Formal Verification
For Fault-Tolerant Algorithms

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA

Goals

- To demonstrate the power of formal methods (and Ehdm and PVS in particular)
  - These algorithms are among the hardest and most subtle
  - Drove much of the development of our tools
- To demonstrate that formal verification is practical
  - Need to make verification of these algorithms routine
  - Requires finding an effective methodology
- To demonstrate that formal verification is useful
  - Find bugs in famous/critical algorithms or their proofs
  - Use formal verification as a tool for design exploration to help create improved algorithms

Fault Tolerant Algorithms

Designed and analyzed with respect to

- System model
  - Synchronous
  - Asynchronous (and restricted forms thereof)
  - Self-stabilizing
- Communications topology (fully connected, ring etc.)
- Fault model
  - Message faults
    - Loss
    - Duplication
    - Reordering
    - Timing
  - Processor faults
    - Stopping faults
    - Byzantine faults
    - Refinements of the above

Classical Problems

- Clock synchronization
- Consensus (many variants)
  - Byzantine Agreement, Interactive Consistency, The Commit Problem
- Atomic Broadcast
- Leader Election
- Group Membership
- Communications protocols

Consensus Problems

- Some/all processors start off with a private value
- Goal is to get all to decide on some common value

Agreement: all nonfaulty processors decide on the same value

Validity: depends on the problem

Termination: two kinds

- **Weak:** only if no faults
- **Strong:** always (aka. nonblocking)

Consensus Problems: Byzantine Agreement

- A distinguished process (the General) has a value, the others (The Lieutenants) must agree on that value
- Validity: if the General is nonfaulty, the agreed value must be his actual value
- System model: synchronous
  - ⋆ Unsolvable for asynchronous with even a single fault
- Fault models:

  Byzantine: no assumptions

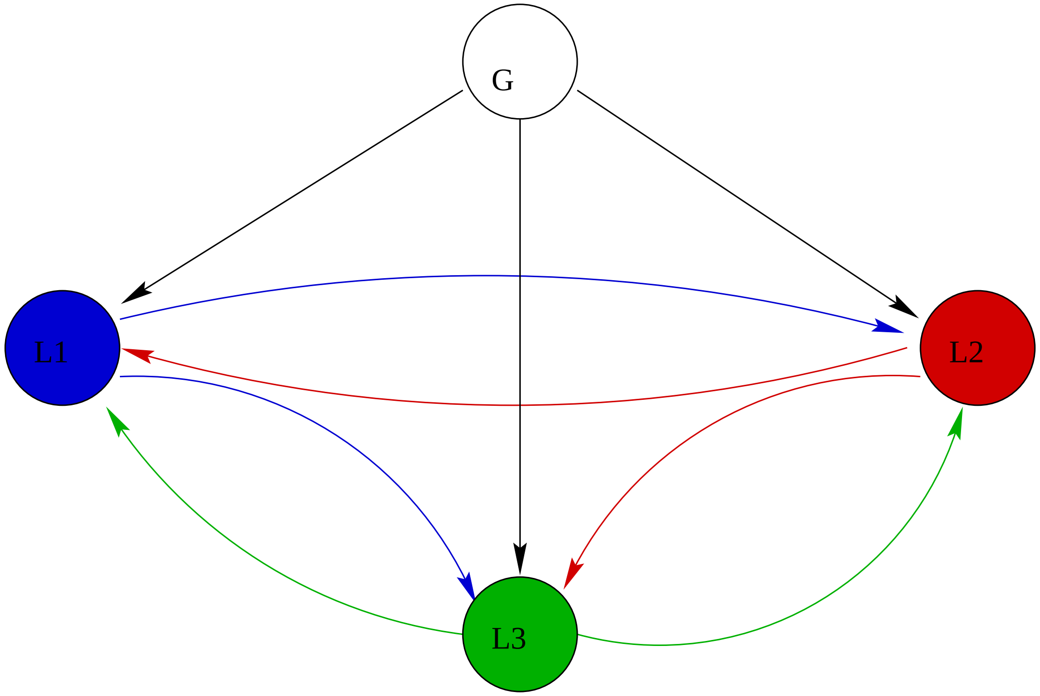
  Hybrid: Byzantine, symmetric, manifest

  Authenticated Byzantine: cannot forge messages from another processor
- Algorithms: oral messages (OM), signed messages and their variants, EIG graph

