JSLC, Grenoble 6–8 November 2001

Formal Verification
For The Time Triggered Architecture

And a Tutorial on Fault-Tolerance for Safety-Critical Embedded Systems

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

The Time-Triggered Architecture: What Is It?

- The Time-Triggered Architecture (TTA) is a platform for safety-critical embedded systems
  - E.g., aircraft and engine flight control, and "by wire" cars
- Functionally, it is a TDMA (time-triggered) serial bus
  - Same as other such platforms:
    SAFEbus, SPIDER, FlexRay
- "Bus" understates its criticality and sophistication
  - It is the safety-critical core of the systems built above it
- Must achieve failure probability below 10^{-10}/hour for 10 hours, maximum outage 10ms
- Incorporates principled, integrated solutions to many of the classical issues in concurrent, real-time, distributed, fault-tolerant systems designs

Applications of TTA and Similar Buses

- Safety-critical embedded systems

  Avionics "functions": flight control, autopilot, autoland, flight management, displays...

  Aircraft "controls": engine controls, thrust reversers, cabin pressurization, brakes, doors and slides, public address,...

  Automotive: "by wire" brakes, suspension, steering,...

- Control laws are 20% of code
- Other 80% is coordination, fault tolerance, redundancy management
  - all failures here
- It must be possible to build such applications on TTA
- Ideally, it should be simple to do so

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
- The bus must not fail, should help applications not to fail

The Additional (New) Challenge: Integration

- Previously, these 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
    - Strong composability is a dual to partitioning
- Allows high-criticality functions to be deconstructed
    - Into components of differing levels
    - Which allows provision of additional capabilities
- The bus has primary responsibility for enforcing partitioning

Why Focus on TTA?

- There are other safety-critical buses
- Avionics: SAFEbus, SPIDER; Automotive: TTA, FlexRay
- I've written a NASA Tech Report and a paper presented at EMSOFT that compare them
  - Use Google to find my home page, follow link to my papers
- 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"
- Developed by the group of Hermann Kopetz, TU Vienna, commercialized by TTTech

Why Formal Verification?

Safety motivation:

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

Developer (TTTech) motivation:

- Nowadays, expected to have at least an informal proof
- 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

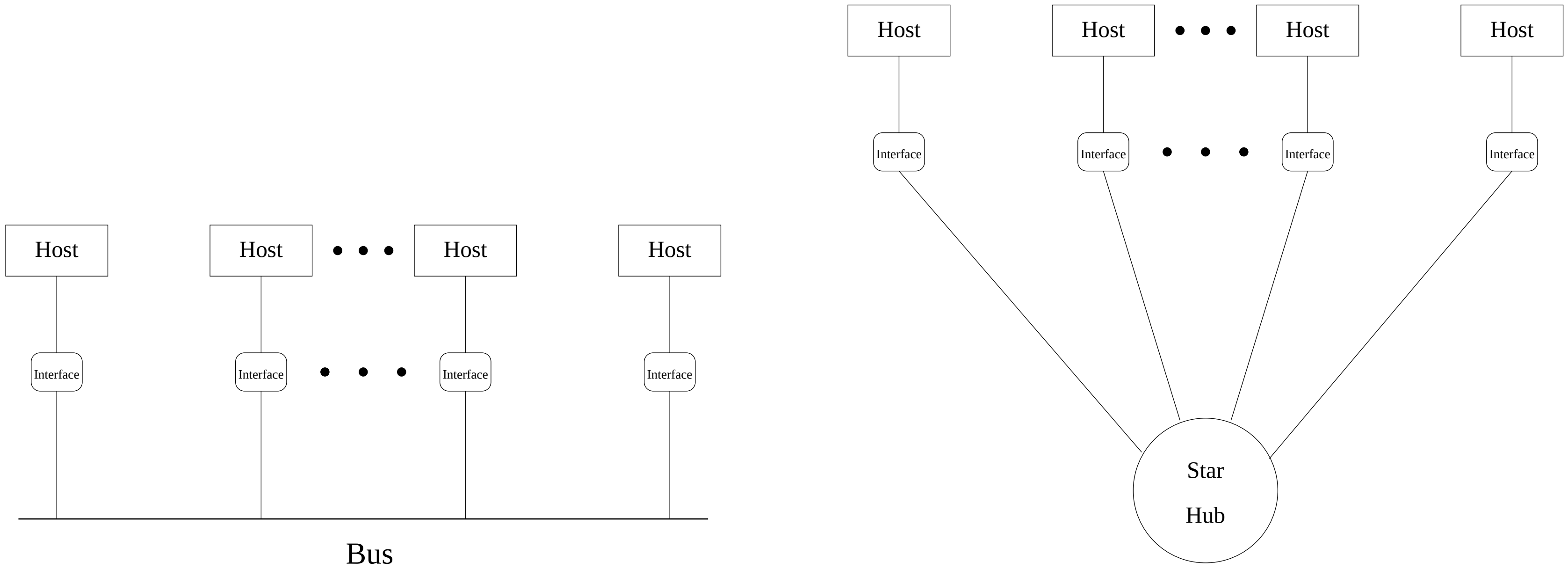
  Pruning of assumptions, strengthening of claims

Formal methods motivation:

