Formal Verification of Fault-Tolerant Distributed Algorithms

Stephan Merz

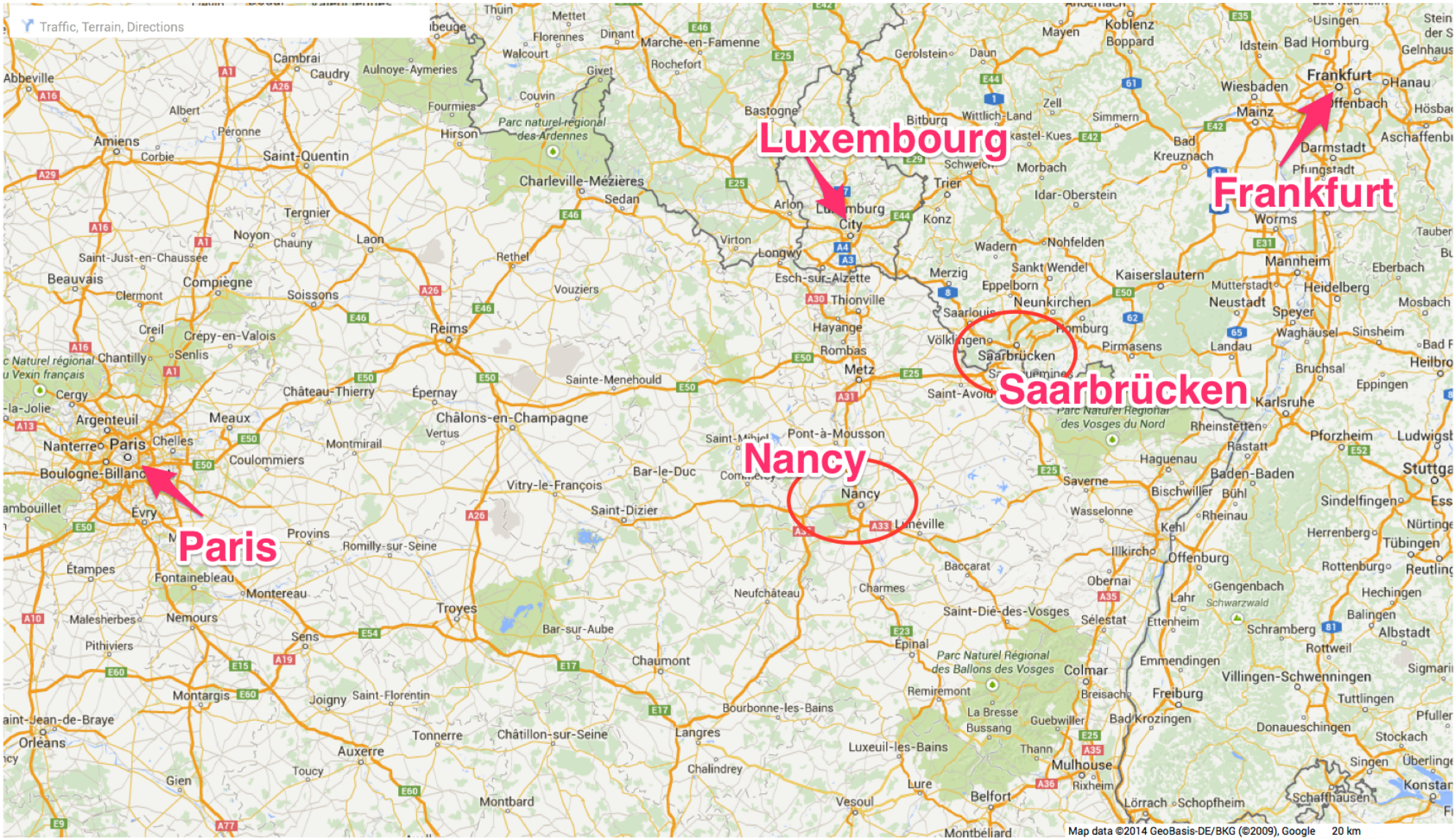
joint work with Bernadette Charron-Bost and Henri Debrat

INRIA Nancy & LORIA

CDZ/NoDI Seminar
Beijing, November 2014

Where are we (1/2)?

Where are we (2/2)?



