7th Australian Workshop on Safety Critical Systems and Software, Adelaide, October 2002

Trends in System Safety–US View

`http://www.csl.sri.com/~rushby/slides/scs02.pdf|ps.gz`

John Rushby

Computer Science Laboratory

SRI International

Menlo Park CA USA

Caveats

- I'm a formal methods researcher
  - Not a practitioner
  - Not a safety guy

  By the way, in the USA, formal methods does not mean Z

- My connections to practice and to safety are through people who sponsor or use our technology, mostly in aerospace
  - NASA
  - US aerospace companies
  - Commercial tool vendors

  And through them I know some of what goes on in other industries, and in Europe

Trend: Maintain Safety, Reduce Cost

- Those industries (notably aerospace) with an established record in safety critical software and systems development
- Are satisfied with safety achieved
    - And their record justifies this
- But dissatisfied with the cost
- Both absolute dollars
- And time to market
- And difficulty customizing/adapting products
- So there is pressure to integrate/automate some of the safety-relevant processes

Trend: Maintain Safety, Integrate Functions

- Previously, safety-critical systems were federated
  - Each had its own fault-tolerant computing system
  - Few interactions between them
- Now becoming integrated
  - Resources shared among systems
  - Stronger interactions among them

  More functionality at less cost
  - Integrated Modular Avionics (IMA)
  - Modular Aerospace Controls (MAC)
  - Integrated steering, brakes, suspension (cars)
- New hazards from fault propagation, and unintended emergent behavior

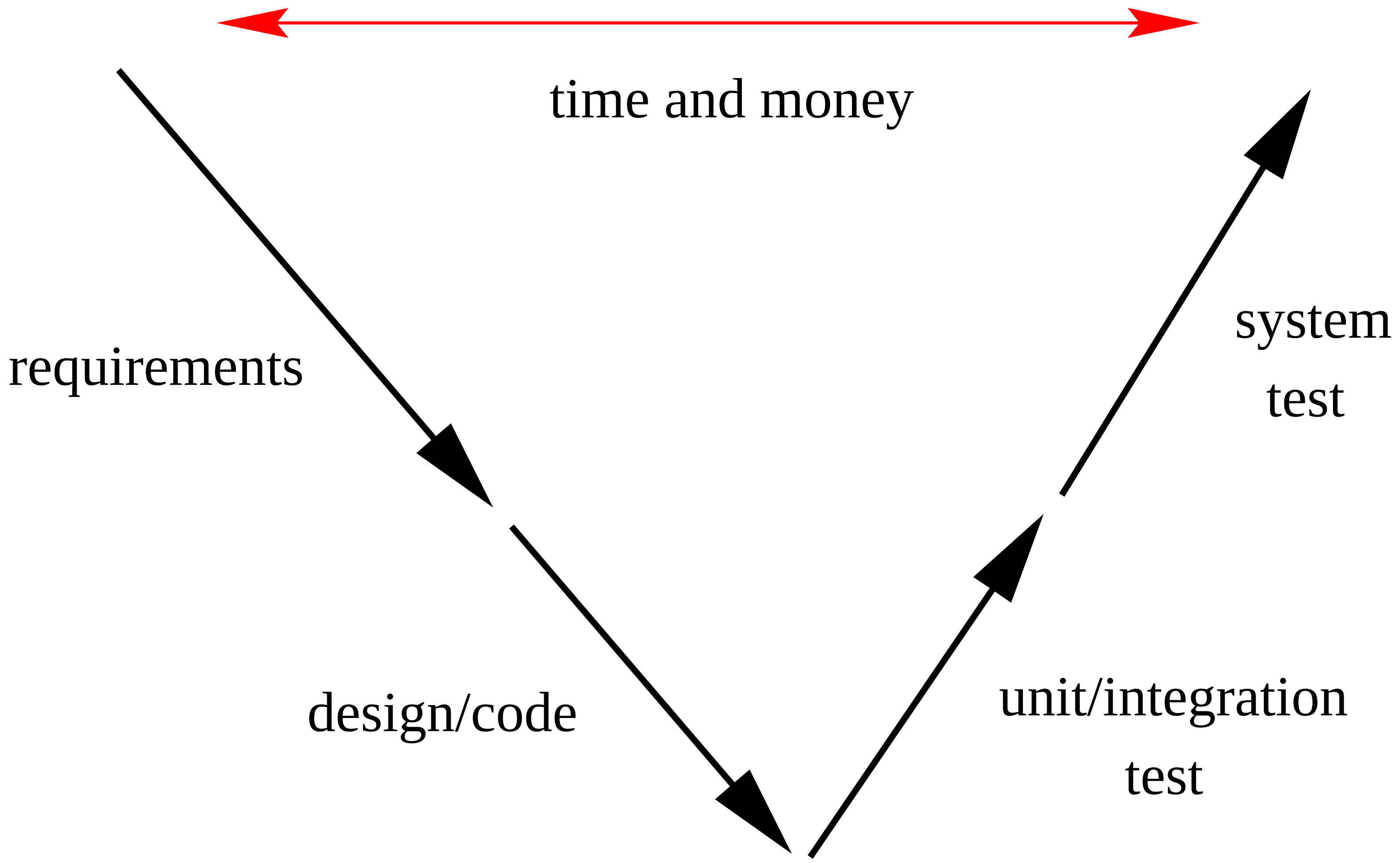
Responses to These Trends

- Model-based development
- Automated support for some safety processes
  - Based on formal analysis
- Standardized platforms
- Modular certification

Model-Based Development

- Organize development of the system around an “executable” model of the system and its environment
  - E.g., in Matlab/Simulink, SCADE/Esterel Studio, Statecharts, UML. . .
- Required for all Airbus contractors
- Comparable Boeing process being defined
- Development in Matlab/Simulink is standard in non safety-critical applications
- Early execution (simulation, animation) helps eliminate many requirements faults
- And integrated models help eliminate interface faults