OM(1) in Pictures
(Four-Processor One-Round Oral Messages)

Withstands one Byzantine fault

- No r-round algorithm can withstand more than r faults
- No n-processor OM algorithm can withstand more than $(n-1)/3$ faults



Synchronous Systems

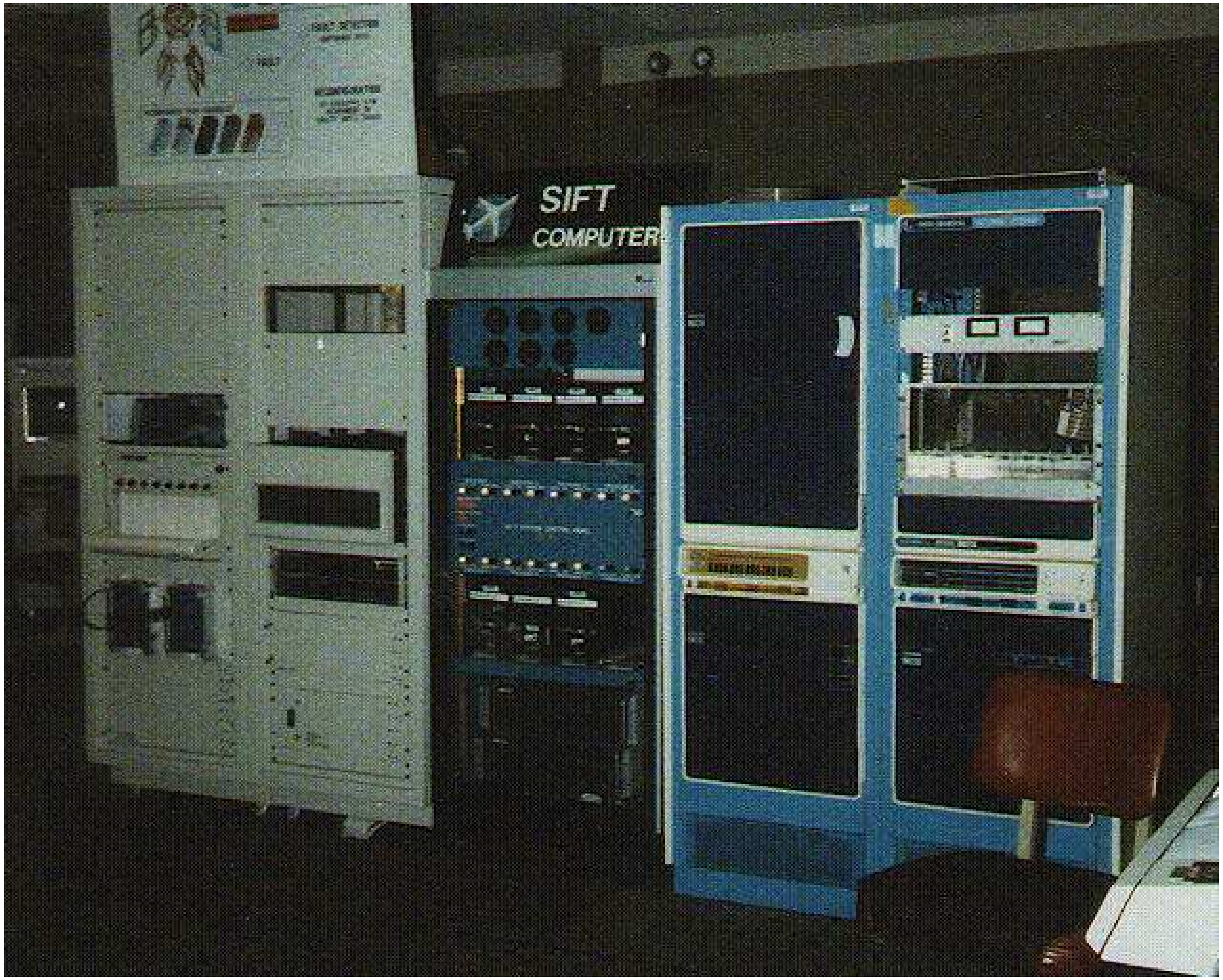
- Known upper bounds on
  - Time required for nonfaulty processors to perform operations
  - Messages delays in the absence of faults
- Assumptions are valid for embedded real-time control systems
- The classical problems of fault-tolerant distributed systems can be solved under these assumptions
  - Consensus (Byzantine Agreement)
  - Group Membership
  - Etc.

