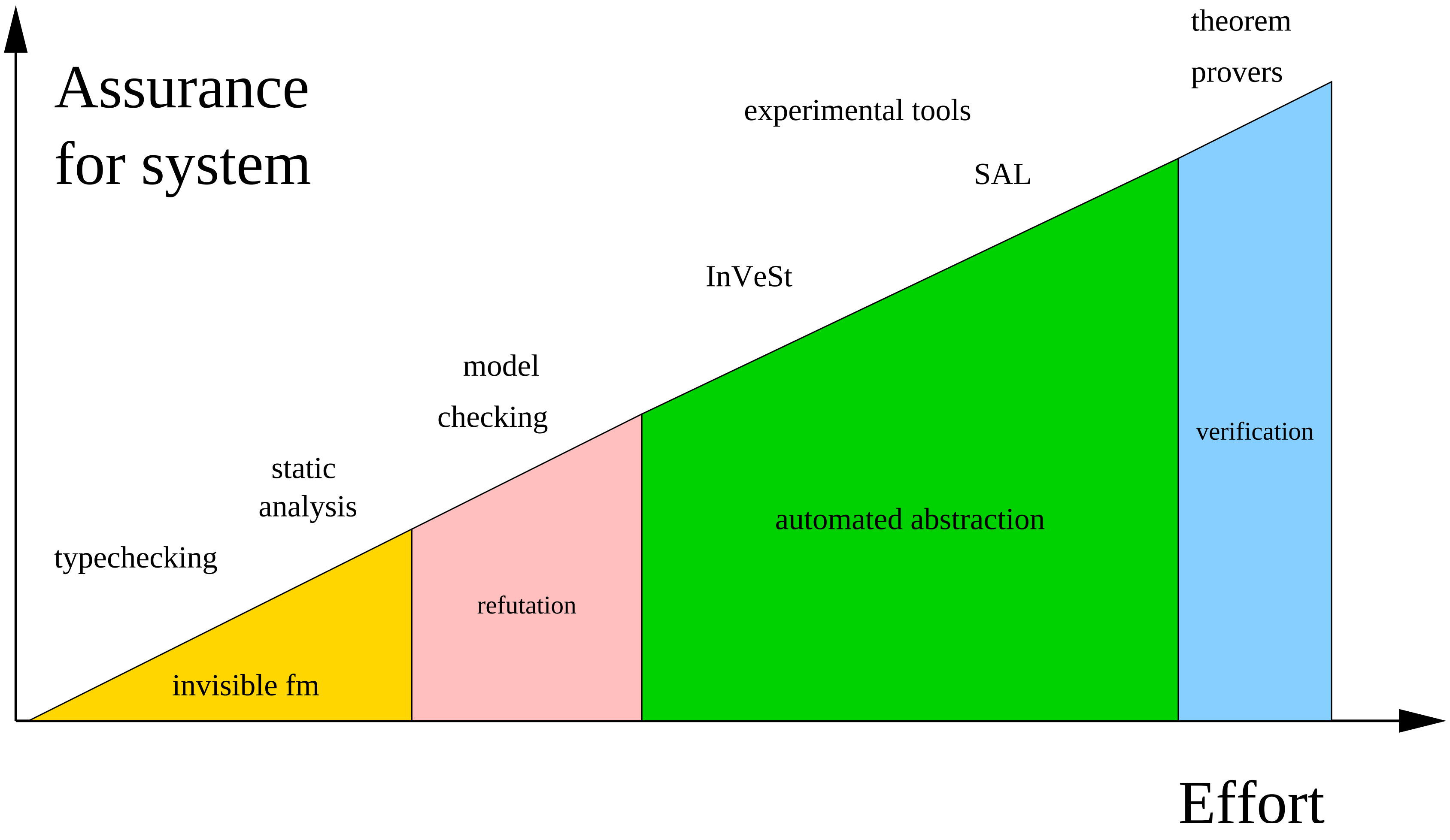
The Wall of Formal Verification



Is Softened by Model Checking



And Will Be Replaced By A Smooth Ramp



Relevance to Dependability: Design and Certification

- Assurance and certification for software is mostly done by controlling, monitoring, and documenting the process used to create it
  - Different industries have different recommended processes (e.g., DO-178B/ED-12B for avionics)
- This is process-based certification
  - Provides no direct evidence about the product