Simplified Vee Diagram

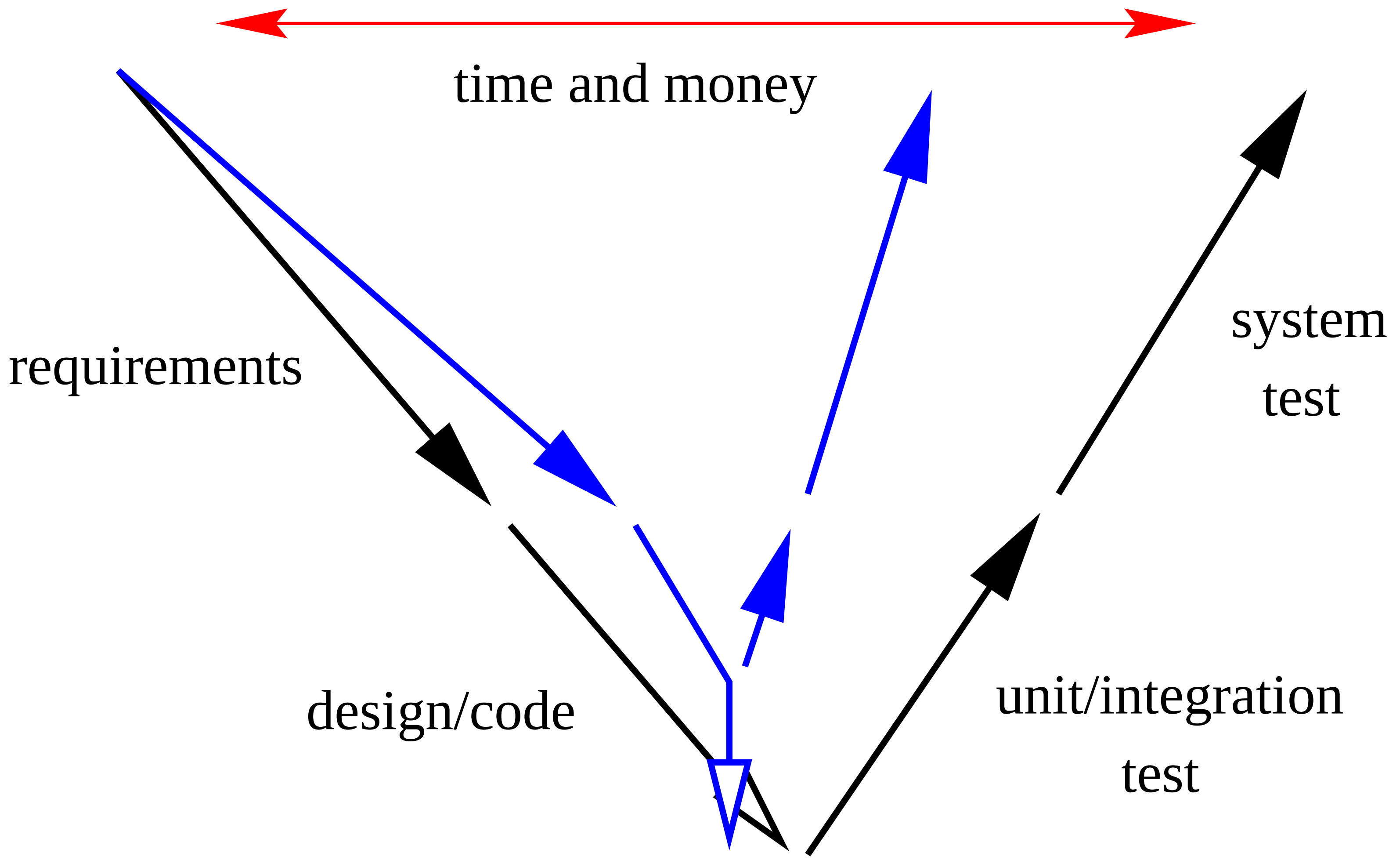


Hope is that model-based methods can tighten the vee

Tightening the Vee

- If model-based methods could reduce requirements faults, we would reduce the amount of rework, and steepen both sides
- If model-based methods could automate some testing procedures, would steepen right side (especially bottom right)
- If model-based methods could automate coding, would steepen bottom left
- If model-based methods could eliminate some testing procedures, would further steepen bottom right

Tightened Vee Diagram



Assurance in Model-Based Development

- Currently, gains in model-based development come from early simulation and integration of plant and (sub)system models
- And generally more integrated specifications and development
- And from automatic code generation in SCADE and Matlab
- The SCADE compiler (to C) is certified by JAA and unit tests are eliminated
    - FAA accepts this in JAA-certified aircraft, but I doubt they would buy it in their own certifications
- But much of this focuses on the control laws
- The real opportunity is in the discrete logic
    - Mode switching, redundancy management, displays etc.

Aside: Characteristics of Safety-Critical Systems

- Safety-critical systems are usually real-time embedded control systems
  - Specified by control engineers
  - Implemented by software engineers
- Only 20% (often much less) of the code implements the (continuous) control laws
- The other 80% is discrete logic
- All the complexity is in the discrete part
  - Redundancy management
  - Mode switching
  - Human factors (automation surprises, mode confusion)

  And it's where all the failures are

Assurance for Discrete Logic

- That is, requirements, specifications, code having lots of discrete conditions
- Whose possible combinations of different behaviors grows exponentially
- So complete testing is infeasible
- And absence of continuity means that extrapolation from incomplete testing is unsound
- However, symbolic analysis can (in principle) consider all cases
  - E.g., examine the consequences of $x < y$ rather than enumerating (1, 2), (1,3), (1, 4), . . . (2, 3),. . .
- This is what formal methods are about

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 separate assurance that model faithfully represents design, design is implemented correctly, environment is modeled faithfully, calculations are performed without error

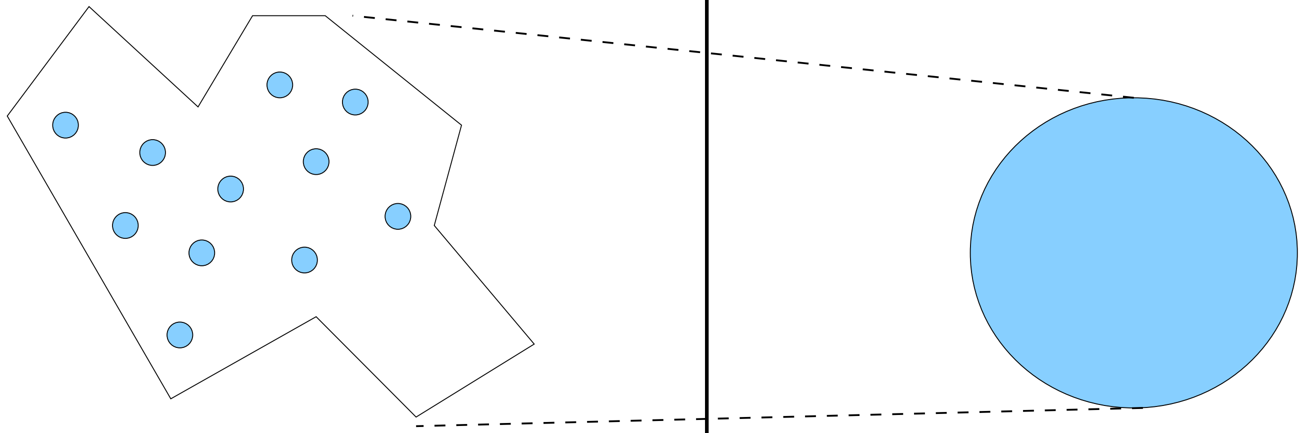
Formal Methods: Analogy with Engineering Math (ctd.)

