Reduction Revisited: Verifying Round-Based Distributed Algorithms

Stephan Merz

INRIA Nancy & LORIA

joint work with Bernadette Charron-Bost, LIX & CNRS

MPC 2010
June 23, 2010

Example: mutual exclusion algorithms

integer turn = 0;
boolean req0, req1 = false;

process P0		process P1
loop		**loop**
nc_0: **skip**;		nc_1: **skip**;
rq_0: req0 := true;		rq_1: req1 := true;
ps_0: turn := 1;	$\|$	ps_1: turn := 0;
wt_0: **await** $\neg$req1 $\vee$ turn = 0;		wt_1: **await** $\neg$req0 $\vee$ turn = 1;
cs_0: **skip**;		cs_1: **skip**;
ex_0: req0 := false;		ex_1: req1 := false;
endloop		**endloop**

- Critical section can be abstracted to atomic step

Example: mutual exclusion algorithms

$$
\begin{array}{ll}
\textbf{integer} & \text{turn} = 0; \\
\textbf{boolean} & \text{req0, req1} = \text{false};
\end{array}
$$

$$
\begin{array}{l}
\textbf{process } \text{P0} \\
\textbf{loop} \\
\quad \text{nc}_0\text{:} \;\; \textbf{skip}; \\
\quad \text{rq}_0\text{:} \;\; \langle \text{req0} := \text{true}; \\
\quad \qquad\quad \text{turn} := 1; \rangle \\
\quad \text{wt}_0\text{:} \;\; \textbf{await } \neg\text{req1} \lor \text{turn} = 0; \\
\quad \text{cs}_0\text{:} \;\; \textbf{skip}; \\
\quad \text{ex}_0\text{:} \;\; \text{req0} := \text{false}; \\
\textbf{endloop}
\end{array}
\quad \Bigg\| \quad
\begin{array}{l}
\textbf{process } \text{P1} \\
\textbf{loop} \\
\quad \text{nc}_1\text{:} \;\; \textbf{skip}; \\
\quad \text{rq}_1\text{:} \;\; \langle \text{req1} := \text{true}; \\
\quad \qquad\quad \text{turn} := 0; \rangle \\
\quad \text{wt}_1\text{:} \;\; \textbf{await } \neg\text{req0} \lor \text{turn} = 1; \\
\quad \text{cs}_1\text{:} \;\; \textbf{skip}; \\
\quad \text{ex}_1\text{:} \;\; \text{req1} := \text{false}; \\
\textbf{endloop}
\end{array}
$$

- Critical section can be abstracted to atomic step
- Is it okay to combine the following actions into an atomic step?
  1. statements rq_i and ps_i

Example: mutual exclusion algorithms

```
            integer   turn = 0;
            boolean   req0, req1 = false;

process P0                                  process P1
loop                                        loop
  nc0:  skip;                                 nc1:  skip;
  rq0:  ⟨req0 := true;                        rq1:  ⟨req1 := true;
         turn := 1;                     ||           turn := 0;
         await ¬req1 ∨ turn = 0;⟩                    await ¬req0 ∨ turn = 1;⟩
  cs0:  skip;                                 cs1:  skip;
  ex0:  req0 := false;                        ex1:  req1 := false;
endloop                                     endloop
```

- Critical section can be abstracted to atomic step
- Is it okay to combine the following actions into an atomic step?
  1. statements rq_i and ps_i
  2. statements rq_i, ps_i, and wt_i

Example: mutual exclusion algorithms

```
                integer   turn = 0;
                boolean   req0, req1 = false;

process P0                                   process P1
loop                                         loop
  nc0:  skip;                                  nc1:  skip;
  rq0:  req0 := true;                          rq1:  req1 := true;
  ps0:  turn := 1;                      ||     ps1:  turn := 0;
  wt0:  await ¬req1 ∨ turn = 0;                wt1:  await ¬req0 ∨ turn = 1;
  cs0:  ⟨skip;                                 cs1:  ⟨skip;
          req0 := false;⟩                              req1 := false;⟩
endloop                                      endloop
```

- Critical section can be abstracted to atomic step
- Is it okay to combine the following actions into an atomic step?
  1. statements rq_i and ps_i
  2. statements rq_i, ps_i, and wt_i
  3. statements cs_i and ex_i

Outline

Reduction: overall idea

- Justify combining subsequent operations into an atomic step
- Fewer atomic steps $\rightsquigarrow$ simpler verification

Theorem (folklore)

One can pretend that a sequence of statements is executed atomically if it contains at most one access to a shared variable.