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

Basic Characteristics of TTA

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

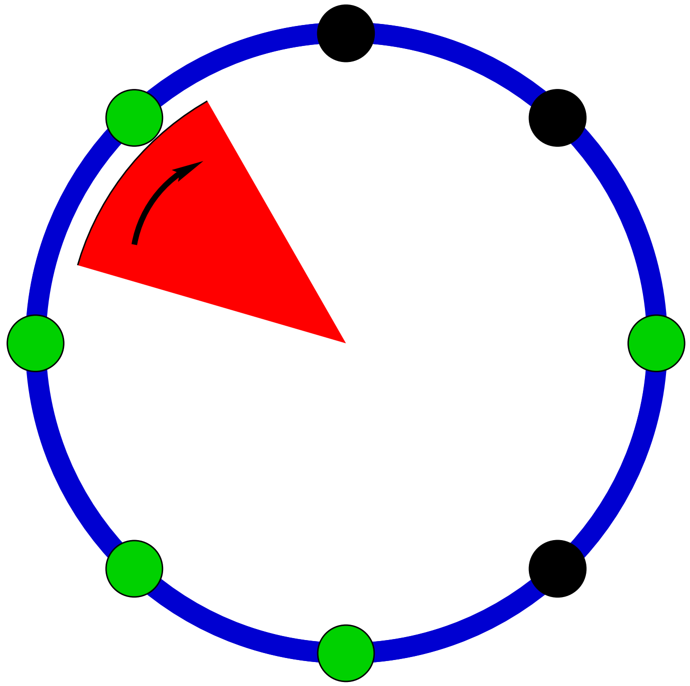


Bus/hub are replicated

- All functionality implemented in the distributed interfaces (called TTP/C controllers)
  - And 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

Fault Hypothesis and Fault Containment Units

- If we're serious about fault tolerance, must state an explicit fault hypothesis
  - The modes (kinds), number, and arrival rate of faults that we tolerate
  - Must be demonstrably realistic
- And should have a never give up (NGU) strategy when it is violated
- And must identify the fault containment units (FCUs) that faults can afflict
  - Faults at different FCUs must be independent
  - Must be demonstrably realistic (separate power, physically apart)
- Architecture must be shown to satisfy the mission requirements under these hypotheses

Formal Verification and Stochastic Modeling

- Architecture must be shown to satisfy the mission requirements under these hypotheses
- Formal verification establishes theorems of the form

  fault hypothesis satisfied $\vdash$ architecture works correctly

- Stochastic modeling establishes the plausibility of the fault hypothesis and its ability to satisfy the mission requirement

  System failures that could lead to a catastrophic failure condition must be "extremely improbable," which means that they must be "so unlikely that they are not anticipated to occur during the entire operational life of all airplanes of one type" ... "When using quantitative analyses... numerical probabilities... on the order of 10^{-9} per flight-hour [FAA Advisory Circular 25.1309-1A]

Specific, Arbitrary, and Hybrid Fault Models

Specific: enumerate the possible fault modes, provide defense for each one

- Need to show no other kind of fault can occur

Arbitrary (aka. Byzantine): no assumptions at all on behavior of faulty elements

- Requires a lot of redundancy
- Could fail under lots of simple faults

Hybrid: combination of the above

- Originally: arbitrary, symmetric, and manifest node faults
- Improvement: adds omission node fault, plus link faults
- Just right

Basic Algorithms of TTA

- Clock synchronization
- Bus window timing
- Group membership
- Clique avoidance
- Interactive consistency
- Startup/restart

TTA Clock Synchronization

- Keeps good clocks close together, in presence of faulty clocks
- Based on the Lundelius-Lynch algorithm
  - Each node collects clock differences wrt. other nodes
  - Takes average of 2nd smallest and 2nd largest as its correction
- But TTA uses only 4 clock differences
- Tolerates a single arbitrary fault

Bus Guardians

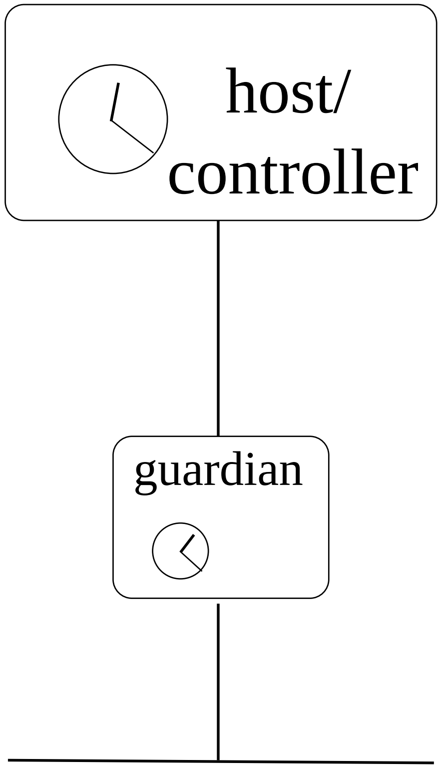
- A faulty node could broadcast at the wrong time
- Or all the time (babbling fault mode)
  - Destroys all good communications
- Must introduce a separate FCU with own clock and knowledge of schedule that mediates access to the bus
- This is a (logical) bus guardian
- Several design choices