- 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.
- Formal calculations (can) cover all modeled behaviors
- If the model is accurate, this provides verification
- If the model is approximate, can still be good for debugging (aka. refutation), test-case generation

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 Calculations: The Basic Challenge

- Build mathematical model of system and deduce properties by calculation
- The applied math of computer science is formal logic
- So 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

Application to Safety Critical Systems

- 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/ED-12B
- That is, formal methods help us move from process-based to product-based assurance

Formal Analysis in Model-Based Development

- The big opportunities are to automate verification of design against requirements
- And to automate generation of unit tests (and maybe integration and some system tests)
  - Airborne software must achieve MC/DC code coverage through functionally derived tests
- Both have potentially large gain in productivity without big certification impact
- And both are quite easy to do for plain state machine specifications
- However, . . .

Industrial Statechart Languages

- Most model-based development environments use some form of Statecharts notation to specify discrete behavior
- The formal semantics of basic Statecharts are quite complicated and less than ideal for formal analysis
- The variants used in Matlab (Stateflow) and UML add further bizarre complexities
- Though Esterel has an attractively simple form (SyncCharts)
- Options worthy of consideration
  - Use Esterel
  - Replace Stateflow by something simpler (e.g., RSML^{-e})
  - Severely subset Stateflow
  - Handle the full complexity

Design Verification in Model-Based Development

- Collins and U of MN used NuSMV to verify 170 requirements (e.g., “at most one lateral mode shall be active at any time”) against an RSML^{-e} specification for an autopilot
  - Analysis takes 4 minutes for all 170 requirements
- Highly automated theorem proving looks feasible for more difficult properties
- Big benefits are earlier and more reliable bug discovery
- And rapid, complete, and reliable reverification as the requirements and design evolve
- Supplements, rather than replaces review of the final, stable, requirements and design

Test Generation in Model-Based Development

- There are many commercial tools based on formal methods
  - E.g., T-VEC, RSI, TGV, STG, . . .
- Most based on model checking technology:
  - Formalize the purpose of the test in CTL or LTL
    - ⋆ Structural coverage criteria can be expressed as CTL or LTL formulas (apply to specification, not code for MC/DC)
  - Negate it and run the model checker
  - Counterexample found by model checker is a trace that satisfies the purpose
- Fully automated
  - Modulo the abstraction to finite state

Key Technology: Bounded Model Checking

- Given a system specified by initiality predicate I and transition relation T, there is a counterexample of length k to invariant P if there is a sequence of states $s_0, \ldots, s_k$ such that

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

- Given a Boolean encoding of I and T, this is a propositional satisfiability (SAT) problem
- SAT solvers have become amazingly fast recently
- Try $k = 1, 2, \ldots$ and submit each instance to a SAT solver
- Needs less tinkering than BDD-based symbolic model checker, can handle bigger systems and find deeper bugs
- Now widely used in hardware verification, test case generation
- But is limited to refutation. . . or is it?

Perimeter Calculation

- We should require that $s_0, \ldots, s_k$ are distinct
  - Otherwise there's a shorter counterexample
- And we should not allow any but s_0 to satisfy I
  - Otherwise there's a shorter counterexample
- If there's no path of length k satisfying these two constraints, and no counterexample has been found of length less than k, then we have verified P
- And paths in control software tend to be quite short

Bounded Model Checking for Infinite State Systems

- We can discharge the BMC and perimeter formulas efficiently for Boolean encodings of finite state systems because SAT solvers do efficient search
- If we could discharge these formulas over richer theories, we could do BMC for state machines over these theories
  - More concise
  - More accurate
- So how about if we combine a SAT solver with a decision procedure—e.g., ICS—for the combined theories?
  - ICS is a set of decision procedures for the combination of integer and real arithmetic, equality with uninterpreted functions symbols, etc. available from SRI

SAT-Based Constraint Satisfaction

- Idea is to extend the efficient search of a modern SAT solver to propositionally complex formulas with interpreted terms at the leaves
  - E.g., $x < y \wedge (f(x) = y \vee 2 * g(y) < \epsilon) \vee \ldots$ for hundreds of thousands of terms
- Replace the terms by propositional variables
- Get a solution from the SAT solver (if none, we are done)
- Restore the interpretation of variables and send the conjunction to the decision procedure
- If satisfiable, we are done
- If not, ask SAT solver for a new assignment

SAT-Based Constraint Satisfaction (ctd)

- But first, do a little bit of work to find some unsatisfiable fragments and send these back to the SAT solver as additional constraints (lemmas)
- Iterate
- Works amazingly well
- Example, given integer x: $(x < 3 \wedge 2x \geq 5) \vee x = 4$
  - Becomes $(p \wedge q) \vee r$
  - SAT solver suggests $p = T, q = T, r = ?$
  - Ask decision procedure about $x < 3 \wedge 2x \geq 5$
  - Add lemma $\neg(p \wedge q)$ to SAT problem
  - SAT solver then suggests $r = T$
  - Interpret as $x = 4$ and we are done

ICSAT (ICS Decision Procedure + SAT)

- We combined ICS with Chaff: worked well, but. . .
  - Chaff wants input in CNF
    (which is expensive to compute)
  - Sometimes does more than we need (in asynchronous composition, we only want assignments to variables of one process, but Don't Cares can interfere with search)
  - As a black box, hard to do efficient incremental restarts
    - ⋆ Note: decision procedure needs to be incremental, too
  - Licensing terms
- We replaced Chaff by a nonclausal solver designed for restarts and Don't Cares and gained another two orders of magnitude

Infinite BMC etc.