- Folk theorem justifies combining cs_i and ex_i (previous example)
- Folk theorem does not justify combining rq_i and ps_i

Reduction: overall idea

- Justify combining subsequent operations into an atomic step
- Fewer atomic steps $\rightsquigarrow$ simpler verification

Theorem (folklore)

One can pretend that a sequence of statements is executed atomically if it contains at most one access to a shared variable.

- Folk theorem justifies combining cs_i and ex_i (previous example)
- Folk theorem does not justify combining rq_i and ps_i
- Consider the single-process program where initially $x = y$

$$y := x + 1;\ x := y$$

Since no variable is shared, it should be equivalent to

$$\langle y := x + 1;\ x := y \rangle$$

Reduction: overall idea

- Justify combining subsequent operations into an atomic step
- Fewer atomic steps $\rightsquigarrow$ simpler verification

Theorem (folklore)

One can pretend that a sequence of statements is executed atomically if it contains at most one access to a shared variable.

- Folk theorem justifies combining cs_i and ex_i (previous example)
- Folk theorem does not justify combining rq_i and ps_i
- Consider the single-process program where initially $x = y$

$$y := x + 1;\ x := y$$

Since no variable is shared, it should be equivalent to

$$\langle y := x + 1;\ x := y \rangle$$

But the latter program satisfies $\Box(x = y)$!

Left and right movers

Definition (Lipton 1975)

An action a is a right mover if whenever αab is a computation where a and b are performed by different processes then αba is also a computation and these computations result in the same state. The definition of a left mover is symmetrical.

- Right mover $\quad s \xrightarrow{ab} t \;\Rightarrow\; s \xrightarrow{ba} t \quad$ for all b
  - right commutes with every action of different processes
  - example: acquisitions of resources (e.g., semaphores)
- Left mover $\quad s \xrightarrow{ba} t \;\Rightarrow\; s \xrightarrow{ab} t \quad$ for all b
  - left commutes with every action of different processes
  - example: releases of resources

R.J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. CACM 18(12):717-721, 1975.

Left and right movers in example

integer turn = 0;
boolean req0, req1 = false;

process P0
loop
  nc_0: **skip**;
  rq_0: req0 := true;
  ps_0: turn := 1;
  wt_0: **await** ¬req1 ∨ turn = 0;
  cs_0: **skip**;
  ex_0: req0 := false;
endloop

‖

process P1
loop
  nc_1: **skip**;
  rq_1: req1 := true;
  ps_1: turn := 0;
  wt_1: **await** ¬req0 ∨ turn = 1;
  cs_1: **skip**;
  ex_1: req1 := false;
endloop

- Actions rq_i are right movers
  - in particular, cannot make **await** condition of other process true
  - formally, $s \xrightarrow{rq_0\ wt_1} t$ implies $s \xrightarrow{wt_1\ rq_0} t$

Left and right movers in example

$$\begin{array}{ll} \textbf{integer} & \text{turn} = 0; \\ \textbf{boolean} & \text{req0, req1} = \text{false}; \end{array}$$

$$\begin{array}{lcl}
\begin{array}{l}
\textbf{process } \text{P0} \\
\textbf{loop} \\
\quad \text{nc}_0\text{:}\ \ \textbf{skip}; \\
\quad \text{rq}_0\text{:}\ \ \text{req0} := \text{true}; \\
\quad \text{ps}_0\text{:}\ \ \text{turn} := 1; \\
\quad \text{wt}_0\text{:}\ \ \textbf{await } \neg\text{req1} \vee \text{turn} = 0; \\
\quad \text{cs}_0\text{:}\ \ \textbf{skip}; \\
\quad \text{ex}_0\text{:}\ \ \text{req0} := \text{false}; \\
\textbf{endloop}
\end{array}
& \| &
\begin{array}{l}
\textbf{process } \text{P1} \\
\textbf{loop} \\
\quad \text{nc}_1\text{:}\ \ \textbf{skip}; \\
\quad \text{rq}_1\text{:}\ \ \text{req1} := \text{true}; \\
\quad \text{ps}_1\text{:}\ \ \text{turn} := 0; \\
\quad \text{wt}_1\text{:}\ \ \textbf{await } \neg\text{req0} \vee \text{turn} = 1; \\
\quad \text{cs}_1\text{:}\ \ \textbf{skip}; \\
\quad \text{ex}_1\text{:}\ \ \text{req1} := \text{false}; \\
\textbf{endloop}
\end{array}
\end{array}$$

