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
loop
  nc0: skip;
  rq0: req0 := true;
  ps0: turn := 1;
  wt0: await ¬req1 ∨ turn = 0;
  cs0: skip;
  ex0: req0 := false;
endloop

process P1
loop
  nc1: skip;
  rq1: req1 := true;
  ps1: turn := 0;
  wt1: await ¬req0 ∨ turn = 1;
  cs1: skip;
  ex1: req1 := false;
endloop
```

- Critical section can be abstracted to atomic step
Example: mutual exclusion algorithms

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

process P0
loop
  nc0:  skip;
  rq0:  ⟨req0 := true;
        turn := 1⟩
  wt0:  await ¬req1 ∨ turn = 0;
  cs0:  skip;
  ex0:  req0 := false;
endloop

process P1
loop
  nc1:  skip;
  rq1:  ⟨req1 := true;
        turn := 0⟩
  wt1:  await ¬req0 ∨ turn = 1;
  cs1:  skip;
  ex1:  req1 := false;
endloop
```

- Critical section can be abstracted to atomic step
- Is it okay to combine the following actions into an atomic step?
  - statements rq_i and ps_i
Example: mutual exclusion algorithms

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

process P0
loop
  nc0: skip;
  rq0: ⟨req0 := true;
       turn := 1;
       await ¬req1 ∨ turn = 0;⟩
  cs0: skip;
  ex0: req0 := false;
endloop

process P1
loop
  nc1: skip;
  rq1: ⟨req1 := true;
       turn := 0;
       await ¬req0 ∨ turn = 1;⟩
  cs1: skip;
  ex1: req1 := false;
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
loop
  nc0: skip;
  rq0: req0 := true;
  ps0: turn := 1;
  wt0: await ¬req1 ∨ turn = 0;
  cs0: ⟨skip; req0 := false;⟩
endloop

process P1
loop
  nc1: skip;
  rq1: req1 := true;
  ps1: turn := 0;
  wt1: await ¬req0 ∨ turn = 1;
  cs1: ⟨skip; req1 := false;⟩
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

1. Reduction Theorems for the Verification of Concurrent Programs
2. Fault-Tolerant Distributed Computing
3. Reduction for Round-Based Distributed Algorithms
4. Experiments: Verification of Consensus Algorithms
5. Conclusion
Reduction: overall idea

- Justify combining subsequent operations into an atomic step
- Fewer atomic steps $\leadsto$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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 $\square(x = y)$!
Left and right movers

**Definition (Lipton 1975)**

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

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

- **Left mover**

  $s \xrightarrow{ba} t \Rightarrow s \xrightarrow{ab} t$ for all $b$
  
  - left commutes with every action of different processes
  - example: releases of resources

Left and right movers in example

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

process P0
loop
  nc0: skip;
  rq0: req0 := true;
  ps0: turn := 1;
  wt0: await ¬req1 ∨ turn = 0;
  cs0: skip;
  ex0: req0 := false;
endloop

process P1
loop
  nc1: skip;
  rq1: req1 := true;
  ps1: turn := 0;
  wt1: await ¬req0 ∨ turn = 1;
  cs1: skip;
  ex1: 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**

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

process P0
loop
  nc0: skip;
  rq0: req0 := true;
  ps0: turn := 1;
  wt0: await ¬req1 ∨ turn = 0;
  cs0: skip;
  ex0: req0 := false;
endloop

process P1
loop
  nc1: skip;
  rq1: req1 := true;
  ps1: turn := 0;
  wt1: await ¬req0 ∨ turn = 1;
  cs1: skip;
  ex1: 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 $s \xrightarrow{wt_1 \ rq_0} t$