- So now we can do BMC over systems defined using terms from the theories decided by ICS
- Not only more general, but sometimes faster too
  - E.g., encoding bitvectors in SAT vs. using the ICS decision procedure for bitvectors
- Additionally, can augment any class of problems traditionally handled by SAT solvers (e.g., AI planning, diagnosis) to descriptions including decided theories
  - E.g., proofs of inductive invariance (the bugbear of automated deduction because of the need to strengthen invariants to make them inductive)

Automated Induction via BMC

- Ordinary inductive invariance (for P):

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

  Step: $P(r_1) \wedge T(r_1, r_2) \supset P(r_2)$

- Extend to induction of depth k

  Basis:

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

  Step: $P(r_1) \wedge T(r_1, r_2) \wedge P(r_2) \wedge \cdots \wedge P(r_{k-1}) \wedge T(r_{k-1}, r_k) \supset P(r_k)$

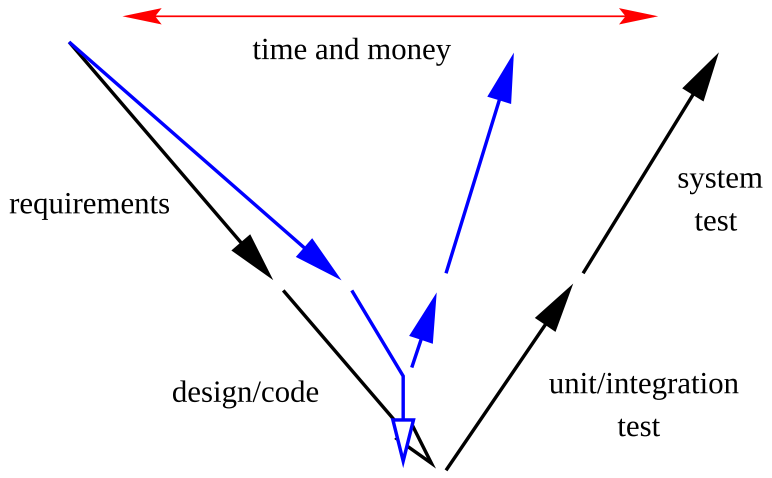
  These are close relatives of the BMC formulas

- Need to avoid loops and degenerate cases in the antecedent paths in the same way as BMC

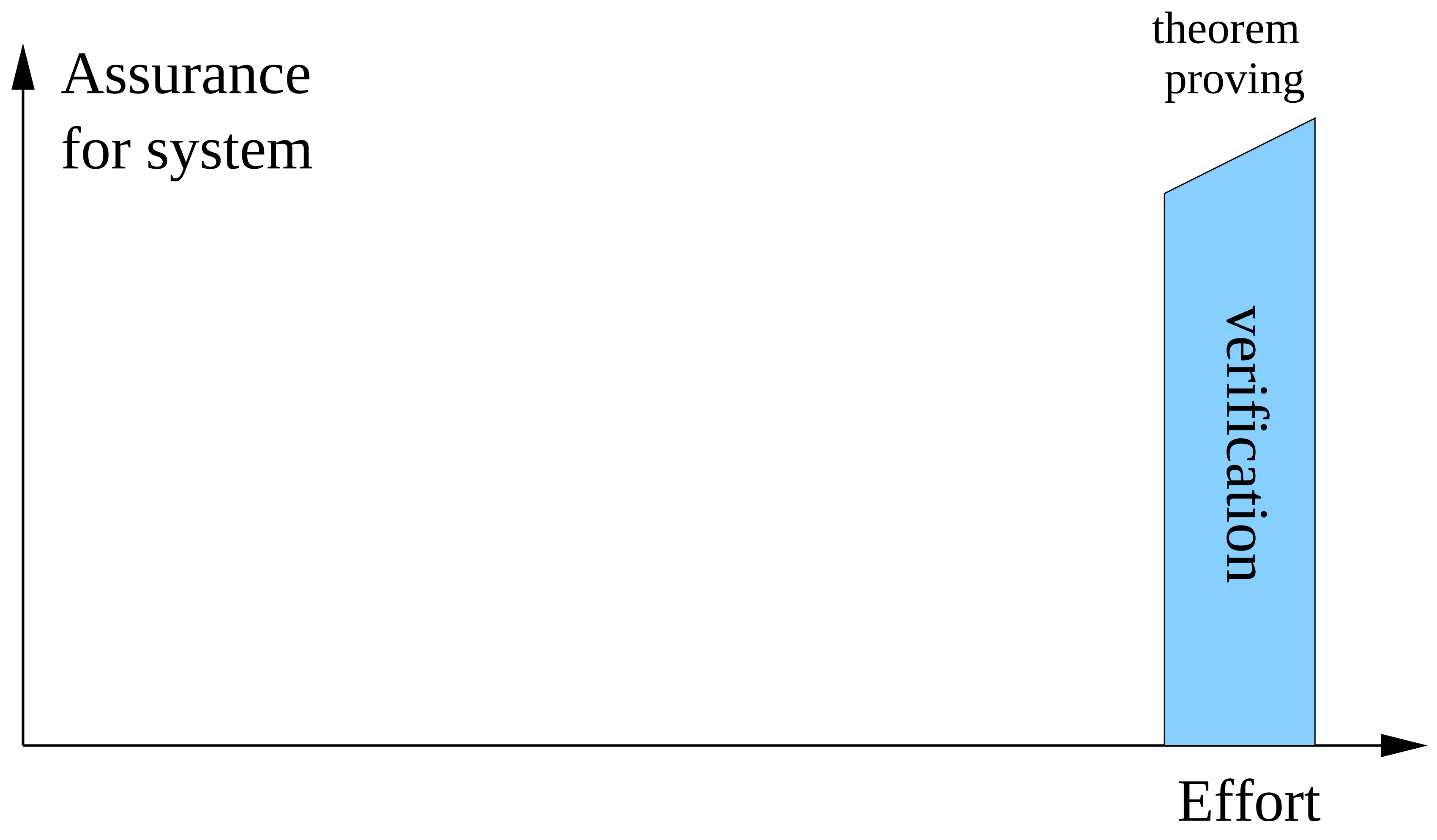
- Induction for $k = 2, 3, 4 \ldots$ sometimes succeeds where $k = 1$ does not