- Actions rq_i are right movers
  - in particular, cannot make **await** condition of other process true
  - formally, $s \xrightarrow{rq_0\ wt_1} t$ implies $s \xrightarrow{wt_1\ rq_0} t$
- Actions cs_i and ex_i are left movers

Left and right movers in example

integer turn = 0;
boolean req0, req1 = false;

process P0		process P1
loop		**loop**
nc_0: **skip**;		nc_1: **skip**;
rq_0: req0 := true;		rq_1: req1 := true;
ps_0: turn := 1;	‖	ps_1: turn := 0;
wt_0: **await** $\neg$req1 $\vee$ turn = 0;		wt_1: **await** $\neg$req0 $\vee$ turn = 1;
cs_0: **skip**;		cs_1: **skip**;
ex_0: req0 := false;		ex_1: req1 := false;
endloop		**endloop**

- Actions rq_i are right movers
  - in particular, cannot make **await** condition of other process true
  - formally, $s \xrightarrow{rq_0\ wt_1} t$ implies $s \xrightarrow{wt_1\ rq_0} t$
- Actions cs_i and ex_i are left movers
- Actions ps_i and wt_i are neither left nor right movers

Lipton's reduction theorem

Theorem (Lipton 1975)

Suppose that $A = A_1; \ldots ; A_k$ *is such that for some i:*

- $A_1, \ldots, A_{i-1}$ *are right movers,*
- $A_{i+1}, \ldots, A_k$ *are left movers,*
- *and each* $A_2, \ldots, A_k$ *can always execute.*

and let P/A *denote the program obtained from* P *by replacing* $A_1; \ldots ; A_k$ *by* $\langle A_1; \ldots ; A_k \rangle$.

Then P *halts iff* P/A *halts and the final states of* P *equal the final states of* P/A.

- Preservation of deadlock-freedom and partial correctness

Application to example

Lipton's theorem justifies reduction to

$$\textbf{integer} \quad turn = 0;$$
$$\textbf{boolean} \; req0, req1 = false;$$

process P0		process P1
loop		**loop**
nc_0: **skip**;		nc_1: **skip**;
rq_0: $\langle$req0 := true;		rq_1: $\langle$req1 := true;
turn := 1;$\rangle$	$\|$	turn := 0;$\rangle$
wt_0: $\langle$**await** $\neg$req1 $\vee$ turn = 0;		wt_1: $\langle$**await** $\neg$req0 $\vee$ turn = 1;
skip;		**skip**;
req0 := false;$\rangle$		req1 := false;$\rangle$
endloop		**endloop**

… but only for proving absence of deadlock

Doeppner's reduction theorem

Theorem

Let Π be a program and S have the form $R; \langle A \rangle; L$ where

- *all actions in R are right movers and*
- *all actions in L are left movers.*

Let $in(S)$ be true iff control resides inside S and Q be an arbitrary predicate.

Then Q is an invariant of Π/S iff $Q \vee in(S)$ is an invariant of Π.

- Generalization of Lipton's theorem to invariant reasoning
- Can be used for proving mutual exclusion of example program

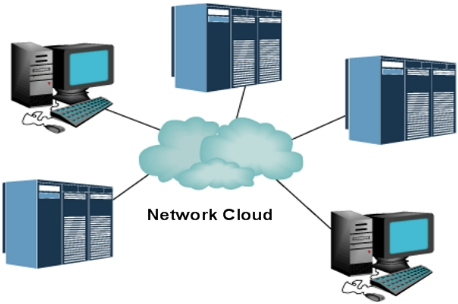
T.W. Doeppner. Parallel program correctness through refinement. POPL 1977 (ACM), pp. 155-169.

Other reduction theorems

- R. Back: *Refining atomicity in parallel algorithms* (1988)
  - first reduction theorem for total correctness
  - needs commutativity hypotheses for actions outside reduced block
- L. Lamport, F. Schneider: *Pretending Atomicity* (1989)
  - generalization of Doeppner's theorem
  - preservation of invariants Q of Π by reduction (explicit reasoning about control being external to reduced block)
- E. Cohen, L. Lamport: *Reduction in TLA* (1998)
  - reformulation of Lamport & Schneider in TLA
  - extension to (certain) liveness properties

Outline

Fault-tolerant distributed algorithms



- local computation of nodes
- asynchronous communication over network
- components may fail: replication & fault-tolerance
- precisely state and prove correctness properties

Representative problem: consensus

- N nodes (processes) agree on a value
  - each node proposes a value initially
  - eventually nodes decide a common value
  - nodes or communication links may fail
- Formal definition: conjunction of four properties

integrity	decided value is among the initial proposals
irrevocability	decisions cannot be undone
agreement	any two nodes decide same value
termination	all (non-failed) nodes decide eventually

- Fundamental problem in fault-tolerant distributed computing

Why is this hard?