SAFEbus: paired interfaces (and buses): each is a guardian for the other

TTA-bus, FlexRay: explicit guardians

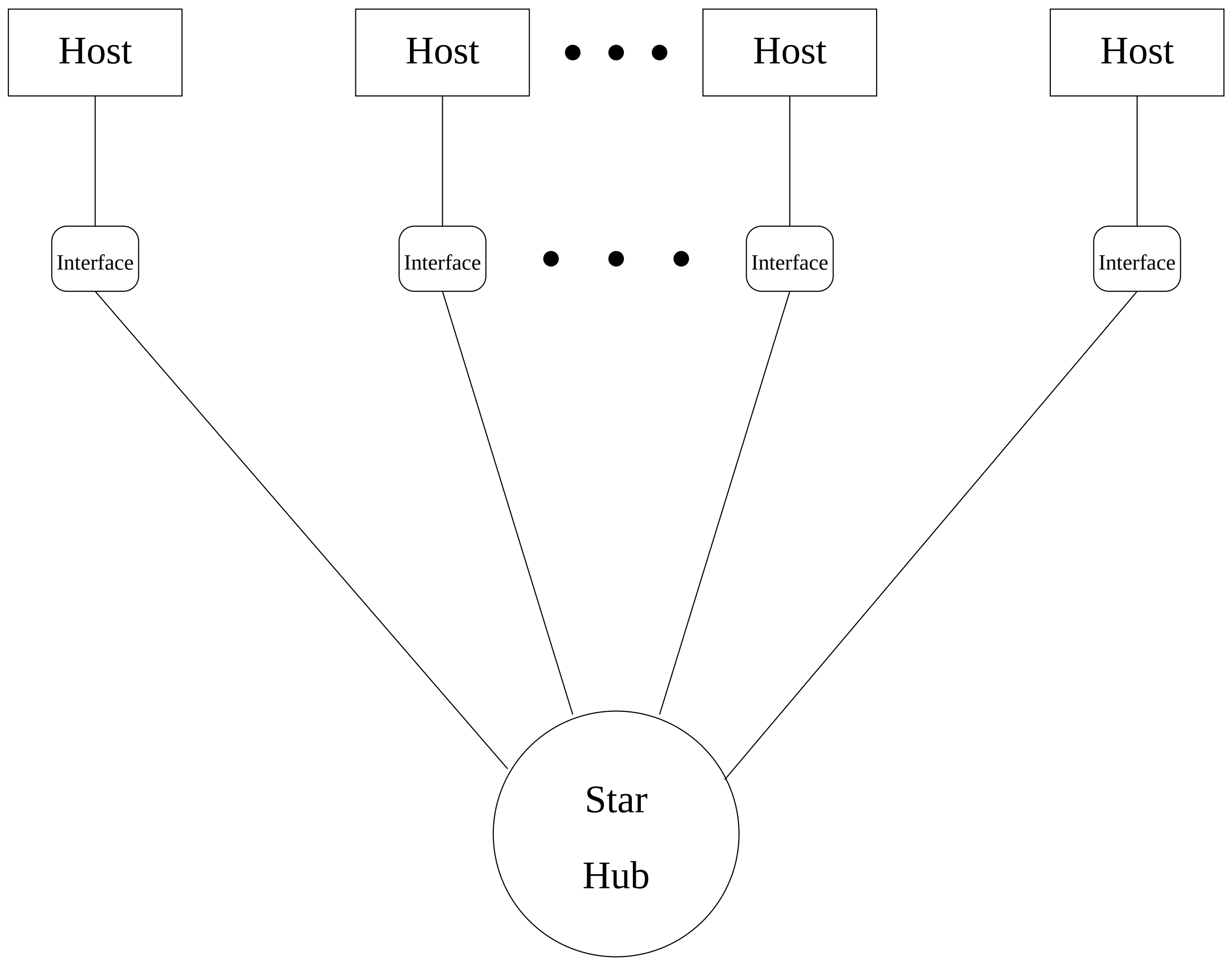
TTA-star: guardian functionality in central hub

Explicit Guardian



- One per bus, or shared?
- Fully independent clock synchronization?

Guardian in Central Hub



Bus Window Timing

- Bus guardian allows its node to write to the bus only during a limited window
- Want the bus guardian window to be as narrow as possible
- But still pass all messages from nonfaulty nodes
- Despite the fact that clocks are only loosely synchronized
- Also, no source or destination addresses are sent with messages
  - These are determined by time message sent
  - Eliminates masquerading, greatly increases bandwidth
- So receivers also maintain a narrow reception window