  Whereas they cannot be solved in asynchronous systems
- We have focused chiefly on synchronous systems

Origins and Applications

- The synchronous system model, the classical problems, and many of the algorithms came from efforts to build ultra-reliable systems for digital flight control
- Notably SIFT (Software Implemented Fault Tolerance), built here in the late 70's
- Based on State Machine Replication
- A lot of the early work on formal methods at SRI was funded in support of SIFT
- Industrial developments included FTP (Draper), MAFT (Allied Signal), TTA (TTTech [Hermann Kopetz]), SAFEbus (Honeywell)
- Sadly, most deployed systems use homespun designs

The SIFT Computer



The Byzantine Generals

Leslie Lamport, Marshall Pease, Rob Shostak

Formal Synchronous System Model

- Algorithms execute in a series of *rounds*, numbered $0, 1, \ldots$
- Each round has two *phases*

Communication Phase: each processor sends messages to (some or all) other processors

- Messages sent, and where to, depend on current state
- $msg_p(s, q)$ is the message sent by p to q when p's state is s

Computation Phase: each processor updates its state

- New state depends on previous state and on messages received during communication phase
- $trans_p(s, i)$ is p's new state, when its current state is s and the set of messages received is i

Synchronous System Model: Operation

- Processors operate in lockstep
  - All perform the communication phase of the current round
  - Then the computation phase
  - Then move on to the next round, and so on

  Byzantine faults: know nothing about messages sent or received by faulty processors, nor their state changes

  Stopping faults: a faulty processor may send some/all/none of its messages in the round it goes faulty, makes no state change, silent thereafter

- Computation and message transmission happen instantaneously and atomically
- Processors are perfectly synchronized and perform their actions simultaneously

Example: OM(1) Algorithm for Consensus

Round 0:

Communication Phase: The transmitter sends its value to the receivers; receivers send no messages

Computation Phase: Each receiver stores the value received from the transmitter in its state

Round 1:

Communication Phase: Each receiver sends value stored in its state to all other receivers; transmitter sends nothing

Computation Phase: Each receiver decides on the majority value among those received from the other receivers and that (stored in its state) received from the transmitter

Synchronous Algorithms as Functional Programs

- Formal verification of a synchronous algorithm can be quite difficult in its classical formulation
  - Rounds and phases have an operational character that is awkward to represent in formal logic
  - Functional programs are much easier
- So establish a systematic transformation between synchronous systems and functional programs.
- Describe by example: OM(1)

Specification of OM(1) as a Functional Program

First step is to model sending of messages

- Function $send(r, v, p, q)$ represents sending of a message with value v from processor p to processor q in round r
- Value of the function is the message received by q
- If p and q are nonfaulty, this value is v:

$$nonfaulty(p) \wedge nonfaulty(q) \Rightarrow send(r, v, p, q) = v,$$

- Otherwise it depends on the fault modes considered
    - Here it is left entirely unconstrained (Byzantine fault model)

Specification of OM(1) as a Functional Program (ctd. 1)

T is the transmitter, v its value, and q an arbitrary receiver

Round 0, communication phase:

T sends v to each q:

$$send(0, v, T, q)$$

Round 0, computation phase: do nothing

(instead of storing value received, q sends it to itself in next phase)

Round 1, communication phase:

Each q sends the value received in the first round to each receiver p (including itself):

$$send(1, send(0, v, T, q), q, p)$$

Specification of OM(1) as a Functional Program (Ctd. 2)

Round 1, computation phase:

p gathers all the messages just received and votes them

- "Gathers" represented by λ-abstraction:

$$\lambda q : send(1, send(0, v, T, q), q, p)$$

  (i.e., a function that, when applied to q, returns the value that p received from q)

- $maj(caucus, votes)$ takes a function *votes* from processors to values, and returns the majority value if one exists, among the processors in *caucus*; otherwise some functionally determined value
- Then p's decision is given by

$$maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p))$$

  where *rcvrs* is the set of all receiver processors.