Summary: Model-Based Development and Assurance

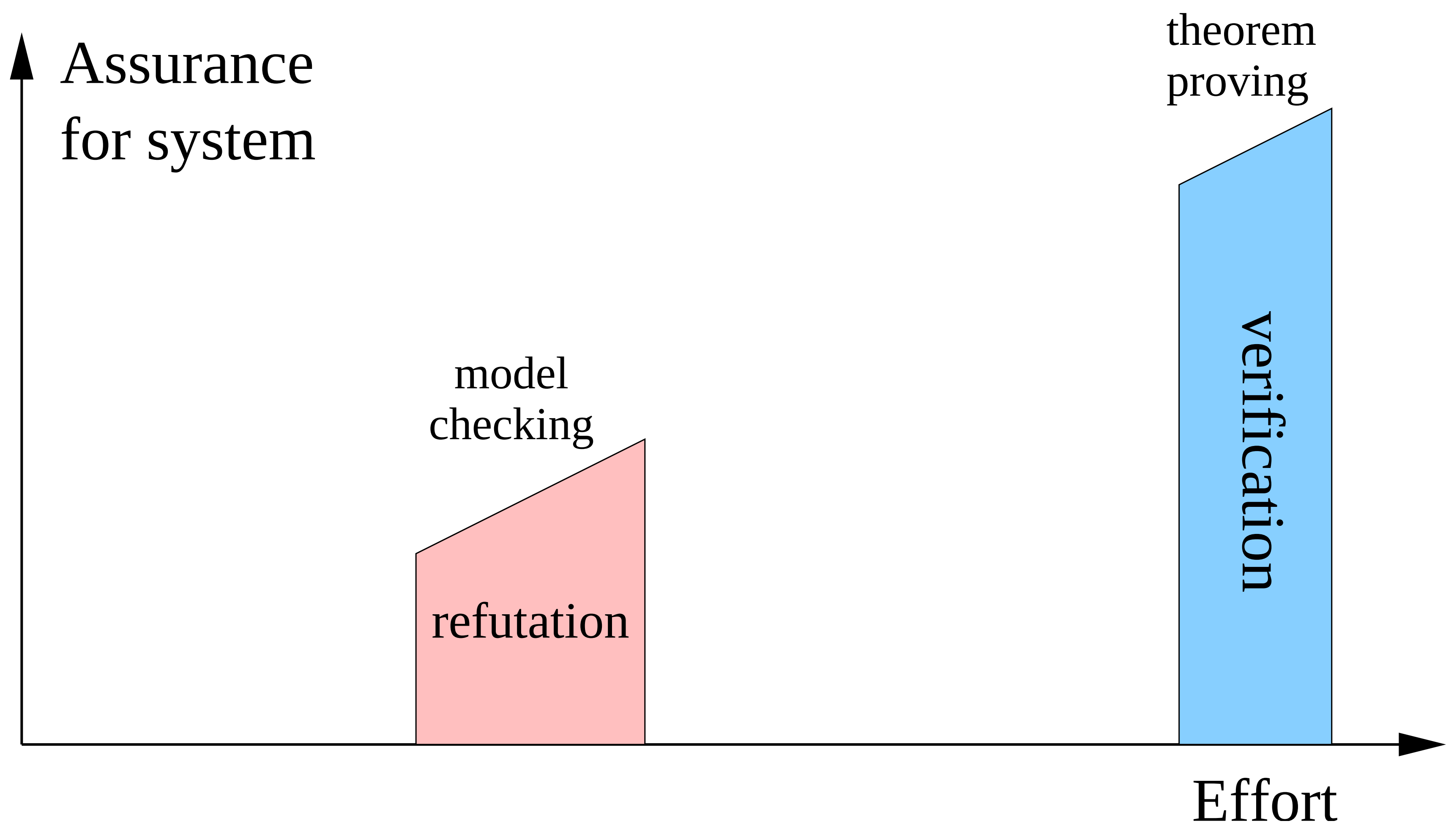
- There is a plausible path to increased productivity for safety-critical software, centered on model-based development
- And automated formal methods can tackle the tasks of
  - Refutation/debugging in the design loop
  - Unit and some systems verification
  - Unit and some system test case generation



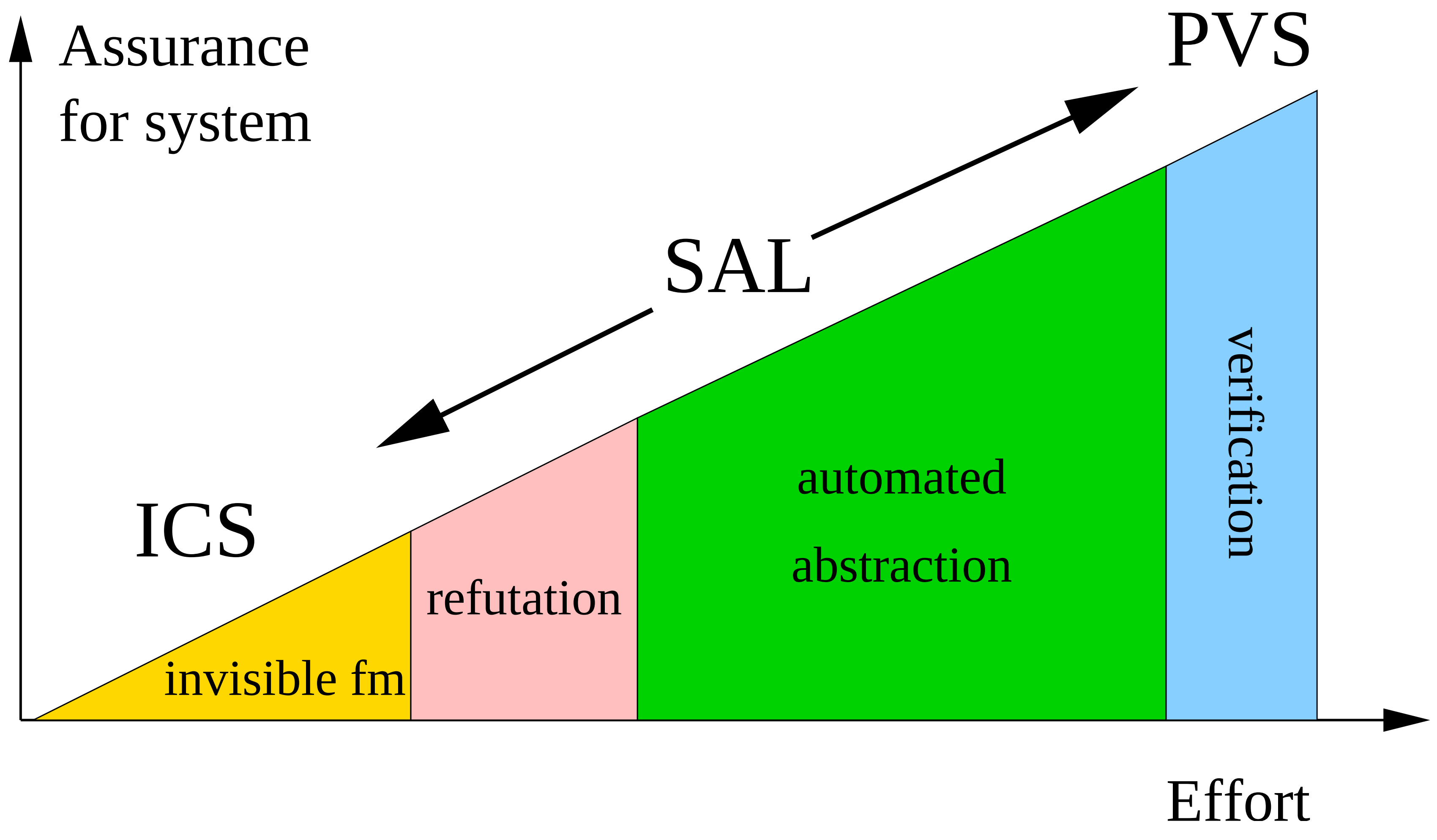
Verification by Traditional Theorem Proving: The Wall