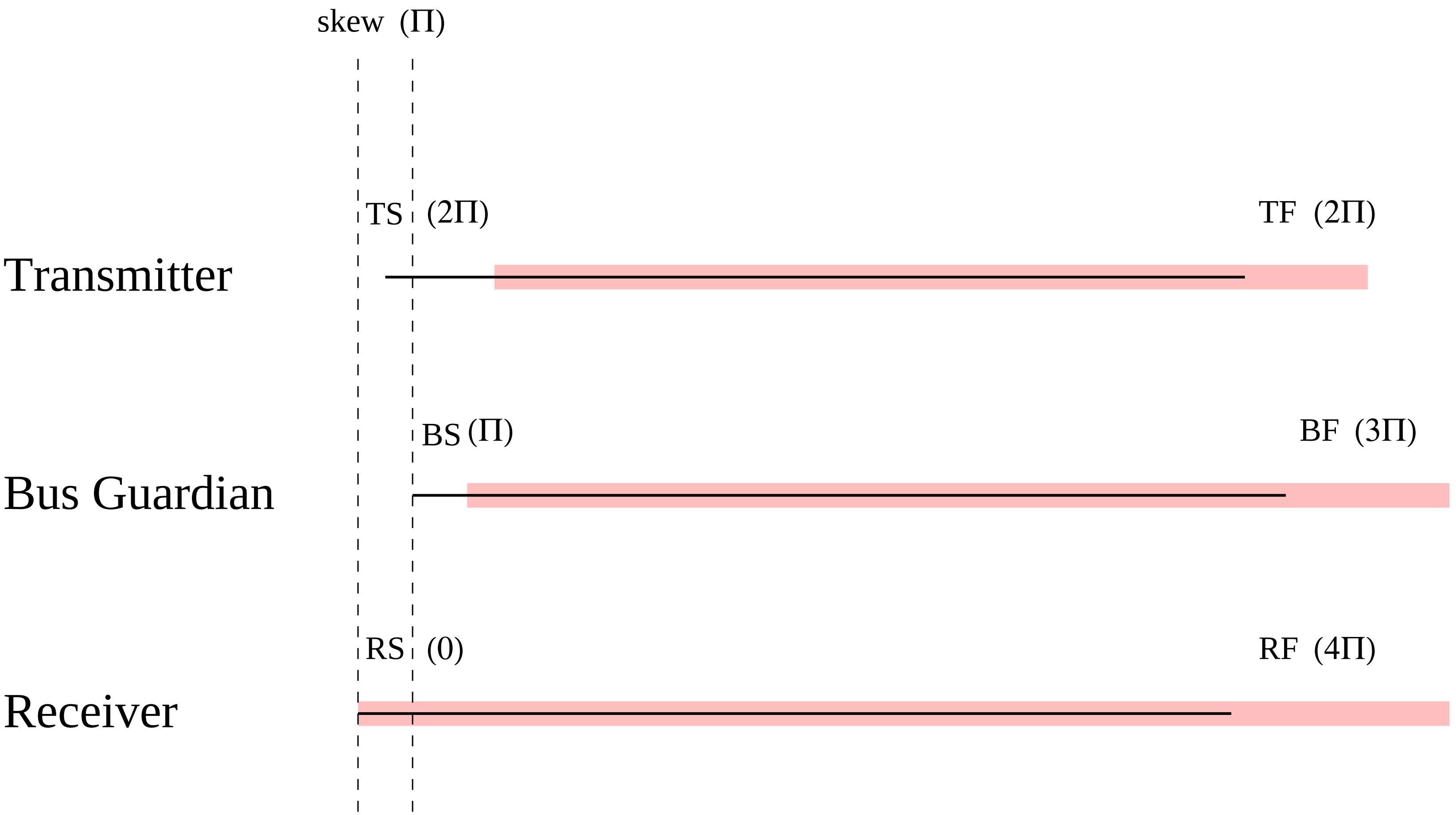
Window Timing: Requirements

- 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.

Group Membership

- Similar to fault diagnosis
- Informs good nodes which other nodes are good
- Needed for internal fault-tolerance of TTA
    - TTA is designed to single fault assumption
    - Membership excludes faulty nodes, can then tolerate new faults
    - Therefore its properties are a strong influence on the fault hypothesis and arrival rate
- Is also an application-level service (see later)

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

TTA Group Membership Algorithm

- Requires only two bits per message (encoded in CRC)
- Each broadcaster acknowledges the previous two
  - Can also be done with only one bit per message (WDAG '97)
- Works only under symmetric fault model
- And no more than one fault per two rounds

Clique Avoidance

- Implementation integrated with membership, but logically a separate algorithm
- Forces agreement on membership when outside fault hypothesis of membership algorithm
  - So part of "never give up" strategy
- May sacrifice validity

Services

- Basics make it possible to build safe, fault tolerant, integrated applications
- May do more to make it easy to build them
- The applications themselves must be fault tolerant

  And must therefore be replicated

  Master/monitor: detect faults and fail silent

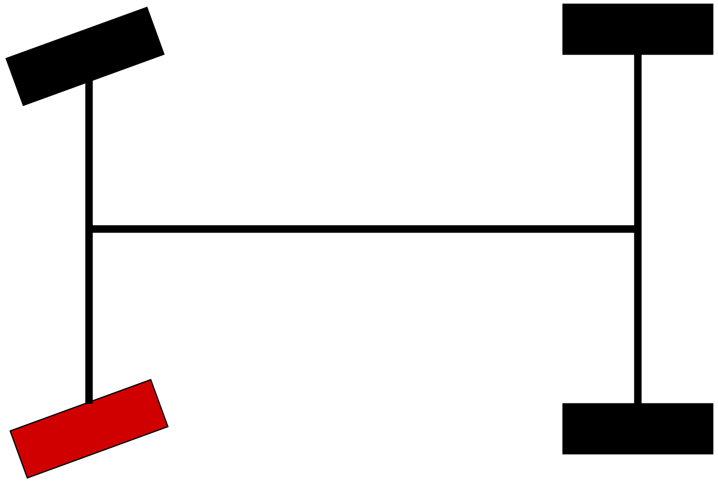
  Master/shadow: self-checking master shuts down, shadow takes over