Specification of OM(1) as a Functional Program (Ctd. 3)

Represented as the (higher-order) function $OM1$:

$$OM1(T, v)(p) = maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p))$$

$OM1(T, v)(p)$ is the decision reached by each receiver p when the (possibly faulty) transmitter T sends the value v

Properties required

Agreement: nonfaulty receivers agree, even if faulty transmitter sends different values

$$nonfaulty(p) \wedge nonfaulty(q) \Rightarrow OM1(T, v)(p) = OM1(T, v)(q)$$

Validity: when the transmitter is nonfaulty, all nonfaulty receivers get the correct value

$$nonfaulty(T) \wedge nonfaulty(p) \Rightarrow OM1(T, v)(p) = v$$

Verification of OM(1) as a Functional Program

- The great advantage of this representation is that it is expressed in regular (higher-order) logic and highly automated theorem proving can be used to verify algorithm properties
- For example: OM(1)

  Agreement: PVS can prove the $n = 4$ instance automatically, requires just eight commands to prove the general case

  Validity: PVS can prove the general case automatically
- It would be much harder to mechanically verify original representation of OM(1) as a synchronous algorithm
- And verification of an event-driven formulation of OM(1) by Lamport and Merz required *significant* effort

Our Verifications of OM-Like Algorithms

- The general case OM(r)
- Extension to hybrid fault model (tolerates more simple faults)
  - Found bug in MAFT algorithm description (MAFT is ok)
  - Developed and verified corrected algorithm
  - Verified version for Draper FTP
  - Developed and model checked variants that use authentication
  - Fault diagnosis algorithms
- These are the most efficient algorithms known for the measure of practical importance (number of faults tolerated in the one-round case)
- Massive case analysis is unreliable when done by hand: really exploits the power the of formal verification

Other Verifications We Have Done

- MJRTY linear-time majority vote
- Byzantine fault-tolerant clock synchronization
  - Interactive convergence algorithm (ICA)
  - General class of all such algorithms (also at NASA)
  - ICA for hybrid fault model
- State machine replication with transient recovery
- Time-triggered implementation of synchronous model
- TTA-like group membership (model checked only so far)
- Three-phase commit

Consensus Problems: Three Phase Commit

- All processes have an opinion: either *commit* or *abort*
- Validity for abort: if any processor has abort initially (or is dead), all must decide abort
- Validity for commit: if all have commit initially and no failures, all must decide commit (stronger: should commit whenever able)
- System model: synchronous
- Fault model: stopping faults
- Algorithms: two-phase commit (weak termination), three-phase commit (strong termination)

Implementations of Algorithms for Synchronous Systems

Have to deal with the reality that events are not instantaneous, atomic, and simultaneous

- Communications and computations take time
  - Timeouts needed to detect failed communications
- Processors are not perfectly synchronized
- And run at different rates