Traditional Model Checking: An Island



An Integrated Picture



Trend (redux): Maintain Safety, Integrate Functions

- Previously, safety-critical systems were federated
  - Each had its own fault-tolerant computing system
  - Few interactions between them
- Now becoming integrated
  - Resources shared among systems
  - Stronger interactions among them

  More functionality at less cost
  - Integrated Modular Avionics (IMA)
  - Modular Aerospace Controls (MAC)
  - Integrated steering, brakes, suspension (cars)
- New hazards from fault propagation, and unintended emergent behavior

Partitioning

- Restores to integrated systems the strong barriers to fault propagation of federated architectures
- Failure of one component must not affect ability of others to function and communicate
- Allows low and high-criticality functions to coexist
- Allows high-criticality functions to be deconstructed
  - Into components of differing levels
  - Which allows provision of additional capabilities
  - Such as health maintenance
- Strong composability is a dual to partitioning

Fault-Tolerant Architectures

- Provide basic services to a collection of host computers
    - Timing, communication, partitioning

  These services must not fail, despite failure of components

- Support fault-tolerant applications in the hosts
    - Consistent message delivery, failure notification
    - E.g., through state machine replication

  These simplify construction of correct fault-tolerant applications

  And must not fail

Fault-Tolerant Architectures (ctd)

- Federated systems generally employ “homespun” fault tolerance mechanisms
  - Uninfluenced by academic knowledge
  - Primary source of failure in some examples
- Integration raises the stakes and architectures generally build on a lineage of research architectures that developed principled solutions to the challenges of concurrent, real-time, distributed, fault-tolerant systems design
  - SIFT (SRI), FTP, FTPP (Draper), MAFT (Allied Signal), MARS (TU Vienna)

The Rôle of Buses

- There must be some communication system for exchanging sensor samples, state data, control signals, actuator outputs
- Many possible topologies, but only a serial bus is economically viable
- The bus is then a critical shared resource
  - Communication must be assured with guaranteed bandwidth, low jitter, low end-to-end latency
  - In the presence of faults
- Bus embodies the fault-tolerant architecture

Safety-Critical Buses

- Avionics: SAFEbus (Honeywell 777 AIMS), SPIDER (NASA)
- Automotive: TTA (TU Vienna, TTTech), FlexRay (Daimler/Chrysler et al)
- I've written a NASA Tech Report and a paper presented at EMSOFT '01 that compare them
- Use Google to find my home page, follow link to my papers

The Move To Standardized Components

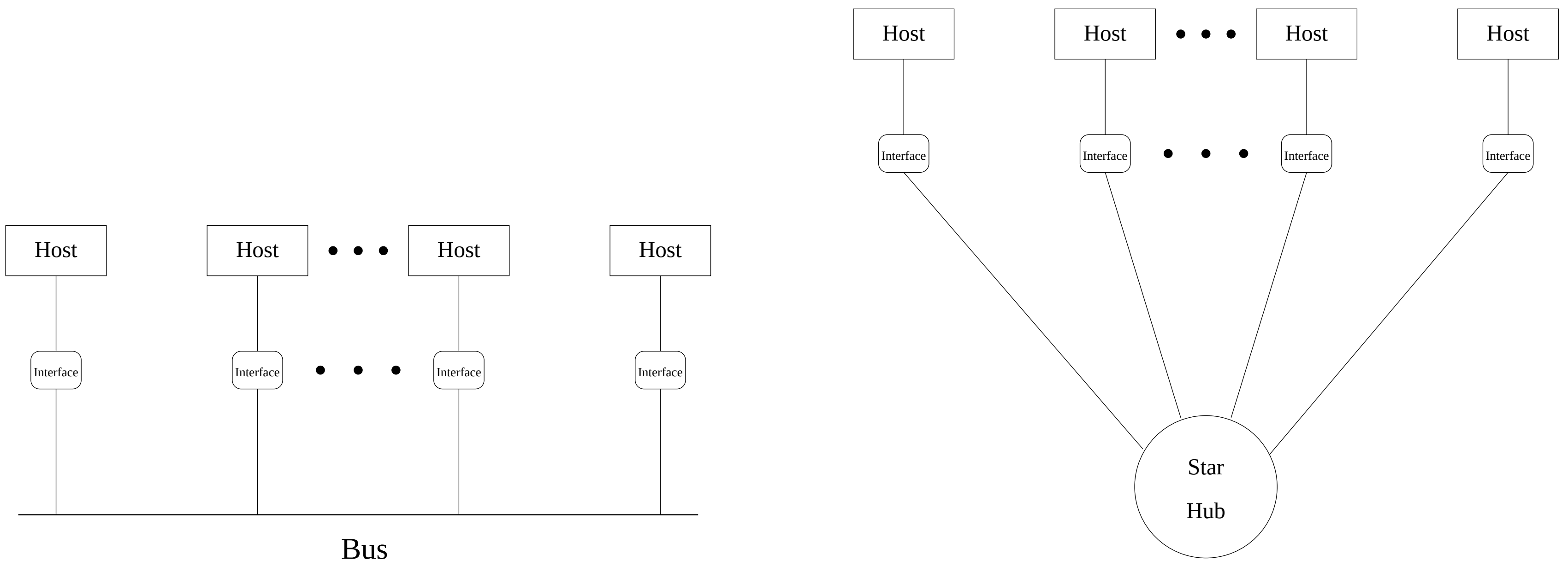
- There is pressure to use COTS components
- Despite their unsuitability for safety-critical applications
- But what if safety-critical components achieved sufficient volume to become COTS?
- The automobile industry is where this could happen
- And an accepted bus architecture is the enabling component

The Time Triggered Architecture (TTA)

- TTA is unique in being developed for mass-market for automobile applications (Audi, PSA etc.) but also used for aircraft applications (Honeywell)
  - “Aircraft safety at automobile cost”
- Example TTA applications
  - Engine controller for an Italian fighter (Honeywell Tucson)
  - Engine controller for F16 (Honeywell Tucson)
  - Environmental control for A380 (Hamilton Sundstrand)
  - GenAv cockpits (Honeywell Olathe)
  - By wire applications in next generation cars (Audi, PSA. . .), Snowcats, . . .

