DCCA 97

Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms

John Rushby

Computer Science Laboratory
SRI International
Menlo Park CA USA

Overview

- Many fault-tolerant algorithms are relatively easy to understand and to verify in an abstract, untimed formulation
- But verifications of implementations, with all their timing parameters, are quite complex
- So split the problem into two parts
  - Verify abstract algorithm for an untimed synchronous system model
    - Must be done for each algorithm
    - Relatively easy—and can itself be split into two parts
  - Verify time-triggered implementation of the untimed model
    - Can be done once-and-for-all
    - Is the main topic of this paper
- Provides simple path from verified design to implementation

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
- Focus here is exclusively on synchronous systems

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
- Computation and message transmission happen instantaneously and atomically
- Processors are perfectly synchronized and perform their actions simultaneously
- No sense of real time (hence *untimed* system model)

Example: Oral Messages Algorithm for Consensus, OM(1)

Transmitter processor has a value to be communicated reliably to three or more *receivers* in the presence of one arbitrary fault

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

Implementing 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

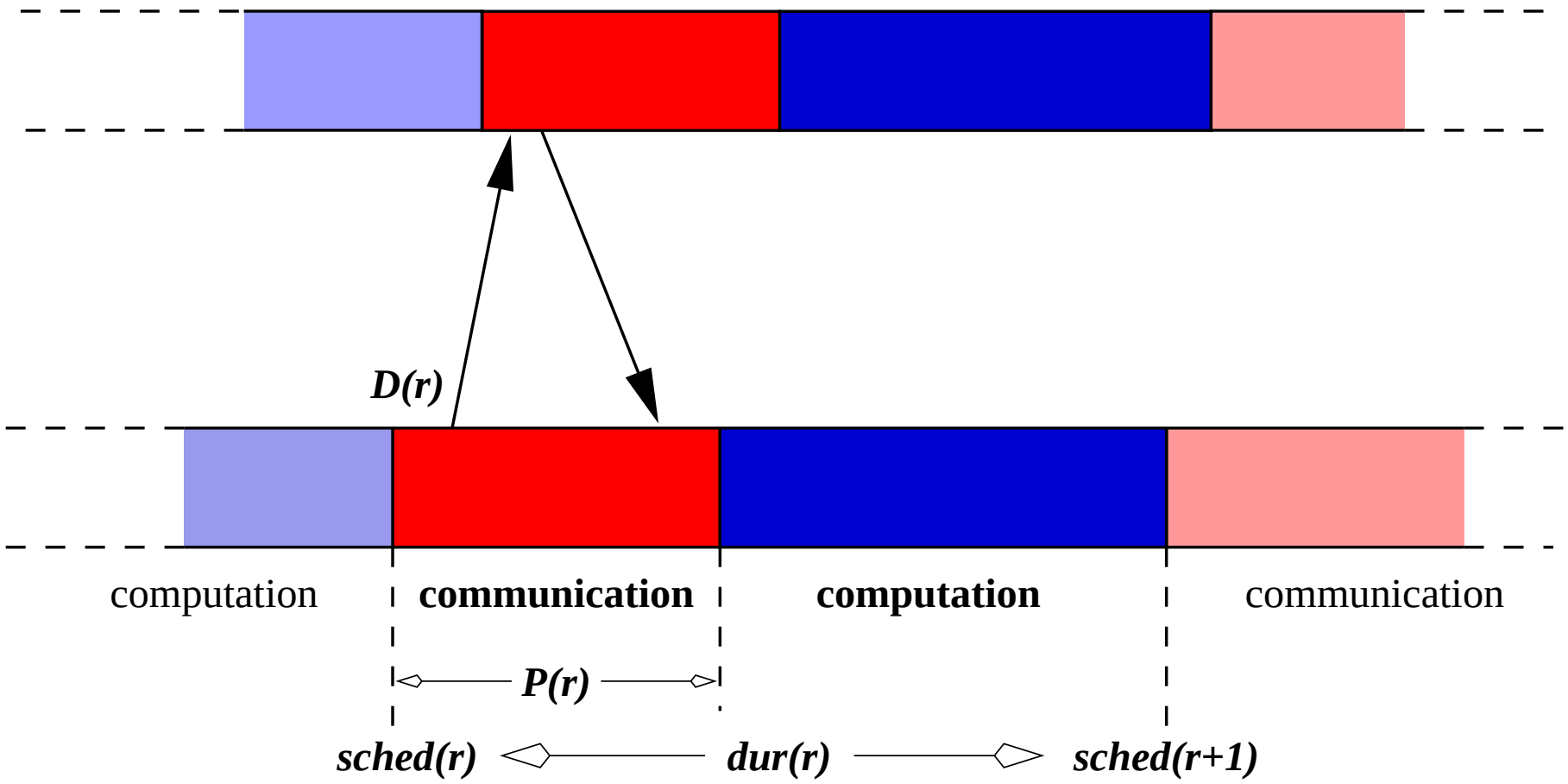
Two approaches

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

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 communications phase has ended
  - Therefore require upper bound on nonfaulty message delays
- Need to arrange pacing and timeouts so that it all works