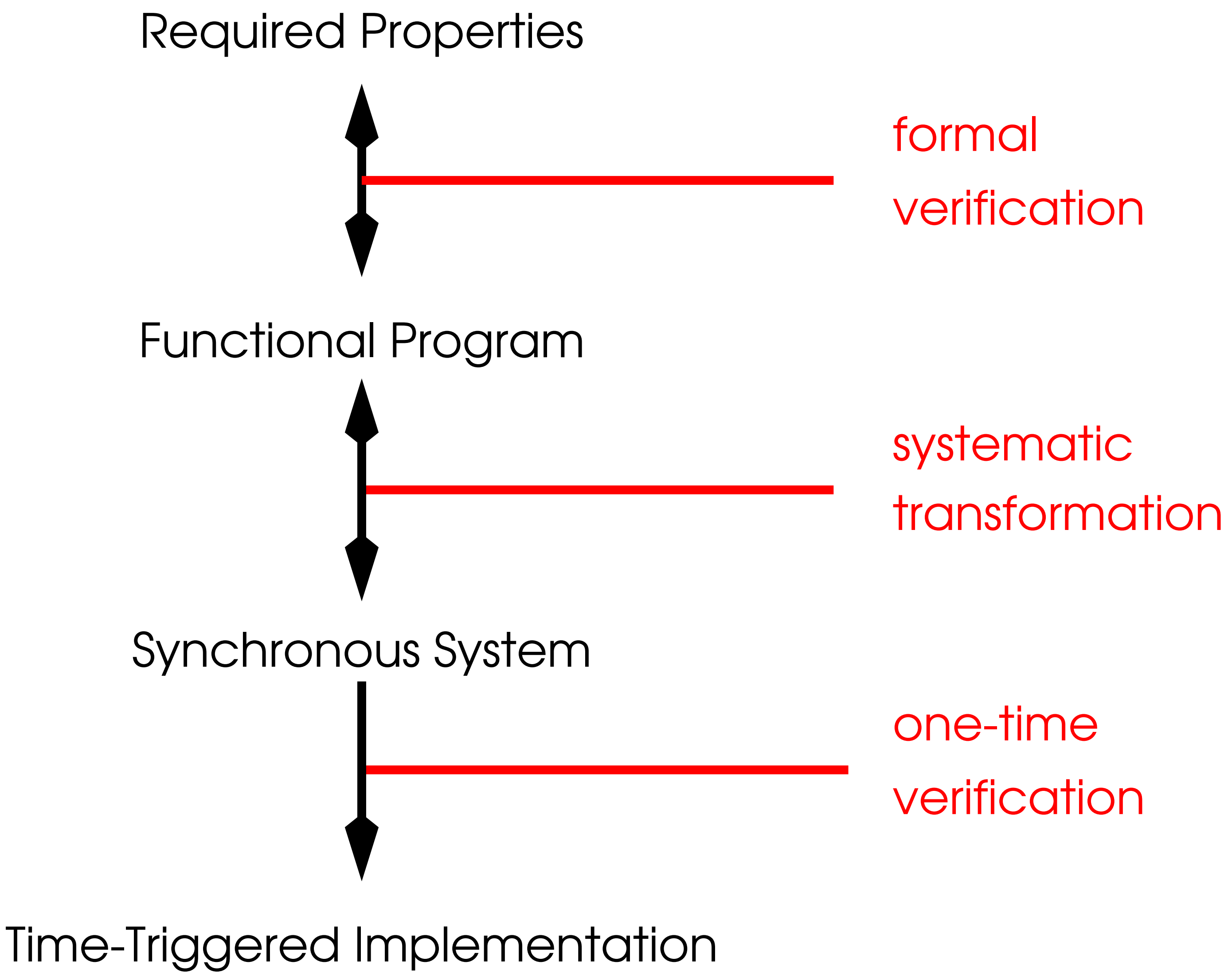
Two approaches

Event triggered: processors react to incoming messages; set timeouts on outgoing messages

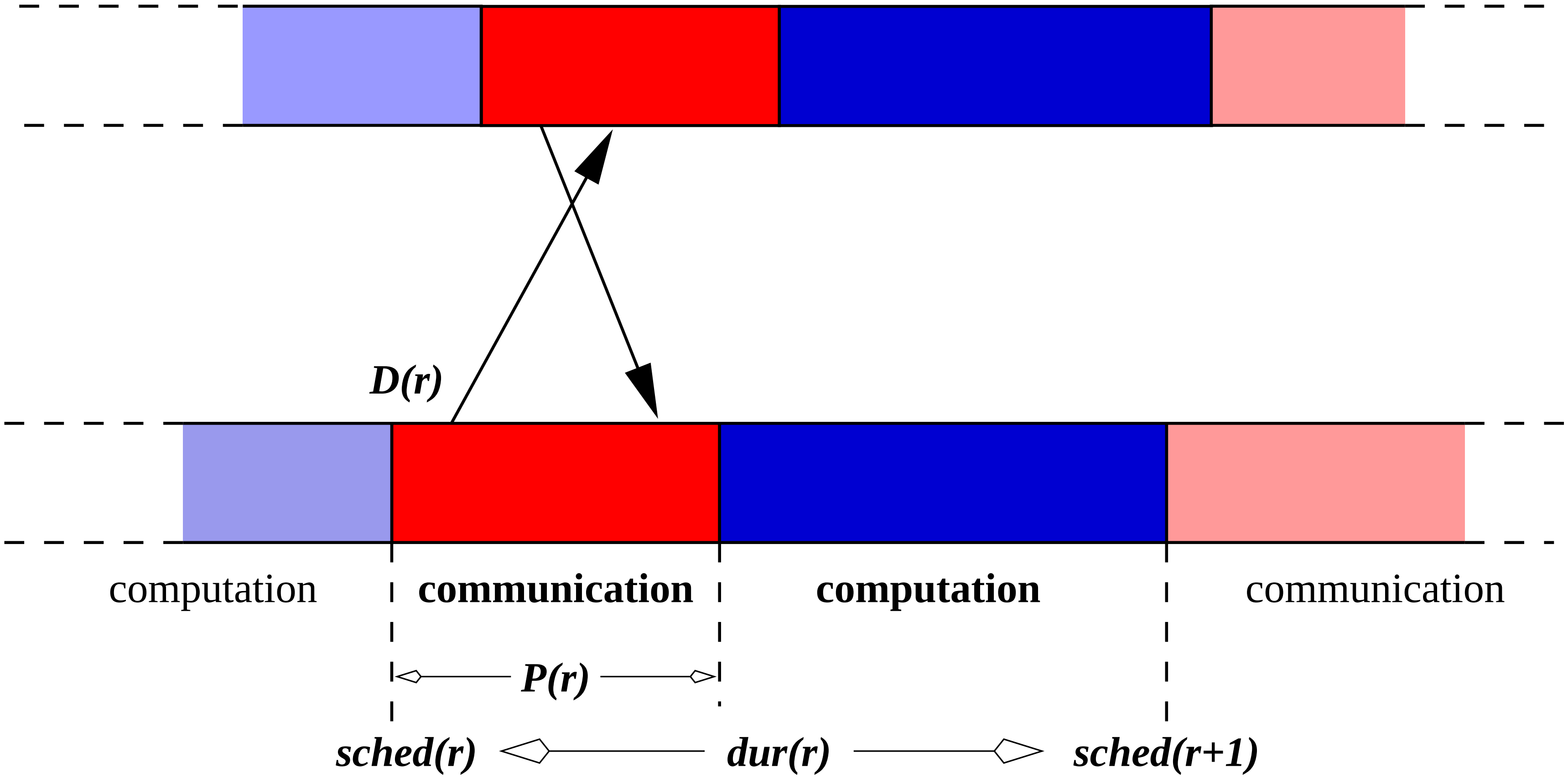
Time triggered: processors perform actions according to a common schedule, driven by their own internal clocks