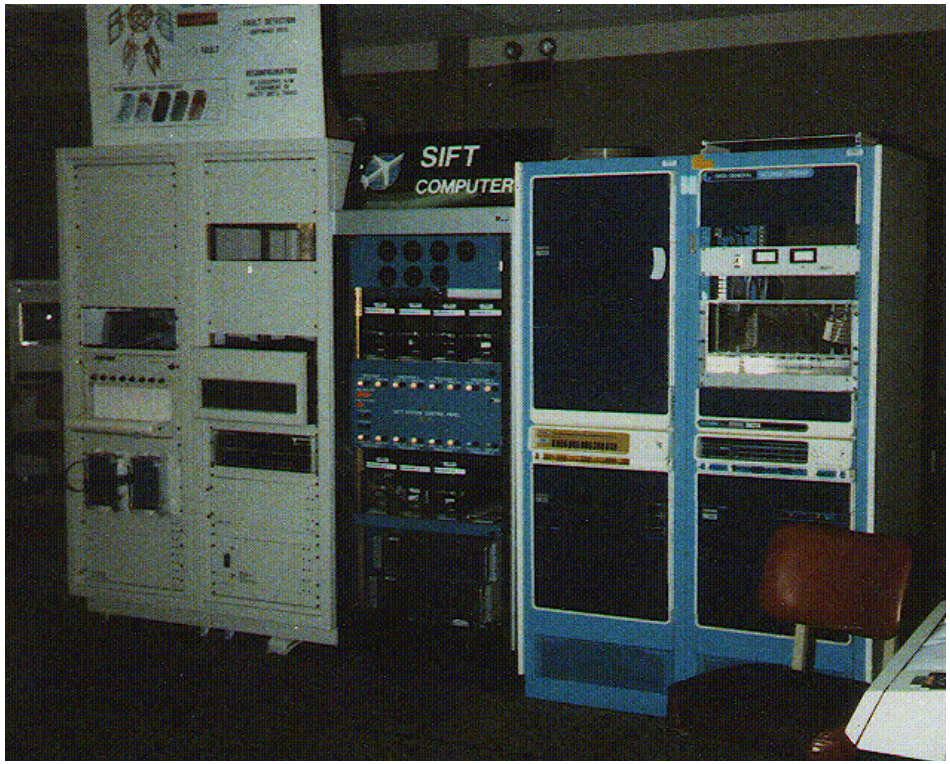
Inria VeriDis team

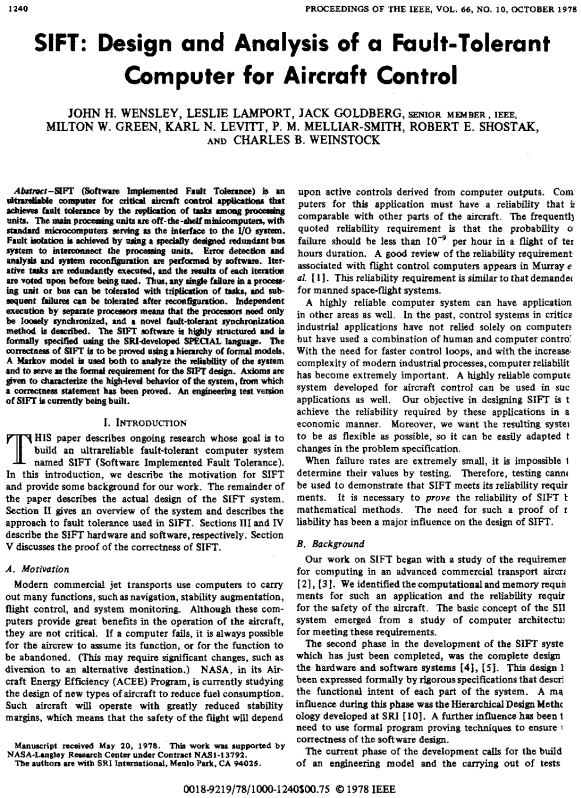
- Formal reasoning for verification
  - automatic theorem proving, decision procedures, and combinations
  - integration with interactive proof platforms, computer algebra, ...
  - main application: verification of programs and systems
- Target domain: distributed algorithms and systems
  - high-level specifications: (Event-)B, TLA$^+$
  - (statistical) model checking of distributed C programs
- Tool development
  - Spass theorem prover, veriT SMT solver
  - Redlog: reasoning about non-linear real arithmetic
  - TLA$^+$ Proof System (MSR-Inria Joint Centre)