- **Actions cs_i and ex_i are left movers**
Left and right movers in example

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

process P0
loop
  nc0: skip;
  rq0: req0 := true;
  ps0: turn := 1;
  wt0: await ¬req1 ∨ turn = 0;
  cs0: skip;
  ex0: req0 := false;
endloop

process P1
loop
  nc1: skip;
  rq1: req1 := true;
  ps1: turn := 0;
  wt1: await ¬req0 ∨ turn = 1;
  cs1: skip;
  ex1: req1 := false;
endloop
```

- **Actions** $rq_i$ **are right movers**
  - in particular, cannot make `await` condition of other process true
  - formally, $s \xrightarrow{rq0 \ wt1} t$ implies $s \xrightarrow{wt1 \ rq0} t$

- **Actions** $cs_i$ and $ex_i$ **are left movers**

- **Actions** $ps_i$ and $wt_i$ **are neither left nor right movers**

Stephan Merz (INRIA Nancy)  Reduction Revisited  MPC 2010 6 / 39
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
Lipton’s theorem justifies reduction to

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

process P0
loop
  nc0: skip;
  rq0: ⟨req0 := true;
       turn := 1;⟩
  wt0: ⟨await ¬req1 ∨ turn = 0;
       skip;
       req0 := false;⟩
endloop

process P1
loop
  nc1: skip;
  rq1: ⟨req1 := true;
       turn := 0;⟩
  wt1: ⟨await ¬req0 ∨ turn = 1;
       skip;
       req1 := false;⟩
endloop
```

… but only for proving absence of deadlock
**Theorem**

Let $\Pi$ 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 $\text{in}(S)$ be true iff control resides inside $S$ and $Q$ be an arbitrary predicate.

Then $Q$ is an invariant of $\Pi/S$ iff $Q \lor \text{in}(S)$ is an invariant of $\Pi$.

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

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 $\Pi$ 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

1. Reduction Theorems for the Verification of Concurrent Programs

2. Fault-Tolerant Distributed Computing

3. Reduction for Round-Based Distributed Algorithms

4. Experiments: Verification of Consensus Algorithms

5. Conclusion
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**

```
p
         s
   sending  

     round r

   receiving

         s'
```

Stephan Merz (INRIA Nancy)  Reduction Revisited  MPC 2010  15 / 39
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 \((State_p, s_{0,p}, S^r_p, T^r_p)_{p \in \text{Proc}, r \in \mathbb{N}}\)
  - process states: sets \(State_p\) with initial states \(s_{0,p} \in State_p\)
  - message sending and state transition
    \[
    S^r_p : State_p \times \text{Proc} \rightarrow \text{Msg}
    \]
    \[
    T^r_p : State_p \times (\text{Proc} \rightarrow \text{Msg}) \rightarrow State_p
    \]
  - domain of second argument of \(T^r_p\): 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: \( \forall p, r : |HO(p, r)| \geq N - f \)
  - 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
\[ x_p := v_p, \text{decide}_p := \text{null} \] \hspace{1cm} (v_p : initial value of p)

For each round \( r \geq 0 \)

\( S^r_p : \) send \( x_p \) to all processes

\( T^r_p : \) if \( |\text{HO}(p, r)| > 2N/3 \) 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

\( \text{decide}_p := x_p \)

Simple but efficient consensus algorithm

- no coordinator needed

- quick convergence if few errors
Representing executions of HO algorithms

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

\begin{verbatim}
process Node(p ∈ Proc)
  state st = s_{0,p};
  integer r = 0;
  for q ∈ Proc do send(p,q,r,S^r_p(st,q)) enddo;
loop
  array rcvd = [q ∈ Proc ↦ null];
  for q ∈ HO(p,r) do rcvd[q] := receive(q,p,r) enddo;
  st, r := T^r_p(st,rcvd), r + 1;
  for q ∈ Proc do send(p,q,r,S^r_p(st,q)) enddo;
end loop
end process
\end{verbatim}

- **Formally:** infinite sequence \(\xi = c_0c_1 \ldots\) of configurations
Representing executions of HO algorithms

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

```plaintext
process Node(p \in \text{Proc})
  state \(st = s_{0,p}\);
  integer \(r = 0\);
  for \(q \in \text{Proc}\) do send\((p, q, r, S^r_p(st, q))\) enddo;
loop
  array \(rcvd = [q \in \text{Proc} \mapsto \text{null}]\);
  for \(q \in \text{HO}(p, r)\) do rcvd\([q]\) := receive\((q, p, r)\) enddo;
  \(st, r := T^r_p(st, rcvd), r + 1\);
  for \(q \in \text{Proc}\) do send\((p, q, r, S^r_p(st, q))\) enddo;
end loop
end process
```

- **Formally**: infinite sequence \(\xi = c_0c_1 \ldots\) of configurations
- **Infinite-state model, due to round numbers**
Outline