Basic Characteristics of TTA

- Exists in both bus and star topologies (logically still a bus)

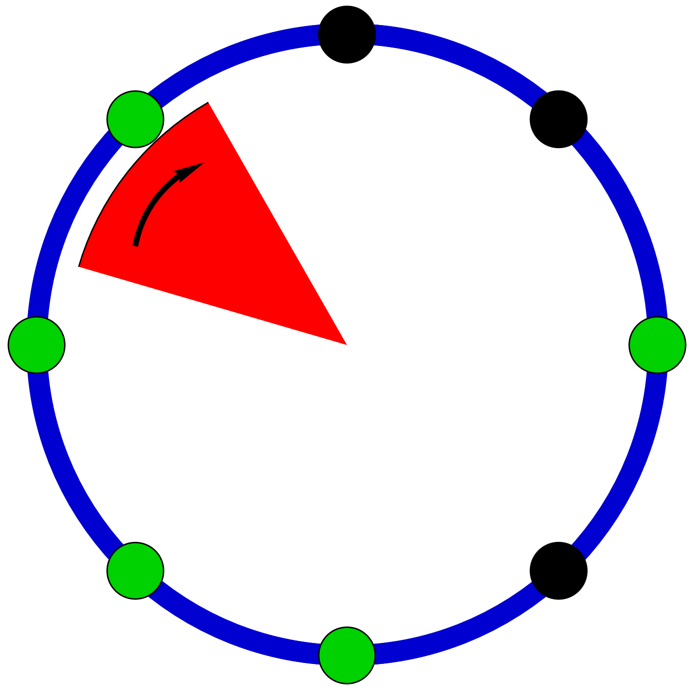


Bus/hub are replicated

- All functionality implemented in the distributed interfaces (called TTP/C controllers)
- And in the hub of the star topology (a modified controller)

Basic Characteristics of TTA (ctd.)

- Creates a synchronous, TDMA ring on a broadcast bus



- Global clock (achieved by synchronizing local clocks)
- Global schedule known at all nodes

Basic Algorithms of TTA

- Clock synchronization
- Bus guardian window timing
- Group membership
- Clique avoidance
- Nonblocking write
- Startup/restart

TTA Activity

- In addition to those of its developers (TTTech)
- There are several activities, EU and US, government, commercial, and academic contributing to development of the safety case for TTA
- Some of these focus on formal verification of the TTA algorithms
- Projects
  - SRI, with Honeywell Tucson and NASA
  - NextTTA: TU Vienna, VERIMAG, Ulm, . . .
- Academic Groups
  - Liafa, Paris 7
  - PAX, Kiel

Formal Verification for TTA

Safety motivation:

- Need all the assurance possible
- Help move certification from process- to product-basis
- Help develop approach to modular certification

Developer (TTTech) motivation:

- Product discriminator
- Formal proof gets into all the corners, may find bugs
- Formal proof exposes assumptions (fault hypotheses)
- Model checking and mechanized proof allow refined design exploration

  Pruning of assumptions, strengthening of claims

Formal methods motivation:

- TTA algorithms are challenging, push the technology of automated verification

The TTA Algorithms Are a Fascinating Challenge

- TTA comprises several algorithms
- That are individually challenging for formal verification
- Even in their “academic” form
  - Hard to do at all
  - Really hard to automate

  Further complicated by practical details
- The algorithms interact in interesting ways
- Some of the most important properties are emergent
  - Consistent message delivery is achieved indirectly, not by an agreement algorithm
  - Partitioning is not ensured by any individual algorithm
- And the top-level properties are tricky to characterize
- See my FTRTFT’02 paper for summary of current state

Summary: Standardized Platforms

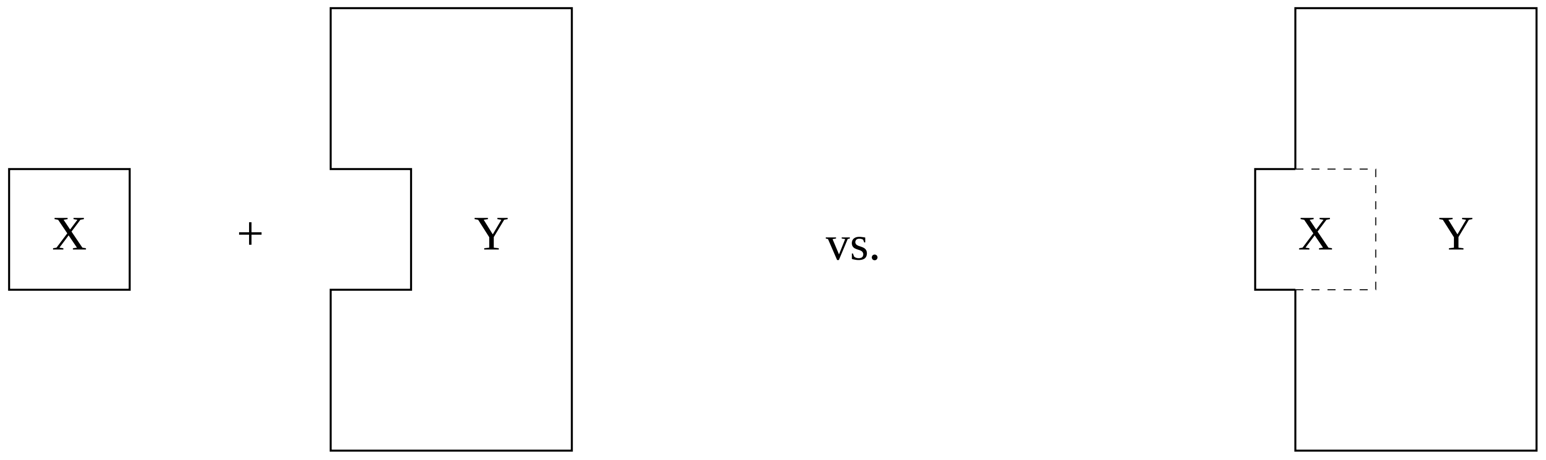
- TTA and comparable bus architectures provide standardized platforms for the safety-critical applications
  - They provide fault tolerance based on rational principles (displacing homespun solutions)
  - And encourage similarly rational approaches to design of fault tolerant applications
- They provide partitioning, thereby supporting both integration and deconstruction of safety-critical applications
- Formal verification of its algorithms is a challenging problem
  - But only needs to be done once
- Formalizing computational model and properties presented to client applications is even more challenging, but crucial
- Can then bring formalization and verification to those clients
- And contemplate modular certification