  Compensation: survivors adjust their behavior to cover for failed component

  Masking: Triple modular redundancy and voting

Applications Need Consistent Knowledge

- 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

Replica Determinism

- All these strategies require that all nonfaulty redundant components have the same state
- That is, have received same sequence of messages
- So need more than "best efforts" message delivery
- Need consensus (aka. Byzantine agreement, interactive consistency)
- Under weakest fault hypothesis (Byzantine) this sets lower bounds (to tolerate t simultaneous faults):
  - $3t+1$ FCUs
  - $2t+1$ disjoint comms paths, or $t+1$ broadcast channels
  - $t+1$ rounds of information exchange

SOS and Asymmetric (Byzantine) Faults

- SOS = slightly out of specification
- Weak power supply or faulty line driver may send intermediate voltages
  - Neither digital 0 nor 1

  Some receivers may see 0, others 1, and others may reject
- Or may send weak (slow rise) edges
  - May look like 0 or 1, depending when sampled

  Some receivers may see 0, others 1, and others may reject
- Or clock drift may put edges at edge of sampling interval
- Or could go metastable
- All these can give rise to asymmetric reception
- Can reduce incidence of these with central hub
- But cannot eliminate at 10^{-10}

Consensus

SAFEbus: Honeywell implementation has an extra communication channel; uses method of Davies and Wakerly

SPIDER: Has redundancy inside central hub; uses variation on Draper FTP algorithm

TTA:

- Provides Group Membership as basic service (assumes benign fault modes)
- With Clique Avoidance as NGU backup (on asymmetric faults)
- Provides Draconian Consensus

FlexRay: no support

Must be provided by application: slow, increased latency;
Likely to get it wrong, or not realize you need it

Startup/Restart

- When a node has heard nothing for a while, sends a wakeup message
- Other nodes may do same thing at the same time
- Collision detection is unreliable
  - So always assume collision occurred
  - Back off for a node-specific interval and retry
  - Stay silent if no response to n retries
- Should get clean wakeup after some small interval
- Need to prove this is achieved, in the presence of faults

Formal Verification of These Algorithms

- Some completed, some in progress
- All performed using PVS
- Builds on work using earlier Ehdm system
- Several contributors apart from me: Friedrich von Henke and Holger Pfeifer (Ulm), Shankar (SRI), and Paul Miner (NASA)

Clock Synchronization

- Byzantine fault-tolerant clock synchronization algorithms are a major challenge for formal verification systems
    - Intricate combination of arithmetic and combinatorial reasoning
- Friedrich von Henke and I were the first to verify one (called interactive convergence) using Ehdm (TSE '93)
    - Subsequently repeated by Bill Young using Nqthm
- All later developments use Ehdm or PVS
- Lundelius-Lynch algorithm formally verified by Shankar (FTRTFT 92) and improved by Paul Miner (MS Thesis)
- Except TTA uses only 4 clock differences
- This variant was verified by group at Ulm (DCCA '97)
- But then lost in a fire

Clock Synchronization (ctd.)

- Don't want merely to recreate the lost Ulm treatment
- Also, satisfaction of mission requirements probably requires a hybrid fault model
  - This will allow formulation of properties when less than 4 good clocks remain, or more than a single fault arrives
- My proposal: verify Ulrich Schmid's treatment of clock synch. under hybrid fault model with link faults (DSN '01)
  - Independently interesting
  - Very fruitful recent collab'n with Schmid on consensus
- Then interpret TTA algorithm in this model with $n - 4$ "permanent" link faults to each node
- Hope PVS gains order of magnitude in efficiency over Ehdm

Verification of Window Timing

- Done by me (Tech Report)
- Straightforward and largely automatic (used as tutorial)

Verification of Group Membership

- Performed by Holger Pfeifer (Forte/PSTV 00)
- Based on an approach developed by me (CAV 00)
- Generates a diagram of possible "configuration" that conveys a lot of insight into the operation of the algorithm
- Proof is completely systematic, but not highly automated
  - Well. . . try it in your prover
- Proof only does validity and agreement
- Should be extended to self-diagnosis and rejoin

Clique Avoidance and Draconian Agreement

The most interesting remaining challenges

Clique Avoidance:

- Need to discover its properties/limits (i.e., what membership properties does it give up, what is its fault-hypothesis?)
- Some work by Merceron (ACSC 01)

Draconian Agreement: what is its fault-hypothesis?