Outline

1. Fault-Tolerant Distributed Algorithms in the Heard-Of Model
2. Representing Executions of HO Algorithms
3. Formal Proofs in Isabelle/HOL
4. Summing Up

The Origins of Fault-Tolerant Computing





1240 PROCEEDINGS OF THE IEEE, VOL. 66, NO. 10, OCTOBER 1978

SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control

JOHN H. WENSLEY, LESLIE LAMPORT, JACK GOLDBERG, SENIOR MEMBER, IEEE, MILTON W. GREEN, KARL N. LEVITT, P. M. MELLIAR-SMITH, ROBERT E. SHOSTAK, AND CHARLES B. WEINSTOCK

Abstract—SIFT (Software Implemented Fault Tolerance) is an ultrareliable computer for critical aircraft control applications that achieves fault tolerance by the replication of tasks among processing units. The main processing units are off-the-shelf minicomputers, with standard microcomputers serving as the interface to the I/O system. Fault isolation is achieved by using a specially designed redundant bus system to interconnect the processing units. Error detection and analysis and system reconfiguration are performed by software. Iterative tasks are redundantly executed, and the results of each iteration are voted upon before being used. Thus, any single failure in a processing unit or bus can be tolerated with triplication of tasks, and subsequent failures can be tolerated after reconfiguration. Independent execution by separate processors means that the processors need only be loosely synchronized, and a novel fault-tolerant synchronization method is described. The SIFT software is highly structured and is formally specified using the SRI-developed SPECIAL language. The correctness of SIFT is to be proved using a hierarchy of formal models. A Markov model is used both to analyze the reliability of the system and to serve as the formal requirement for the SIFT design. Axioms are given to characterize the high-level behavior of the system, from which a correctness statement has been proved. An engineering test version of SIFT is currently being built.

I. INTRODUCTION

THIS paper describes ongoing research whose goal is to build an ultrareliable fault-tolerant computer system named SIFT (Software Implemented Fault Tolerance). In this introduction, we describe the motivation for SIFT and provide some background for our work. The remainder of the paper describes the actual design of the SIFT system. Section II gives an overview of the system and describes the approach to fault tolerance used in SIFT. Sections III and IV describe the SIFT hardware and software, respectively. Section V discusses the proof of the correctness of SIFT.

A. Motivation

Modern commercial jet transports use computers to carry out many functions, such as navigation, stability augmentation, flight control, and system monitoring. Although these computers provide great benefits in the operation of the aircraft, they are not critical. If a computer fails, it is always possible for the aircrew to assume its function, or for the function to be abandoned. (This may require significant changes, such as diversion to an alternative destination.) NASA, in its Aircraft Energy Efficiency (ACEE) Program, is currently studying the design of new types of aircraft to reduce fuel consumption. Such aircraft will operate with greatly reduced stability margins, which means that the safety of the flight will depend upon active controls derived from computer outputs. Com puters for this application must have a reliability that is comparable with other parts of the aircraft. The frequently quoted reliability requirement is that the probability o failure should be less than 10^{-9} per hour in a flight of ten hours duration. A good review of the reliability requirement associated with flight control computers appears in Murray *et al.* [1]. This reliability requirement is similar to that demanded for manned space-flight systems.

A highly reliable computer system can have application in other areas as well. In the past, control systems in critica industrial applications have not relied solely on computers but have used a combination of human and computer contro With the need for faster control loops, and with the increase complexity of modern industrial processes, computer reliabilit has become extremely important. A highly reliable compute system developed for aircraft control can be used in suc applications as well. Our objective in designing SIFT is t achieve the reliability required by these applications in a economic manner. Moreover, we want the resulting syste to be as flexible as possible, so it can be easily adapted t changes in the problem specification.

When failure rates are extremely small, it is impossible t determine their values by testing. Therefore, testing canne be used to demonstrate that SIFT meets its reliability requir ments. It is necessary to *prove* the reliability of SIFT b mathematical methods. The need for such a proof of r liability has been a major influence on the design of SIFT.