Modular Certification

- Safety-critical buses like TTA allow several safety-critical functions to coexist
- Modular certification is the ideal that each function could be largely “precertified” on its own
- Final certification is an integration of the separately precertified components
- Currently the smallest unit of certification is a complete airplane or engine
- But RTCA SC200 and its European equivalent are holding joint meetings to develop a basis for separate certification

Modular Certification: The Ideal

- Certification argument for system Y with component/subsystem X makes use of separately certified claims about X's properties at its interfaces
- As opposed to opening up the design of X
- Pictorially:



- Think of X as some onboard function of airplane Y

Modular Certification: The Benefits

- Assuming we have certified X and Y “in isolation”
- The incremental cost of the modular certification X+Y should be less than that of the joint certification $\boxed{\text{XY}}$
- Especially if X is reused across many applications
    - Attractive to suppliers of X
- Or if there are many X’s and the owner of Y only has to develop the integration argument
    - Attractive to developer of Y
- Requires that reasoning about properties at the interface is simpler than reasoning about the design

Modular Certification: The Difficulty

- Much of the effort in certification is not in showing that things work right when all is going well
- But in showing that the system remains safe in the face of hazards
- In the case of X+Y, we have to consider all the hazards that X can pose to Y and vice-versa
- Hazards include malfunction as well as loss of function
- Difficult to anticipate all the hazards X might pose to Y without knowing quite a lot about Y, and vice-versa
  - Think of Concorde’s tires
- Hazards may bypass the traditional notion of “interface”

Compositional Verification

By Assume-Guarantee Reasoning

- The idea of verifying properties of one component X_1 on the basis of assumptions about another X_2 (and, in general, X_3, $X_4 \ldots$), and vice-versa is called assume-guarantee reasoning and is fairly well developed in computer science
- But there are two challenges:
  - The approach looks circular, so how do we get a sound method? (not further developed here)
  - Assume-guarantee reasoning is used for verification, but we are interested in certification

Certifying Combinations of Software

- Certification differs from verification in that we have to take failures and hazards into account
- Maybe we can split the problem into normal/abnormal cases
  - Can we certify the normal operation of X_1 and X_2 by assume/guarantee methods?
    - ⋆ E.g., the thrust reverser does its thing assuming the engine controller supplies some sensor readings
    - ⋆ Yes, this is similar to other assume/guarantee applications
  - Can we do something similar for the hazards?
    - ⋆ That is, provide guarantees on the behavior of X_1 when X_2 does not fulfill its assumptions?
    - ⋆ This is the hard one!

Assume/Guarantee Reasoning Over Hazards

How can X_2's failure to fulfill its guarantees disturb X_1?

Behavioral failure: X_1 depends on data or control values from X_2 in such a way that its computation becomes hazardous when X_2 fails to satisfy its guarantees

- E.g., X_1 relies on sensor data from X_2 with no independent backup

Interface failure: X_2 affects X_1 through channels other than their defined interface

- E.g., X_2 corrupts or monopolizes resources used by X_1 (memory, communications bandwidth)

Function failure: X_1 cannot guarantee the safety of its function if X_2 allows the system it controls to malfunction

These are the only hazards X_2 can pose to X_1

Controlling The Hazards

- We must control these three classes of hazards
- Some require that X_1 and X_2 have certain properties
- And some require architectural properties of the environment in which X_1 and X_2 operate

The Environment Must Enforce Partitioning

- Interface failure cannot be allowed; X_1, X_2,. . . must operate within an environment that enforces partitioning
  - Such as the traditional federated architecture
  - Or an IMA or MAC architecture such as SAFEbus or TTA
- Must ensure that no failure of X_2 can affect the basic operation or timing of X_1, nor its communications with nonfaulty X_3, X_4,. . .
- Top-level requirement specification for partitioning:
  - Behavior perceived by nonfaulty components must be consistent with some behavior of faulty components interacting with it through specified interfaces
- A partitioning architecture must be certified to satisfy this requirement

Normal and Abnormal Assumptions and Guarantees

- In most concurrent programs one component cannot work without the other
  - E.g., in a communications protocol, what can the sender do without a receiver?
- But the software of different aircraft functions must not be so interdependent
  - In fact, X_1 must not depend on X_2
  - In worst case, X_1 must provide safe operation of its function in the absence of any guarantees from X_2
  - Though may need to assume some properties of the function controlled by X_2 (e.g., thrust reverser may not depend on engine controller, but may depend on engine remaining under control)

Normal and Abnormal Assumptions and Guarantees (ctd)

- In general, X_1 should provide a graduated series of guarantees, contingent on a similar series of assumptions about X_2
  - These can be considered the normal behavior of X_1 and one or more abnormal behaviors
- A component may be subjected to external failures of one or more of the components with which it interacts
  - Recorded in its abnormal assumptions of those components
- A component may also suffer internal failures
  - Documented as its internal fault hypothesis

Components Must Avoid Behavioral and Function Failure

True guarantees: under all combinations of failures consistent with its internal fault hypothesis and abnormal assumptions, the component must be shown to satisfy one or more of its normal or abnormal guarantees.

Safe function: under all combinations of faults consistent with its internal fault hypothesis and abnormal components, the component must be shown to perform its function safely (e.g., if it is an engine controller, it must control the engine safely)

Avoiding Domino Failures

- If X_1 suffers a failure that causes its behavior to revert from guarantee $G(X_1)$ to $G'(X_1)$
- May expect that X_2's behavior will revert from $G(X_2)$ to $G'(X_2)$
- Do not want the lowering of X_2's guarantee to cause a further regression of X_1 from $G'(X_1)$ to $G''(X_1)$ and so on

  Controlled failure: there should be no domino effect.

  Arrange assumptions and guarantees in a hierarchy from 0 (no failure) to j (rock bottom). If all internal faults and all external guarantees are at level i or better, component should deliver its guarantees at level i or better

  This subsumes true guarantees

Summary

- Model-based development creates an environment in which automated formal methods can significantly reduce costs and enhance safety with little certification impact
- Standardized safety-critical bus architectures provide a rational foundation for application development, fault tolerance, and integration
  - And are the targets of very searching formal analysis
- There is a plausible approach to modular certification that depends on three properties
  - Partitioning
  - Safe function
  - Controlled failure (which subsumes true guarantees)
- Together, these promise enhanced safety, at reduced cost