Replica Determinism (Multiple Round Algorithms)

- If Draconian Consensus is inadequate, must resort to multiple-round algorithms
  - E.g., OMH(r) (FTCS 93) or ZA(r) (DCCA '95)
- Ulrich Schmid and Bettina Weiss have extended several consensus algorithms to a realistic hybrid fault model with link faults
- I have formally verified OMH(r) under this model
- Formalization revealed that all consensus algorithms need to be modified to ensure that "slightly faulty" senders maintain agreement with receivers
- Also led (through three iterations) to formulation in which node and link faults are independent
- Very fruitful use of formal verification in design loop

Interaction of Membership and Synchronization

- Each depends on the other
- How to break the circularity?
- There are assume/guarantee methods that do this
- Ken McMillan has a rule that is appropriate here: breaks the dependency by time
  - Membership at round t depends on synchronization up to round $t-1$
  - Synchronization at round t depends on membership up to round $t-1$

Interaction of Membership and Synchronization (ctd)

- McMillan's rule: H is a "helper" property, $\Box$ is the "always" modality of Linear Temporal Logic (LTL), and $p \triangleright q$ means that if p is always true up to time t, then q holds at time $t+1$ (i.e., p fails before q)

$$\frac{\begin{array}{c}\langle H\rangle X_1 \langle P_2 \triangleright P_1\rangle \\ \langle H\rangle X_2 \langle P_1 \triangleright P_2\rangle\end{array}}{\langle H\rangle X_1 || X_2 \langle \Box(P_1 \wedge P_2)\rangle}$$

- I have formally verified McMillans's rule
- Now plan to apply it to synchronization/membership
  - Here, X_1 is membership, X_2 is synchronization
- Holger Pfeifer is working on the same problem from a different direction

Other Issues Regarding Basic Algorithms

- Requirements on schedule/MEDL design
- Reintegration of failed nodes
- Mode changes and other reconfigurations
- Integration of all of these

Top-Level Issues

- Partitioning
  - The main issue for aircraft certification
  - It's what allows several "functions" to be integrated on single platform
  - 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
  - Need formalize this and verify it for TTA
- The Time-Triggered Model of Computation
  - I would like to have a crisp formalization of this (cf. "synchronous system")
  - Tom Henzinger has Giotto: a time-triggered language, but that's not what I'm after

Top-Level Issues (ctd)

- The Time-Triggered Model of Computation (ctd)
  - Hermann Kopetz has a whole philosophy of this
    - ⋆ E.g., Temporal firewalls, composability arguments
  - I want to give a formal account for this (cf. Paul Caspi's rational reconstruction for CriSys)
- Modular Certification
  - How to certify components separately
  - And glue them together
  - Certification differs from verification in that you have to take faults (hazards) seriously
  - Trying assume-guarantee approach, based on normal and (multiple) abnormal assumptions and guarantees
  - May help explain Perrow's concerns, and Kopetz' recommendation for elementary interfaces

Utility?

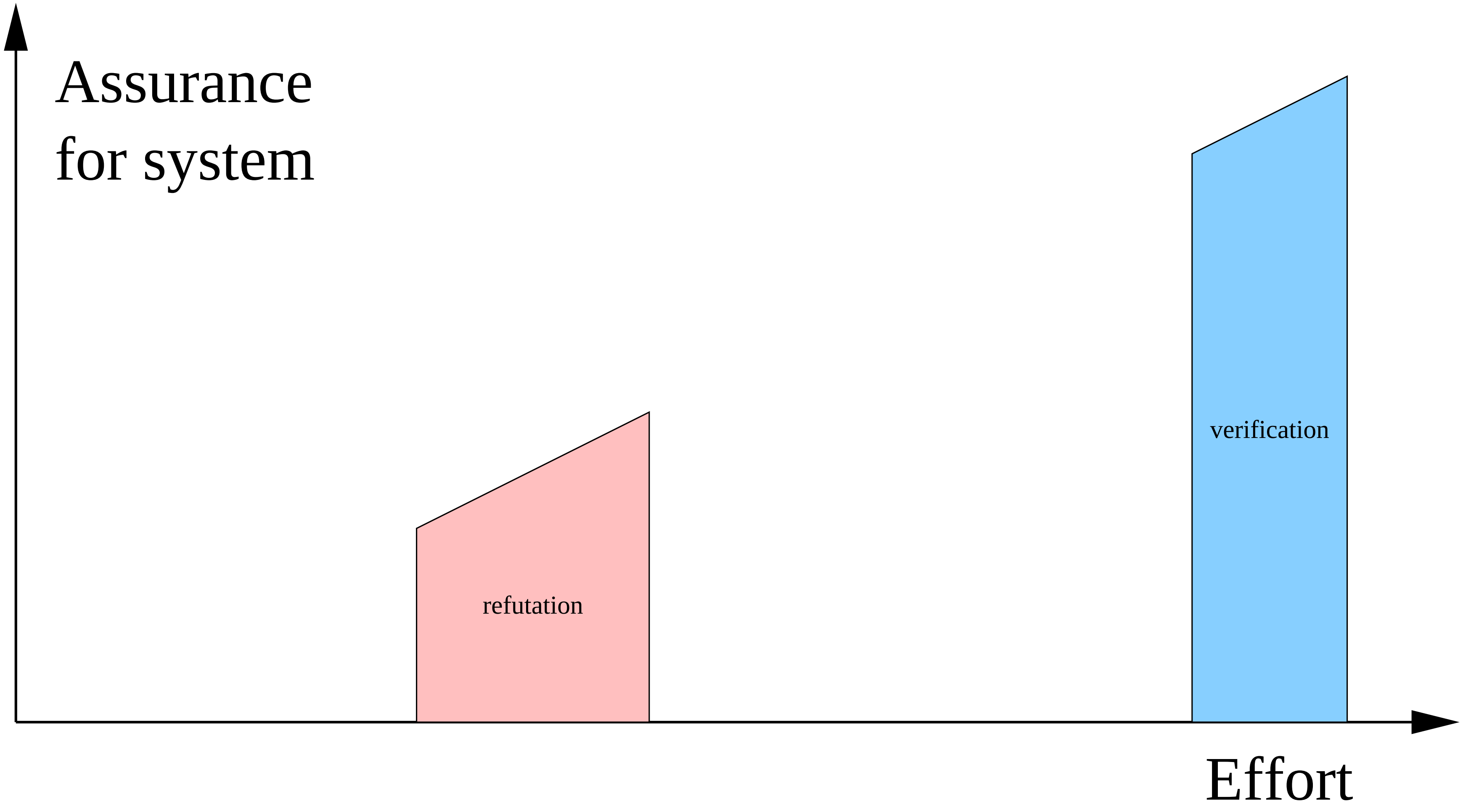
- The completed verifications will have obvious utility in certification
- But the main benefits are sharpened statements of assumptions and properties
- And clarification of interactions and interdependencies among the algorithms
- Stimulates useful dialog with the designers of TTA
- And provides education for potential users of TTA
- Severe test of PVS capabilities and automation