Theorem (Fischer, Lynch, Paterson 1985)

The Consensus problem cannot be solved in an asynchronous system where at least one process may fail (by crashing).

- But: many consensus algorithms exist (and work well in practice)

Why is this hard?

Theorem (Fischer, Lynch, Paterson 1985)

The Consensus problem cannot be solved in an asynchronous system where at least one process may fail (by crashing).

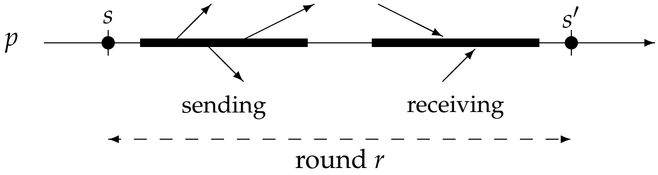
- But: many consensus algorithms exist (and work well in practice)
- Basis: relax some assumption of FLP theorem
  - introduce timeouts: being late is a failure
  - assume reliable (broadcast) communication
  - augment system by an oracle to detect failures
- Verification of consensus algorithms
  - difficult proofs … often absent or informal
  - DiskPaxos: careful paper proof (30 pages for 0.5 page algorithm)
- Can we help make verification simpler?

Heard-Of Model (Charron-Bost & Schiper, 2006)

- Algorithmic model for fault-tolerant distributed algorithms
  - uniform treatment of all (benign) errors
  - do not identify “culprit” or “type” of failure

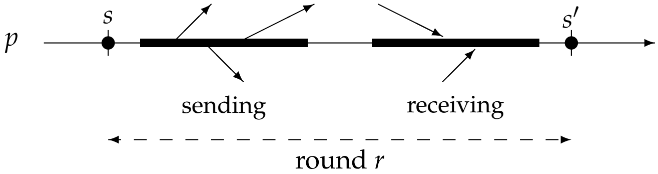
Heard-Of Model (Charron-Bost & Schiper, 2006)

- Algorithmic model for fault-tolerant distributed algorithms
  - uniform treatment of all (benign) errors
  - do not identify "culprit" or "type" of failure
- Round-based computation model



Heard-Of Model (Charron-Bost & Schiper, 2006)

- Algorithmic model for fault-tolerant distributed algorithms
  - uniform treatment of all (benign) errors
  - do not identify “culprit” or “type” of failure
- Round-based computation model



  - rounds: local structure of process computation
  - state s' computed from s and received messages
  - heard-of set $HO(p,r)$: processes from which messages are received
  - communication-closed rounds: discard late messages

Formal representation of HO algorithms

- Collection of processes $(State_p, s_{0,p}, S_p^r, T_p^r)_{p \in Proc, r \in \mathbb{N}}$
  - process states: sets $State_p$ with initial states $s_{0,p} \in State_p$
  - message sending and state transition

$$S_p^r : State_p \times Proc \to Msg$$
$$T_p^r : State_p \times (Proc \rightharpoonup Msg) \to State_p$$

  - domain of second argument of T_p^r: heard-of set $HO(p, r)$

- For simplicity: deterministic processes
  - algorithm behavior determined by collection of heard-of sets
  - extension to non-deterministic processes straightforward

Communication predicates

- Algorithms do not work in presence of arbitrary failures
  - safety: restrict number or extent of errors
  - liveness: assume eventual functioning of components
- Sample communication predicates

non-split rounds	$\forall p, q, r : HO(p,r) \cap HO(q,r) \neq \emptyset$
$\leq f$ failures	$\forall p, r : \lvert HO(p,r) \rvert \geq N - f$
event. uniform	$\exists r_0 \in \mathbb{N}, P \subseteq Proc : \forall r \geq r_0, q \in Proc : HO(q,r) = P$

- Observations (Charron-Bost & Schiper)
  - standard failure assumptions can be expressed in terms of *HO* sets