    "We cannot show how well we've done, so we'll show how hard we tried"
- Hugely expensive, and a brake on innovation
- And little evidence that it really assures 10^{-9}

Product-Based Certification For Software

- Product-based certification
  - Is that which concerns properties of the product itself
- For continuous systems, can be done by testing and extrapolation
- But for discrete systems, testing cannot provide evidence beyond 10^{-4}
  - Complete testing is infeasible: 114,000 years for 10^{-9}
  - And extrapolation from incomplete tests is unjustified
- For 10^{-9}, must really consider of all possible behaviors
  - This can be achieved using formal methods to calculate properties of mathematical models of the software
  - Validation of models, and consistency between model and code, by traditional methods (for the time being)

Product-Based Certification For Software (ctd.)

- Using formal calculations, some 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

  Language from DO-178B
- DO178B, DO254 (complex hardware), and most other safety standards allow or encourage such use of formal methods

A Place for Formal Verification

- Replace homespun designs for redundancy management with middleware or architectural frameworks based on rigorous design principles, separation of concerns
  - That are assured to the highest degree
  - Including extensive formal verification

  For example, TTA, DEOS, and successors
- Leave adoption to market forces (fight COTS with COTS)
- Work out the principles of (formal) compositional product-based assurance
  - Based on standardized frameworks
  - So that a market for certified components can develop

A Place for Formal Verification: Longer Term

- Systems are more than collections of components
- At the top, they are mostly specified and designed by systems and control engineers working with models and simulations (e.g., Matlab/Simulink)
  - Need assurance for these models, and for the computational math routines employed (which will happily integrate over a singularity)
  - Formal analysis methods can already help here
- Develop rigorous intellectual path, and formally-assured tool-chain, from these models to component-based implementations running on assured frameworks
- Requires that formal methods become accepted by practicing engineers and integrated in their tools

A Place For Model Checking

- Heavy-duty theorem proving is currently probably too expensive for anything but major infrastructure (like TTA)
- Lite (invisible) theorem proving has great promise but needs to get inside tools (see later)
- But model checking is viable as a superior debugging tool for any design elements that have a lot of concurrent activity—either components (redundancy) or between system and environment (fault arrivals, timing)
- Model checking opens the door to more general use of formal methods
- The modeling languages of some formal methods are excellent notations in their own right, provide superior documentation

Making A Place For Formal Methods: Inside Tools

- You cannot buy a simple Statecharts tool today
  - Can only buy integrated systems that provide desired capability, plus numerous others of varied quality
- Encourage development of standardized "tool bus" driven off a database
  - Like the auto industry is trying to do
- So that vendors can develop value-added components
- And formal methods capabilities can be inserted "invisibly" into traditional tools
  - E.g., Extended static checking for Simulink
- Support the basic research in automated deduction and formal analysis of software that produces the technology

Places for Formal Methods: Longer Term

- Beyond single systems
  - Develop rational architectures for safety-critical multi-agent systems such as air-traffic management (free-flight), automobile convoys, UAV swarms
  - And the formal methods to analyze these
- Human factors (the dominant cause of airplane accidents)
  - Modern cognitive science views the mind as an information processor
  - Can build formal models of cognitive abilities (e.g., mental models) and analyze them in juxtaposition with automation to find "automation surprises"
- Basic research: support for fundamental technologies (again)
  - Formal spec'n, automated deduct'n, test-generation; assurance for comput'n math, simul'n, auto code gen