Next Steps

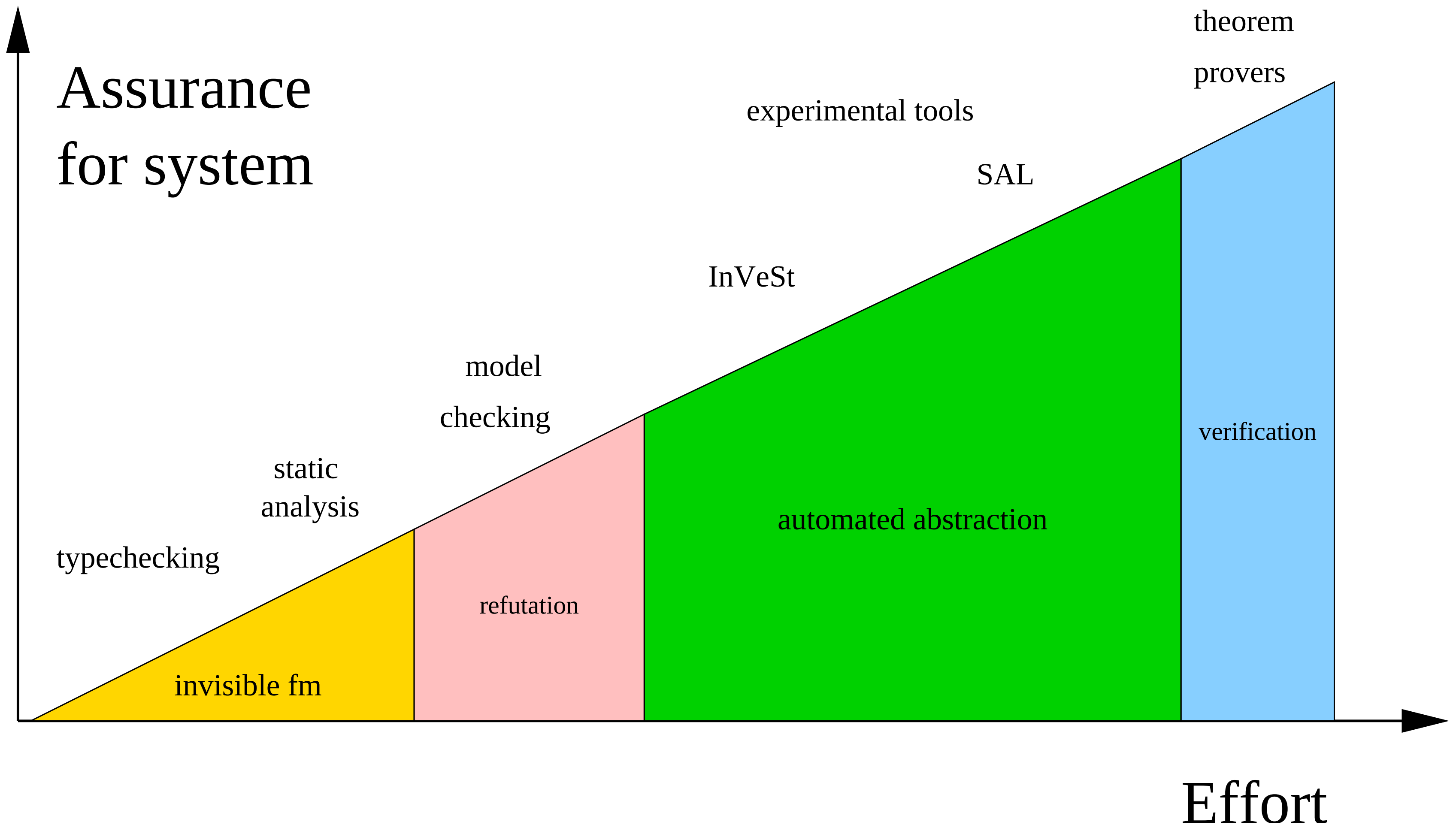
- Want to support developers of applications to run on TTA
- Should be able to verify their designs
  - Expressed in e.g., Lustre or Simulink
- And their transformation into fault-tolerant implementations running on TTA
- Formalization needs to be largely transparent
- And verification must be largely automatic
  - Need test vectors as well as formal proofs
- We cannot do all of this: concentrate on providing basic toolkits for others