B. Background

Our work on SIFT began with a study of the requiremer for computing in an advanced commercial transport aircra [2], [3]. We identified the computational and memory requir ments for such an application and the reliability requir for the safety of the aircraft. The basic concept of the SI system emerged from a study of computer architectu for meeting these requirements.

The second phase in the development of the SIFT syste which has just been completed, was the complete design the hardware and software systems [4], [5]. This design h been expressed formally by rigorous specifications that descri the functional intent of each part of the system. A ma influence during this phase was the Hierarchical Design Metho ology developed at SRI [10]. A further influence has been t need to use formal program proving techniques to ensure correctness of the software design.

The current phase of the development calls for the build of an engineering model and the carrying out of tests

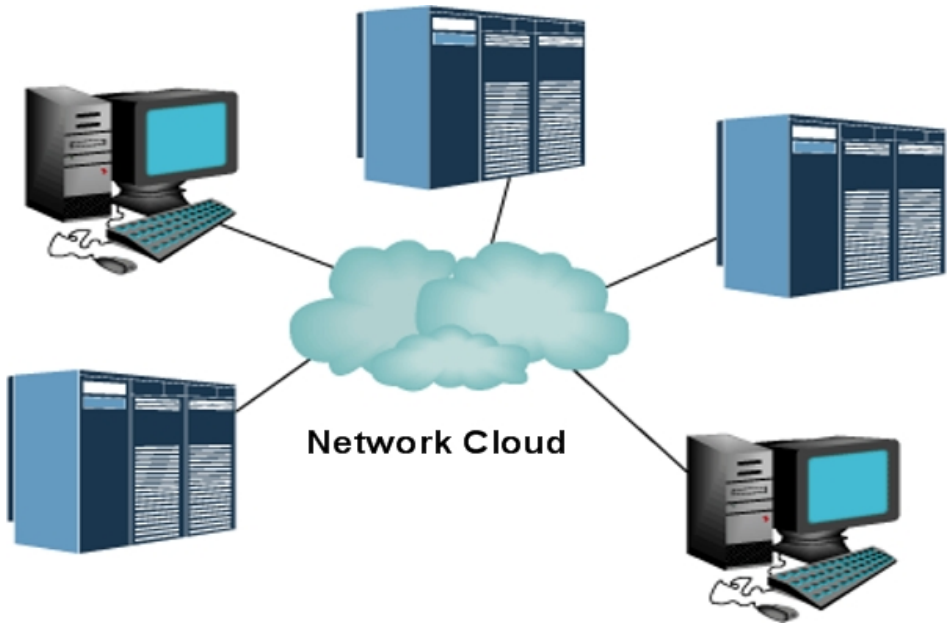
Manuscript received May 20, 1978. This work was supported by NASA-Langley Research Center under Contract NAS1-13792.
The authors are with SRI International, Menlo Park, CA 94025.

0018-9219/78/1000-1240$00.75

- replication for fault tolerance
- synchronous, round-based model of computation

image courtesy John Rushby, SRI

Fault-Tolerance in Distributed Systems



- local computation & asynchronous communication
- components may fail: redundancy for fault-tolerance
- formally express and prove correctness properties

The Consensus Problem

- 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, in different ways
- Formally: conjunction of four properties

Integrity	decisions are among the initial proposals
Irrevocability	decisions cannot be undone
Agreement	any two nodes decide the same value
Termination	all (non-failed) nodes decide eventually

- Paradigmatic problem in fault-tolerant distributed computing

Why Is This Difficult?

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
  - partially synchronous computation
  - timeouts, reliable (broadcast) communication
  - augment system by oracles: failure detectors

Why Is This Difficult?