- Preferred for critical app'ns: SAFEbus, TTP, Shinkansen

Verification Path from Algorithm to Implementation



Time-Triggered System Model



Issues in Verifying the Time-Triggered Implementation

- Processor clocks are not perfectly synchronized
  - One processor may send message before or after another one expects it; may not even be on the same round
  - Therefore require a bound on synchronization skew
  - Can be ensured by clock synchronization algorithms
- Processor clocks do not run at the same rate
  - Durations of the phases may differ on different processors
  - Therefore require that good processors' clocks run at rates within some bound of each other
- Unpredictable delays in message transmission
  - Message may arrive after comms phase has ended
  - So require upper bound on nonfaulty message delays
- Need to arrange pacing and timeouts so that it all works

Asynchronous Algorithms

- The asynchronous system model is appropriate for wide area networks (cannot provide an upper bound on message/processor delays)
- Nothing resembling consensus can be achieved in the asynchronous system model in the presence of even trivial faults
  - Cannot distinguish a slow processor from a failed one
  - Basically rules out all fault tolerant algorithms with the exception of communications protocols
- Chief issue in verification of communications protocols seems to be invention/generation of auxilliary invariants

Pretending Synchrony

- Imposing timeouts and pretending the system is really synchronous does not work
  - Set the timeout large and the system will be slow to detect faults
  - Set the timeout small and you will get many failures
    - ⋆ Which will overwhelm the fault tolerance of your synchronous algorithm
  - Problem is that synchronous algorithms do not maintain their safety properties when the synchrony assumption is violated
- Therefore need "somewhat synchronous" models

Somewhat Synchronous System Models

Partially synchronous: there is an upper bound, but you don't know what it is

Timed asynchronous: system alternates between periods of stability (synchronous) and instability (no synchronous)

- Algorithms are safe and live during stable periods, safe during unstable periods

Asynchronous with failure detectors: (abstract timeouts)

- Can accuse processors of having failed; may be able to retract accusations
- Various properties, such as: eventually no good processor is accused and all failed ones are
- Algorithms work with failure detector of suitable class
- Hierarchy of problems by failure detector required

Challenges

- Improve automation in PVS so verification of synchronous algorithms is truly routine (mainly quantifier instantiation)
- Try to automate more of the verification of clock synchronization (timed automata?)
- Develop methods (abstraction and invariant generation) to make verification of communications protocols routine
- Develop methodology and automation for verifying timed-asynchronous and asynchronous with failure detector algorithms comparable to what we have for synchronous algorithms
- Demonstrate verification of a stack of fault-tolerant protocols