The Wall of Formal Verification

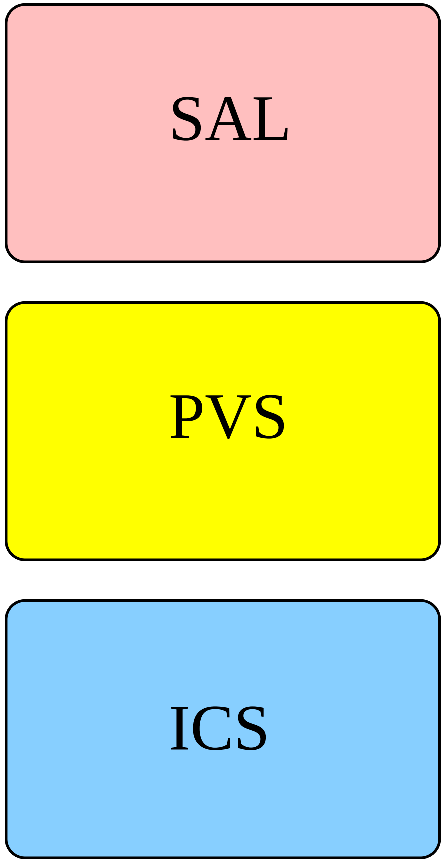
The Islands of Refutation and Verification



A Smooth Slope of Formal Methods



What We Are building



ICS = Integrated Canonizer-Solver (= ICanSolve)

ICS

- The goal is an integrated, modular set of decision procedures

  Integrated: they decide the combination of their theories

  Modular: separate decision procedures for each theory

- Shostak's procedure (used in STP, Ehdm, PVS) is integrated, but incomplete, and not modular

- Also want to extend to nonlinear arithmetic, other new theories, provide quantifier elimination, soundness guarantees

ICS Theory

- Ruess and Shankar (LICS 01) gives (for the first time) a version of Shostak's method that is sound and complete for the combination of equality and uninterpreted function symbols
- There is a modular extension to other theories in which equality is the only predicate symbol (modular like Nelson-Oppen, but efficient like Shostak)
  - Bitvectors, tupling and projection, updates
- And another extension to theories with additional predicate symbols (submitted to FroCoS)
  - Linear arithmetic, predicate calculus

ICS Pragmatics

- Decision procedures should be available standalone
  - To encourage invisible formal methods
- As well as suitable for use in PVS and other theorem provers
  - So have richer interface than other decision procedures
- Core Shostak method (almost) formally verified in PVS by Jonathan Ford
- Share PVS brand
- Plan to license ICS for commercial use

ICS Implementation

- Released ICS in July 2001: `http://www.ICanSolve.com`
  - This version is not modular
  - Used at IRISA for MC/DC test generation, Intel etc.
- New (modular) version being tested in PVS 3.0

SAL

- Can apply a lot more automation if you know you are dealing with a state machine rather than arbitrary math
- Could just recognize a stylized use of PVS and add the automation to PVS (first version of InVeSt did this)
- But there are lots of languages and tools for state machines so get more synergy if the state machine representation is conveniently accessible
- Our representation is the SAL language
  - Does for state machine (transition relation) languages what PVS did for specification languages
  - Coherent combination and generalization of many ideas
  - Easy to translate in and out of SAL
  - And quite a nice language in its own right

SAL Tools

- It's intended that SAL should provide access to external tools (e.g., model checkers, abstractors) via translation
  - XML used as the interchange format
  - The SAL language is actually defined by its XML DTD
  - But there is a concrete syntax with parser and unparser to/from XML
- Also want to provide building blocks and open environment so that users can tailor and customize tools for their own needs
- SAL uses a lot of PVS infrastructure (but can be used without PVS)
  - Typechecker built on that of PVS 3.0
  - May generate TCCs (though these can be invariants, rather than pure formulas)

Exploring SAL Specifications

- A good explicit-state model checker is essential for exploring transition-relation specifications
  - Murϕ is no longer supported
  - Spin has a truly horrible language; depth-first search gives enormously long counterexamples
- Want customizable search strategies (e.g., for reductions based on symmetry, or partial order, or to home in on troublesome parts of the state space), or to evaluate functions rather than predicates
- And visualization of state space and counterexamples
- Also want simulation, debugging

SAL Execution Toolkit SALENV

- A open toolkit for building exploration environments for SAL
  - Built by writing little Scheme programs over an API for statespace traversal
- The SAL model checker is constructed in this way
  - Does full LTL model checking (very efficiently)
  - Can do depth-first search, breadth-first, and various combinations
  - Search strategy can be customized by the user (by writing a short Scheme program)
  - Also provides nice visualizations
- Will be released in a few weeks time
- Allows very nice treatment of DEOS scheduling
  - Simple customization computes maximum blocking time

Summary

- TTA is the last best hope for introducing rational fault-tolerance to distributed embedded systems
  - Displacing homespun solutions
- Analysis of its algorithms is a challenging and interesting problem for formal verification
  - But only needs to be done once
- Formalizing the computational model and properties presented to its client applications is crucial
- Can then bring formalization and verification to those clients
- In the form of "disappearing formal methods"