HO Consensus Algorithm: One-Third Rule

Initialization

$x_p := v_p$, $decide_p := null$ *(v_p : initial value of p)*

For each round $r \geq 0$

S_p^r : send x_p to all processes

T_p^r : **if** $|HO(p,r)| > 2N/3$ **then**

set x_p to smallest among the most frequently received values

if more than $2N/3$ values received are equal to x_p **then**

$decide_p := x_p$

Simple but efficient consensus algorithm

- no coordinator needed
- quick convergence if few errors

Representing executions of HO algorithms

- Fine-grained execution for HO collection $(HO(p,r))_{p \in Proc, r \in \mathbb{N}}$
  - message receptions, local transitions, message sending
  - verify correctness for all HO collections

process Node($p \in Proc$)
  state $st = s_{0,p}$;
  integer $r = 0$;
  for $q \in Proc$ **do** $send(p,q,r,S_p^r(st,q))$ **enddo**;
  loop
    array $rcvd = [q \in Proc \mapsto null]$;
    for $q \in HO(p,r)$ **do** $rcvd[q] := receive(q,p,r)$ **enddo**;
    $st, r := T_p^r(st, rcvd), r+1$;
    for $q \in Proc$ **do** $send(p,q,r,S_p^r(st,q))$ **enddo**;
  end loop
end process

- Formally: infinite sequence $\xi = c_0 c_1 \ldots$ of configurations

Representing executions of HO algorithms

- Fine-grained execution for HO collection $(HO(p,r))_{p \in Proc, r \in \mathbb{N}}$
  - message receptions, local transitions, message sending
  - verify correctness for all HO collections

process Node($p \in Proc$)
  state $st = s_{0,p}$;
  integer $r = 0$;
  for $q \in Proc$ **do** $send(p,q,r,S_p^r(st,q))$ **enddo**;
  loop
    array $rcvd = [q \in Proc \mapsto null]$;
    for $q \in HO(p,r)$ **do** $rcvd[q] := receive(q,p,r)$ **enddo**;
    $st, r := T_p^r(st, rcvd), r+1$;
    for $q \in Proc$ **do** $send(p,q,r,S_p^r(st,q))$ **enddo**;
  end loop
end process

- Formally: infinite sequence $\xi = c_0 c_1 \ldots$ of configurations
- Infinite-state model, due to round numbers

Outline

First reduction

- Remember left and right movers?
  - send actions are left movers
  - receive actions are right movers

(assuming infinite network capacity)

First reduction

- Remember left and right movers?
  - send actions are left movers
  - receive actions are right movers

$\left(\begin{array}{l}\text{assuming infinite}\\ \text{network capacity}\end{array}\right)$

- This motivates the following reduction:

process Node($p \in Proc$)
  $\langle$**state** $st = s_{0,p}$;
   integer $r = 0$;
   for $q \in Proc$ **do** $send(p, q, r, S_p^r(st, q))$ **enddo**$\rangle$;
  loop
    $\langle$**array** $rcvd = [q \in Proc \mapsto null]$;
     for $q \in HO(p, r)$ **do** $rcvd[q] := receive(q, p, r)$ **enddo**;
     $st, r := T_p^r(st, rcvd), r + 1$;
     for $q \in Proc$ **do** $send(p, q, r, S_p^r(st, q))$ **enddo**$\rangle$;
  end loop
end process

More reduction

- Processes execute rounds atomically

init	*init*	*rnd* 0	*init*	*rnd* 0	*rnd* 1	*rnd* 0	*rnd* 1	*rnd* 2	*rnd* 1	···

- Can we do any better?

More reduction

- Processes execute rounds atomically

init	*init*	*rnd* 0	*init*	*rnd* 0	*rnd* 1	*rnd* 0	*rnd* 1	*rnd* 2	*rnd* 1	$\cdots$

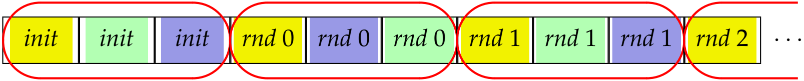
- Can we do any better?
- Remember communication-closed rounds
  - round rnd_p^m right-commutes with rnd_q^n if $m > n$
  - messages sent during rnd_q^n did not influence rnd_p^m
- Rearrange execution so that executions of same round are adjacent

init	*init*	*init*	*rnd* 0	*rnd* 0	*rnd* 0	*rnd* 1	*rnd* 1	*rnd* 1	*rnd* 2	$\cdots$

More reduction

- Processes execute rounds atomically

- Can we do any better?
- Remember communication-closed rounds
  - round rnd_p^m right-commutes with rnd_q^n if $m > n$
  - messages sent during rnd_q^n did not influence rnd_p^m
- Rearrange execution so that executions of same round are adjacent