Clocks

- Each processor has a clock, that reads *clocktime*
  - Clocktimes denoted by upper-case letters (T, Σ etc.),
- There is an abstract, universal, time called *realtime*
  - Realtimes denoted by lower-case letters (t, ρ etc.)
- $C_p(t)$ is the clocktime on p's clock at realtime t

Clock Assumptions

Monotonicity: Nonfaulty clocks are monotonic increasing functions:

$$t_1 < t_2 \Rightarrow C_p(t_1) < C_p(t_2)$$

Clock Drift Rate: Nonfaulty clocks drift from realtime at a rate bounded by a small positive quantity ρ (typically $\rho < 10^{-6}$):

$$(1 - \rho)(t_1 - t_2) \leq C_p(t_1) - C_p(t_2) \leq (1 + \rho)(t_1 - t_2)$$

Clock Synchronization: The clocks of nonfaulty processors are synchronized within some small clocktime bound Σ:

$$|C_p(t) - C_q(t)| \leq \Sigma$$

Achieving these requires care in implementation, since some clock synchronization algorithms violate monotonicity. However, monotonicity can always be achieved, with no loss of precision

Time-Triggered System Model

Each processor

- Starts round r at clocktime $sched(r)$ by its local clock
- Sends its messages $D(r)$ clocktime units into the round
- Starts computation phase $P(r)$ clocktime units into the round
  - So duration of r'th communication phase is $P(r)$
- Finishes the round after $dur(r)$ clocktime units
  - $dur(r) = sched(r+1) - sched(r)$
  - So duration of r'th computation phase is $dur(r) - P(r)$

Additional Assumption

Maximum Delay: messages are received within δ realtime units

Constraints

1. $dur(r) > P(r) > D(r) > 0$
   - The communication phase is of positive duration
   - The computation phase starts after the messages are sent and is of positive duration

2. $D(r) \geq \Sigma$
   - The delay before messages are sent is greater than the clock skew (so messages do not arrive while the receiving processor is still in the previous round)

3. $P(r) > D(r) + \Sigma + (1 + \rho)\delta$
   - The communication phase must last long enough that all messages have time to reach their destination processor while it is still in its communication phase

Fault Model

- Faults are modeled as changes in the msg_p and $trans_p$ functions
- Will prove that untimed model and time-triggered implementation have same behavior, given same msg_p and $trans_p$ functions, for *any* such functions
- Thus, if an algorithm is proved fault tolerant in the untimed model with respect to a fault model that can be expressed as perturbations to the msg_p and $trans_p$ functions, then implementation inherits those fault-tolerance properties
- However, implementation admits new faults
  - Loss of clock synchronization
  - Shared buses (babbling idiot fault mode)

  Must take care to minimize these and to ensure that those not masked are transformed into simplest of the modeled faults

Correspondence between Rounds

- Want to ensure that untimed synchronous model and its time-triggered implementation produce same behavior
  - i.e., prove that state of the system at the start of each round is the same in both model and implementation
  - But when does a round start in the implementation?
- Define the *global start* for round r to be the realtime $gs(r)$ when the processor with the slowest clock begins round r
- Then $gs(r)$ satisfies the constraints:

$$\forall q : C_q(gs(r)) \geq \mathit{sched}(r),$$

and

$$\exists p : C_p(gs(r)) = \mathit{sched}(r)$$

(intuitively, p is the processor with the slowest clock)

Correctness

Theorem: Given the same initial states and same msg_p and $trans_p$ functions, the state of each processor in the untimed synchronous system at the start of the r'th round is the same as its state at time $gs(r)$ in the time-triggered implementation

Proof: By induction—see paper for details

Formal Verification: Has been formally specified and mechanically verified using SRI's verification system, PVS

- Formal verification took about a day
- Allowed easy generalization from fixed offsets D and P to round-specific $D(r)$ and $P(r)$
- See long version of paper—available on the Web at `http://www.csl.sri.com/dcca97.html`
- PVS specification and proof files available there also

Synchronous Algorithms as Functional Programs

- Theorem establishes correctness of time-triggered implementations for synchronous algorithms
- But formal verification of a synchronous algorithm can still be quite difficult
  - 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 $\mathit{send}(r, v, p, q)$ 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:

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

- 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:

$$\mathit{send}(0, v, T, q)$$

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):

$$\mathit{send}(1, \mathit{send}(0, v, T, q), q, p)$$

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:

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

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

- $\mathit{maj}(\mathit{caucus}, \mathit{votes})$ 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

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

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 $OM1$:

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

$OM1(T, v)(p)$ 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

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

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

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

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)

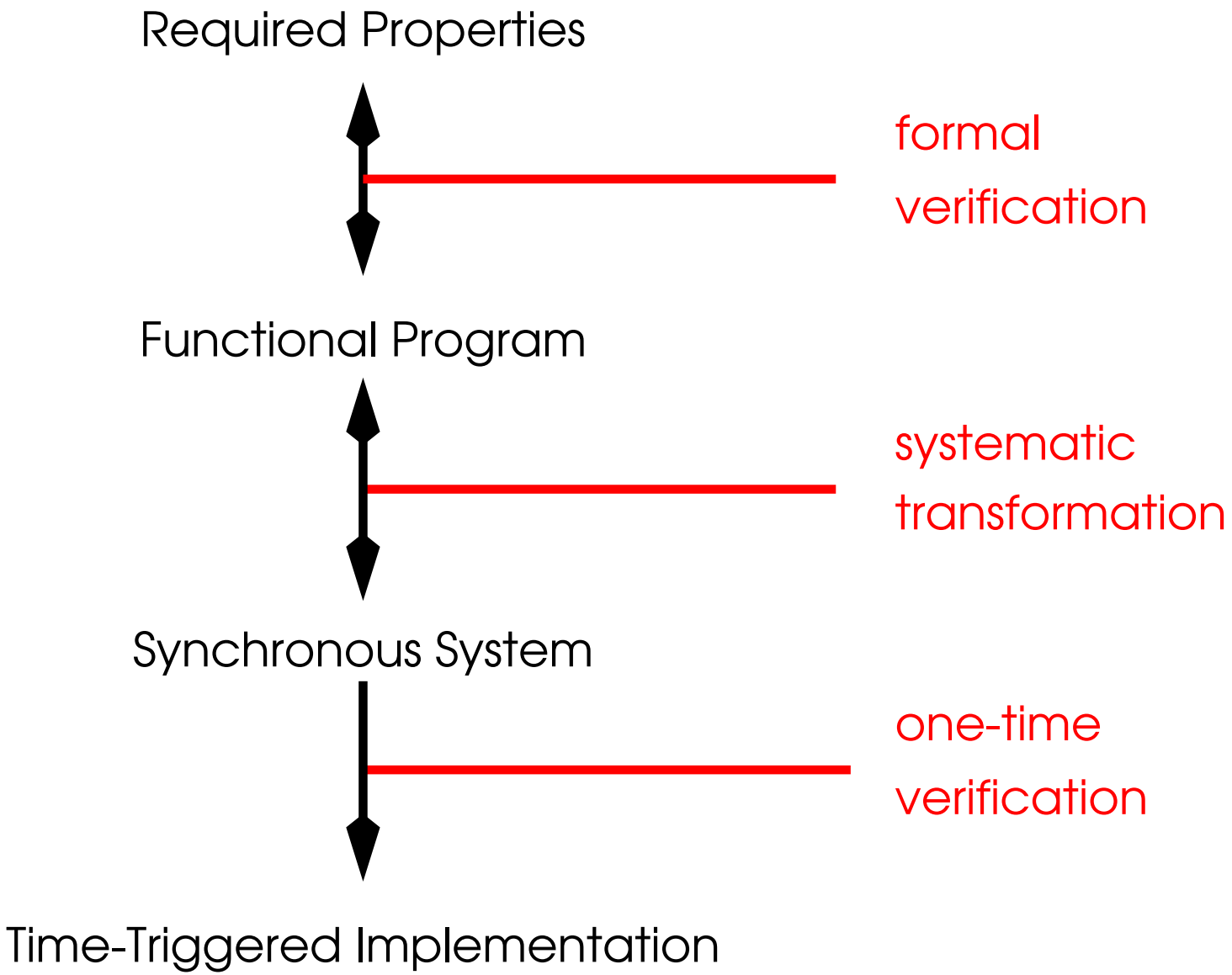
  Agreement: PVS can prove the $n = 4$ instance automatically, requires just eight commands to prove the general case

  Validity: PVS can prove the general case automatically

- It would be much harder to do mechanized formal verification for 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

Summary


Required Properties
formal
verification
Functional Program
systematic
transformation
Synchronous System
one-time
verification
Time-Triggered Implementation

Conclusions and Future Work

- This approach reduces and systematizes the effort required to verify (some) time-triggered algorithms
- Simplicity of the formulation and proof of the theorem suggests that time-triggered systems are the natural realization of the synchronous system model
- Future work:
  - Formalize and verify the transformation between synchronous systems and functional programs
  - Apply to more difficult algorithms
    - Currently working on a group membership algorithm similar to that in TTP