1. Reduction Theorems for the Verification of Concurrent Programs
2. Fault-Tolerant Distributed Computing
3. Reduction for Round-Based Distributed Algorithms
4. Experiments: Verification of Consensus Algorithms
5. Conclusion
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 (assuming infinite network capacity)

- This motivates the following reduction:

```plaintext
process Node(p ∈ Proc)
  ⟨
  state st = s_{0,p};
  integer r = 0;
  for q ∈ Proc do send(p, q, r, S^r_p(st, q)) enddo;
  loop
    ⟨
      array rcvd = [q ∈ Proc ↵ null];
      for q ∈ HO(p, r) do rcvd[q] := receive(q, p, r) enddo;
      st, r := T^r_p(st, rcvd), r + 1;
      for q ∈ Proc do send(p, q, r, S^r_p(st, q)) enddo;
    end loop
  end loop
end process
```
More reduction

- Processes execute rounds atomically

- 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 | ...
```

- Can we do any better?

- Remember communication-closed rounds
  - round \( \text{rnd}_p^m \) right-commutes with \( \text{rnd}_q^n \) if \( m > n \)
  - messages sent during \( \text{rnd}_q^n \) did not influence \( \text{rnd}_p^m \)

- 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 | ...
```
More reduction

- Processes execute rounds atomically

\[
\begin{array}{cccccccccc}
\text{init} & \text{init} & \text{rnd} \ 0 & \text{init} & \text{rnd} \ 0 & \text{rnd} \ 1 & \text{rnd} \ 0 & \text{rnd} \ 1 & \text{rnd} \ 2 & \text{rnd} \ 1 \\
\end{array}
\]

- Can we do any better?

- Remember communication-closed rounds
  - round \( \text{rnd}^m_p \) right-commutes with \( \text{rnd}^n_q \) if \( m > n \)
  - messages sent during \( \text{rnd}^n_q \) did not influence \( \text{rnd}^m_p \)

- Rearrange execution so that executions of same round are adjacent

\[
\begin{array}{cccccccccc}
\text{init} & \text{init} & \text{init} & \text{rnd} \ 0 & \text{rnd} \ 0 & \text{rnd} \ 0 & \text{rnd} \ 1 & \text{rnd} \ 1 & \text{rnd} \ 1 & \text{rnd} \ 2 \\
\end{array}
\]

- 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** \( \sigma_0 \sigma_1 \ldots \)  
  \[ (\sigma_i : \text{Proc} \rightarrow \text{State}) \]
  - \( \sigma_0(p) = s_{0,p} \)
  - \( \sigma_{r+1}(p) = T^r_p(\sigma_r(p), \text{rcvd}(p,r)) \)

  where \( \text{rcvd}(p,r) = [q \in HO(p,r) \mapsto S^r_q(\sigma_r(q),p)] \)

- **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 \( \sigma_0\sigma_1\ldots \) (\( \sigma_i : \text{Proc} \rightarrow \text{State} \))
  - \( \sigma_0(p) = s_{0,p} \)
  - \( \sigma_{r+1}(p) = T^r_p(\sigma_r(p), rcvd(p, r)) \)
  
    where \( rcvd(p, r) = [q \in HO(p, r) \mapsto S^r_q(\sigma_r(q), p)] \)

- 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 $\zeta = c_0 c_1 \ldots$
    \[ \zeta^p = c_0.st(p), c_1.st(p), \ldots \]
  - $p$-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
  \[ \zeta \approx \sigma \iff \mathbb{L}(\zeta^p) = \mathbb{L}(\sigma^p) \text{ for every } p \in \text{Proc} \]
  - Local views equal up to stuttering, for every process
Theorem (Reduction)

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

Proof. For \(\xi = c_0c_1 \ldots\), define sequence \(\sigma = ([p \in \text{Proc} \mapsto c_{p, \ell_p} \cdot \text{st}(p)])_{r \in \mathbb{N}}\) where

\[
\begin{align*}
\ell_0^p &= 0 \\
\ell_{r+1}^p &= k + 1 \quad \text{if} \ (c_k, c_{k+1}) \text{ is } (r + 1)\text{st local transition of } p.
\end{align*}
\]

Then \(\sigma\) is a coarse-grained execution for the same HO collection.