- Executions of same round by different processes are independent

Coarse-grained model of executions

- Unit of atomicity: entire system rounds
  - all processes simultaneously perform transition for same round
  - corresponds to "nice" executions in the fine-grained model
- Coarse-grained execution $\sigma_0\sigma_1\ldots$ $\quad(\sigma_i : Proc \to State)$
  - $\sigma_0(p) = s_{0,p}$
  - $\sigma_{r+1}(p) = T_p^r\big(\sigma_r(p), rcvd(p,r)\big)$

  where $\quad rcvd(p,r) = [q \in HO(p,r) \mapsto S_q^r(\sigma_r(q),p)]$
- Coarse abstraction of distributed execution
  - no need for explicit representation of network
  - no round numbers: "synchronized" processes

Coarse-grained model of executions

- Unit of atomicity: entire system rounds
  - all processes simultaneously perform transition for same round
  - corresponds to “nice” executions in the fine-grained model
- Coarse-grained execution $\sigma_0\sigma_1\ldots \qquad (\sigma_i : Proc \to State)$
  - $\sigma_0(p) = s_{0,p}$
  - $\sigma_{r+1}(p) = T_p^r\big(\sigma_r(p), rcvd(p,r)\big)$

    where $\quad rcvd(p,r) = [q \in HO(p,r) \mapsto S_q^r(\sigma_r(q), p)]$
- Coarse abstraction of distributed execution
  - no need for explicit representation of network
  - no round numbers: “synchronized” processes

⇒ How exactly does the reduced model relate to the original one?

Relating fine- and coarse-grained executions

- Fine-grained model contains more detail
- Compare executions w.r.t. the "local views" of processes
  - p-view of fine-grained execution $\xi = c_0 c_1 \ldots$

    $$\xi^p = c_0.st(p), c_1.st(p), \ldots$$

  - p-view of coarse-grained execution $\sigma = \sigma_0 \sigma_1 \ldots$

    $$\sigma^p = \sigma_0(p), \sigma_1(p), \ldots$$

  - p-views are sequences of states of p and can be compared
- Executions equivalent iff indistinguishable by any process

  $$\xi \approx \sigma \quad \text{iff} \quad \natural(\xi^p) = \natural(\sigma^p) \ \text{ for every } p \in \mathit{Proc}$$

  - local views equal up to stuttering, for every process

Reduction theorem

Theorem (Reduction)

Given a HO collection $(HO(p,r))$ and a fine-grained execution ξ there exists a coarse-grained execution σ for the same HO collection such that $\sigma \approx \xi$.

Proof. For $\xi = c_0 c_1 \ldots$, define sequence $\sigma = ([p \in Proc \mapsto c_{\ell_r^p}.st(p)])_{r \in \mathbb{N}}$

where $\begin{cases} \ell_0^p = 0 \\ \ell_{r+1}^p = k+1 \quad \text{if } (c_k, c_{k+1}) \text{ is } (r+1)\text{st local transition of } p. \end{cases}$

Then σ is a coarse-grained execution for the same HO collection.

Moreover, $\natural(\sigma^p) = \natural(\xi^p)$ for all $p \in Proc$. Q.E.D.

- Converse theorem is trivially true

"Local" properties

- Application of reduction theorem to verification
  - many properties depend only on local views
  - these can be verified by considering only coarse-grained executions

- Local properties P of executions

$$\rho_1 \models P \quad \text{iff} \quad \rho_2 \models P \qquad \text{whenever } \rho_1 \approx \rho_2$$

"Local" properties

- Application of reduction theorem to verification
  - many properties depend only on local views
  - these can be verified by considering only coarse-grained executions
- Local properties P of executions

$$\rho_1 \models P \quad \text{iff} \quad \rho_2 \models P \qquad \text{whenever } \rho_1 \approx \rho_2$$

- The following LTL-X properties are local
  - formulas $Q(p)$ built solely from p's state variables
  - arbitrary first-order combinations of local properties
  - but: temporal combinations need not be local, consider:

$$\bigwedge_{p,q \in Proc} \Box(rnd_p = rnd_q) \qquad \text{(where } rnd_p \text{ is the current round of } p\text{)}$$

Consensus as a local property

- Integrity

$$\bigwedge_{p \in Proc} \forall v \neq null : \left(\Diamond(decide_p = v) \Rightarrow \bigvee_{q \in Proc} x_q = v \right)$$

- Irrevocability

$$\bigwedge_{p \in Proc} \forall v \neq null : \Box(decide_p = v \Rightarrow \Box(decide_p = v))$$

- Agreement

$$\bigwedge_{p,q \in Proc} \forall v, w \neq null : \Diamond(decide_p = v) \wedge \Diamond(decide_q = w) \Rightarrow v = w$$

- Termination

$$\bigwedge_{p \in Proc} \Diamond(decide_p \neq null)$$

Outline

Finite-state model checking

- Verification of finite instances of algorithms
  - model coarse-grained executions for fixed number of processes
  - non-deterministic choice of HO sets at every transition
  - resulting model is finite-state
- Generic TLA$^+$ module *HeardOf*
  - high-level definition of coarse-grained HO semantics
  - pre-define useful communication predicates
  - concrete algorithms obtained later as instances
- Here: favor clarity over efficiency

Generic TLA$^+$ module

MODULE *HeardOf*

EXTENDS *Naturals*
CONSTANTS $Proc, State, Msg, nPhases, IniSt(\_), Send(\_,\_,\_,\_), Trans(\_,\_,\_,\_)$
VARIABLES *phase*, *state*, *heardof*

$$Init \triangleq \land\ phase = 0$$
$$\qquad \land\ state = [p \in Proc \mapsto IniSt(p)]$$
$$\qquad \land\ heardof = [p \in Proc \mapsto \{\}]$$

$$Step(HO) \triangleq \text{LET } rcvd(p) \triangleq \{\langle q, Send(q, phase, state[q], p)\rangle : q \in HO[p]\}$$
$$\text{IN } \land\ phase' = (phase + 1)\ \%\ nPhases$$
$$\land\ state' = [p \in Proc \mapsto Trans(p, phase, state[p], rcvd(p))]$$
$$\land\ heardof' = HO$$

$$Next \triangleq \exists HO \in [Proc \to \text{SUBSET } Proc] : Step(HO)$$

$$NoSplit(HO) \triangleq \forall p, q \in Proc : HO[p] \cap HO[q] \neq \{\}$$

$$NextNoSplit \triangleq \exists HO \in [Proc \to \text{SUBSET } Proc] : NoSplit(HO) \land Step(HO)$$

$$Uniform(HO) \triangleq \exists S \in \text{SUBSET } Proc : HO = [q \in Proc \mapsto S]$$

$$InfiniteUniform \triangleq \Box\Diamond Uniform(heardof)$$

Remarks

- Definitions closely parallels "paper" version
  - expressiveness of TLA$^+$ leads to perspicuous formulation
  - (auxiliary) variable *heardof* records HO sets during a run
  - mainly used for debugging and printing counter-examples
- Formulation of communication predicates
  - safety predicates: add to next-state relation
  - liveness predicates: natural expression in temporal logic
  - used to express correctness properties

One-Third Rule in TLA$^+$ (1/3)

MODULE *OneThirdRule*

EXTENDS *Naturals*, *FiniteSets*
CONSTANT N
VARIABLES *phase*, *state*, *heardof*

$$
\begin{array}{lcl}
nPhases & \triangleq & 1 \\
Proc & \triangleq & 1..N \\
InitValue(p) & \triangleq & 10 * p \\
Value & \triangleq & \{InitValue(p) : p \in Proc\} \\
Msg & \triangleq & Value \\
null & \triangleq & 0 \\
ValueOrNull & \triangleq & Value \cup \{null\} \\
State & \triangleq & [x : Value, decide : ValueOrNull]
\end{array}
$$

- definition of constant parameters for *OneThirdRule* algorithm
- arbitrary definition of (initial) values of a process

One-Third Rule in TLA$^+$ (2/3)

$$
\begin{array}{lll}
IniSt(p) & \triangleq & [x \mapsto InitValue(p), decide \mapsto null] \\
Send(p, ph, s, q) & \triangleq & s.x \\
Trans(p, ph, s, rcvd) & \triangleq &
\end{array}
$$

$$
\begin{array}{l}
\text{IF } Cardinality(rcvd) > (2 * N) \div 3 \\
\text{THEN LET } Freq(v) \triangleq Cardinality(\{q \in Proc : \langle q, v\rangle \in rcvd\}) \\
\qquad\qquad MFR(v) \triangleq \forall w \in Value : Freq(w) \leq Freq(v) \\
\qquad\qquad min \triangleq \text{CHOOSE } v \in Value : MFR(v) \land \forall w \in Value : MFR(w) \Rightarrow v \leq w \\
\qquad\quad \text{IN} \quad [x \mapsto min, \\
\qquad\qquad\quad decide \mapsto \text{IF } Freq(min) > (2 * N) \div 3 \text{ THEN } min \text{ ELSE } s.decide] \\
\text{ELSE } s
\end{array}
$$

$$\text{INSTANCE } HeardOf$$

- definition of the send and state transition functions
- instantiation of generic module