Moreover, \(\sharp(\sigma^p) = \sharp(\xi^p)\) for all \(p \in \text{Proc}\). 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

\[ \rho_1 \models P \iff \rho_2 \models P \quad \text{whenever } \rho_1 \approx \rho_2 \]
“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

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

- 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:

  $$\bigwedge_{p,q \in \text{Proc}} \Box (\text{rnd}_p = \text{rnd}_q) \quad \text{(where } \text{rnd}_p \text{ is the current round of } p)$$
Consensus as a local property

- **Integrity**

\[ \bigwedge_{p \in \text{Proc}} \forall v \neq \text{null} : \left( \Diamond (\text{decide}_p = v) \Rightarrow \bigvee_{q \in \text{Proc}} x_q = v \right) \]

- **Irrevocability**

\[ \bigwedge_{p \in \text{Proc}} \forall v \neq \text{null} : \Box (\text{decide}_p = v \Rightarrow \Box (\text{decide}_p = v)) \]

- **Agreement**

\[ \bigwedge_{p,q \in \text{Proc}} \forall v, w \neq \text{null} : \Diamond (\text{decide}_p = v) \land \Diamond (\text{decide}_q = w) \Rightarrow v = w \]

- **Termination**

\[ \bigwedge_{p \in \text{Proc}} \Diamond (\text{decide}_p \neq \text{null}) \]
Outline

1. Reduction Theorems for the Verification of Concurrent Programs
2. Fault-Tolerant Distributed Computing
3. Reduction for Round-Based Distributed Algorithms
4. Experiments: Verification of Consensus Algorithms
5. Conclusion
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

\[\text{MODULE \textit{HeardOf}}\]

\textbf{EXTENDS Naturals}

\textbf{CONSTANTS} \textit{Proc, State, Msg, \textit{nPhases}, \textit{IniSt}(.), \textit{Send}(.,.,.,.), \textit{Trans}(.,.,.,.)}

\textbf{VARIABLES} \textit{phase, state, heardof}

\begin{align*}
\text{\textit{Init}} & \triangleq \land \text{phase} = 0 \\
& \qquad \land \text{state} = [p \in \textit{Proc} \mapsto \textit{IniSt}(p)] \\
& \qquad \land \text{heardof} = [p \in \textit{Proc} \mapsto \{\}]
\end{align*}

\begin{align*}
\text{\textit{Step}(\textit{HO})} & \triangleq \text{LET } \text{rcvd}(p) \triangleq \{\langle q, \textit{Send}(q, \textit{phase}, \textit{state}[q], p) \rangle : q \in \textit{HO}[p]\} \\
& \quad \text{IN } \land \text{phase}' = (\text{phase} + 1) \% \textit{nPhases} \\
& \qquad \land \text{state}' = [p \in \textit{Proc} \mapsto \textit{Trans}(p, \textit{phase}, \textit{state}[p], \textit{rcvd}(p))] \\
& \qquad \land \text{heardof}' = \textit{HO}
\end{align*}

\begin{align*}
\text{\textit{Next}} & \triangleq \exists \textit{HO} \in [\textit{Proc} \rightarrow \textbf{SUBSET} \textit{Proc}] : \textit{Step}(\textit{HO}) \\
\text{\textit{NoSplit}(\textit{HO})} & \triangleq \forall p, q \in \textit{Proc} : \textit{HO}[p] \cap \textit{HO}[q] \neq \{\} \\
\text{\textit{NextNoSplit}} & \triangleq \exists \textit{HO} \in [\textit{Proc} \rightarrow \textbf{SUBSET} \textit{Proc}] : \textit{NoSplit}(\textit{HO}) \land \textit{Step}(\textit{HO}) \\
\text{\textit{Uniform}(\textit{HO})} & \triangleq \exists S \in \textbf{SUBSET} \textit{Proc} : \textit{HO} = [q \in \textit{Proc} \mapsto S] \\
\text{\textit{InfiniteUniform}} & \triangleq \Box \lozenge \textit{Uniform(heardof)}
\end{align*}
Remarks