One-Third Rule in TLA$^+$ (3/3)

$$
\begin{aligned}
&Safety &&\triangleq\ Init \land \Box[Next]_{vars}\\
&Liveness &&\triangleq\ \Box\Diamond(Uniform(heardof) \land Cardinality(heardof) > (2 * N) \div 3)\\
&Integrity &&\triangleq\ \forall p \in Proc : state[p].decide \in ValueOrNull\\
&Irrevocability &&\triangleq\ \forall p \in Proc : \Box[state[p].decide = null]_{state[p].decide}\\
&Agreement &&\triangleq\ \forall p, q \in Proc : (state[p].decide \neq null \land state[q].decide \neq null\\
& && \qquad\qquad \Rightarrow state[p].decide = state[q].decide)\\
&Termination &&\triangleq\ \forall p \in Proc : \Diamond(state[p].decide \neq null)
\end{aligned}
$$

THEOREM $Safety \Rightarrow \Box(Integrity \land Agreement) \land Irrevocability$

THEOREM $Safety \land Liveness \Rightarrow Termination$

- definition of correctness properties
- formulation of correctness theorems, under precise hypotheses

Results of verification

	OneThirdRule		UniformVoting	
	$N = 3$	$N = 4$	$N = 3$	$N = 4$
states	5633	9,830,401	21,351	15,865,770
distinct	11	150	122	887
time (s)	1.87	939	13.8	1330

- Model checking feasible for small instances
  - high branching factor: exploration of all HO collections
  - many redundant states generated
- Symbolic model checking can be more efficient
  - more complicated encodings necessary for tools like NuSMV
  - cf. work by Tsuchiya and Schiper: Paxos for 10 processes

Verification in Isabelle/HOL

Similar overall model

- main difference: introduction of types
- generic *HeardOf* module represented as an Isabelle locale

$$
\begin{array}{l}
\textbf{locale } \mathit{HOAlgorithm} = \\
\textbf{fixes} \\
\quad \mathit{nPhases} :: \mathit{nat} \ \textbf{ and} \\
\quad \mathit{iniSt} :: {}'\mathit{proc} \to {}'\mathit{pst} \ \textbf{ and} \\
\quad \mathit{send} :: {}'\mathit{proc} \to \mathit{nat} \to {}'\mathit{pst} \to {}'\mathit{proc} \to {}'\mathit{msg} \ \textbf{ and} \\
\quad \mathit{trans} :: {}'\mathit{proc} \to \mathit{nat} \to {}'\mathit{pst} \to ({}'\mathit{proc} \rightharpoonup {}'\mathit{msg}) \to {}'\mathit{pst} \\
\textbf{assumes} \\
\quad \mathit{nSteps} : 0 < \mathit{nPhases} \ \textbf{ and} \\
\quad \mathit{finiteProc} : \mathit{finite}(\mathit{UNIV} :: {}'\mathit{procset})
\end{array}
$$

- defines generic behavior of HO algorithms
- proves useful rules, such as induction over executions

Proof of correctness

- Validity: standard invariance proof
- Irrevocability and agreement via sequence of lemmas
  1. if process decides on value v then more than $2N/3$ processes contain v in their x field
  2. if more than $2N/3$ processes send v and process p hears from more than $2N/3$ processes then p updates its x field to v
  3. whenever process has decided on v then more than $2N/3$ processes contain v in their x field
  4. hence, processes cannot decide on different values
- Liveness: symbolically execute uniform rounds

Proof of correctness

- Validity: standard invariance proof
- Irrevocability and agreement via sequence of lemmas
  1. if process decides on value v then more than $2N/3$ processes contain v in their x field
  2. if more than $2N/3$ processes send v and process p hears from more than $2N/3$ processes then p updates its x field to v
  3. whenever process has decided on v then more than $2N/3$ processes contain v in their x field
  4. hence, processes cannot decide on different values
- Liveness: symbolically execute uniform rounds
- Proof lengths in Isar (including model and explanations)
  - 8 pages for generic module and lemmas
  - 8 pages for *OneThirdRule*
  - 25 pages for *LastVoting* (cf. 130 pages for fine-grained model!)

Outline

Reduction: a revival?

- Recast of classical theorems
  - identify left and right movers for coarser unit of atomicity
  - distributed algorithms present interesting opportunities
  - substantial reduction of verification effort possible
- Transcend historical formulations
  - beyond programming-language based presentations
  - wide interpretation of "processes" (e.g., set of rounds)
  - verify safety *and* liveness properties
- Ongoing / future work
  - establish more general reduction theorems
  - better syntactic characterization of local properties
  - implementation of reduction in verification tools