- Definitions closely parallels “paper” version
  - expressiveness of TLA\(^+\) leads to perspicuous formulation
  - (auxiliary) variable \textit{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

\[ nPhases \triangleq 1 \]

\[ Proc \triangleq 1..N \]

\[ InitValue(p) \triangleq 10 \times p \]

\[ Value \triangleq \{InitValue(p) : p \in Proc\} \]

\[ Msg \triangleq Value \]

\[ null \triangleq 0 \]

\[ ValueOrNull \triangleq Value \cup \{null\} \]

\[ State \triangleq [x : Value, decide : ValueOrNull] \]

- definition of constant parameters for *OneThirdRule* algorithm
- arbitrary definition of (initial) values of a process
One-Third Rule in TLA+ (2/3)

\[
\text{IniSt}(p) \triangleq [x \mapsto \text{InitValue}(p), \text{decide} \mapsto \text{null}]
\]

\[
\text{Send}(p, ph, s, q) \triangleq s.x
\]

\[
\text{Trans}(p, ph, s, rcvd) \triangleq
\]

\[
\text{IF } \text{Cardinality}(rcvd) > (2 \ast N) \div 3
\]

\[
\text{THEN LET } \text{Freq}(v) \triangleq \text{Cardinality}\{q \in \text{Proc} : \langle q, v \rangle \in rcvd\}
\]

\[
\text{MFR}(v) \triangleq \forall w \in \text{Value} : \text{Freq}(w) \leq \text{Freq}(v)
\]

\[
\min \triangleq \text{CHOOSE } v \in \text{Value} : \text{MFR}(v) \land \forall w \in \text{Value} : \text{MFR}(w) \Rightarrow v \leq w
\]

\[
\text{IN } [x \mapsto \min, \text{decide} \mapsto \text{IF } \text{Freq}(\min) > (2 \ast N) \div 3 \text{ THEN } \min \text{ ELSE } s.\text{decide}]
\]

\[
\text{ELSE } s
\]


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

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

Safety $\triangleq$ $Init \land \square [\text{Next}]_{vars}$

Liveness $\triangleq$ $\square \Diamond (\text{Uniform(heardof)} \land \text{Cardinality(heardof)} > (2 \ast N) \div 3)$

Integrity $\triangleq$ $\forall p \in \text{Proc} : \text{state}[p].\text{decide} \in \text{ValueOrNull}$

Irrevocability $\triangleq$ $\forall p \in \text{Proc} : \square [\text{state}[p].\text{decide} = \text{null} ]_{\text{state}[p].\text{decide}}$

Agreement $\triangleq$ $\forall p, q \in \text{Proc} : (\text{state}[p].\text{decide} \neq \text{null} \land \text{state}[q].\text{decide} \neq \text{null} \Rightarrow \text{state}[p].\text{decide} = \text{state}[q].\text{decide})$

Termination $\triangleq$ $\forall p \in \text{Proc} : \Diamond (\text{state}[p].\text{decide} \neq \text{null})$

THEOREM Safety $\Rightarrow \square (\text{Integrity} \land \text{Agreement}) \land \text{Irrevocability}$

THEOREM Safety $\land$ Liveness $\Rightarrow$ Termination

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

<table>
<thead>
<tr>
<th></th>
<th>OneThirdRule</th>
<th></th>
<th>UniformVoting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N = 3$</td>
<td>$N = 4$</td>
<td>$N = 3$</td>
</tr>
<tr>
<td>states</td>
<td>5633</td>
<td>9,830,401</td>
<td>21,351</td>
</tr>
<tr>
<td>distinct</td>
<td>11</td>
<td>150</td>
<td>122</td>
</tr>
<tr>
<td>time (s)</td>
<td>1.87</td>
<td>939</td>
<td>13.8</td>
</tr>
</tbody>
</table>

- 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

```isabelle
locale HOAlgorithm = 
  fixes 
  nPhases :: nat and
  initSt :: 'proc → 'pst and
  send :: 'proc → nat → 'pst → 'proc → 'msg and
  trans :: 'proc → nat → 'pst → ('proc → 'msg) → 'pst
  assumes 
  nSteps : 0 < nPhases and
  finiteProc : finite(UNIV :: 'procset)
```

- 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

1. Reduction Theorems for the Verification of Concurrent Programs
2. Fault-Tolerant Distributed Computing
3. Reduction for Round-Based Distributed Algorithms
4. Experiments: Verification of Consensus Algorithms
5. Conclusion
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