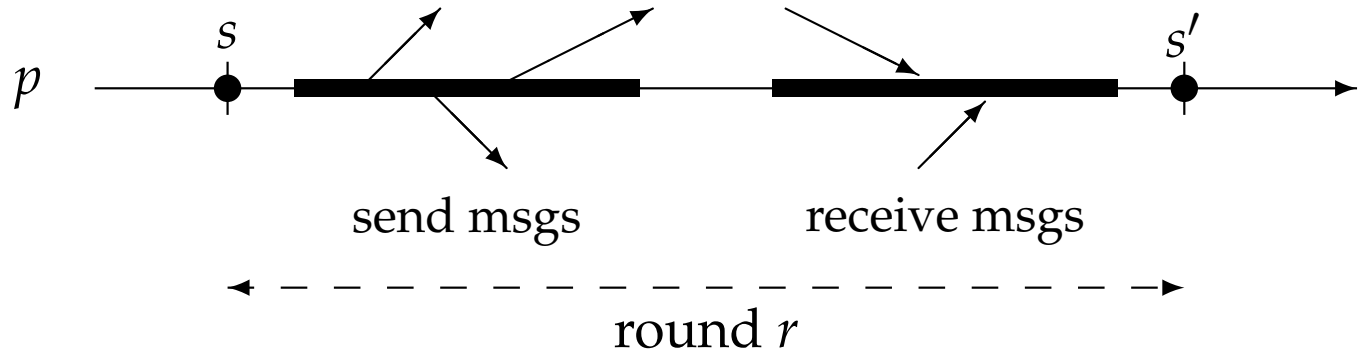
Theorem (Fischer, Lynch, Paterson 1985)

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
  - partially synchronous computation
  - timeouts, reliable (broadcast) communication
  - augment system by oracles: failure detectors
- Verification of Consensus algorithms
  - tricky correctness arguments ... often absent or informal
  - different models of computations and failures
  - few careful paper proofs (e.g., 30 pages for 1 page algorithm)
- How can we help make verification simpler?

Round-Based Distributed Algorithms

- Observation: algorithms are structured in "rounds"



  - process-local organization of computation
  - state s' computed from s and messages received
  - communication-closed rounds: discard late messages

- Heard-Of model (Charron-Bost & Schiper, 2009)
  - heard-of set $HO(p, r)$: processes from which messages are received
  - failure hypotheses described by predicates on HO sets
  - uniform representation of models of computation and faults

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 per round

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

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

  - 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
- Extension to Byzantine failures: value faults
  - additional collection of "safe" heard-of sets $SHO(p, r) \subseteq HO(p, r)$
  - reception of arbitrary values from processes in $HO(p, r) \setminus SHO(p, r)$

HO Consensus Algorithm: One-Third Rule

Initialization

$x_p := v_p$, $decide_p := null$ *(v_p : value initially proposed by p)*

For each round $r \geq 0$

S_p^r : send x_p to all processes

T_p^r : **if** $|HO(p,r)| > \frac{2}{3} N$ **then**

set x_p to smallest among the most frequently received values

if more than $\frac{2}{3} N$ values received are equal to x_p **then** $decide_p := x_p$

Simple but efficient Consensus algorithm

- resilient to crash faults of $< \frac{1}{3}$ of all nodes
- no coordinator needed
- quick convergence if few errors

Communication Predicates

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

- Communication predicates: examples

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	$\exists r_0 \in \mathbb{N}, P \subseteq Proc : \forall r \geq r_0, q \in Proc : HO(q, r) = P$

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

Outline

1. Fault-Tolerant Distributed Algorithms in the Heard-Of Model
2. Representing Executions of HO Algorithms
3. Formal Proofs in Isabelle/HOL
4. Summing Up

“Fine-Grained” Executions of HO Algorithms

- Verification for every HO collection $(HO(p,r))_{p \in Proc, r \in \mathbb{N}}$

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

"Fine-Grained" Executions of HO Algorithms

- Verification for every HO collection $(HO(p,r))_{p \in Proc, r \in \mathbb{N}}$

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: definition of state transition system
  - interleaving of actions of individual nodes
  - no particular representation of communication-closed rounds
  - infinite-state model, due to round numbers

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.

Example: Peterson's Algorithm

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** ¬req1 ∨ turn = 0;		wt_1: **await** ¬req0 ∨ 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 $s \xrightarrow{wt_1\ rq_0} t$

Example: Peterson's Algorithm

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

Example: Peterson's Algorithm

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

More Reduction Theorems

- Doeppner 1977: reduction for invariant proofs
  - similar to Lipton's theorem
  - record when control is inside the reduced block, adapt invariant
  - T.W. Doeppner. *Parallel program correctness through refinement.* POPL 1977.
- Lamport & Schneider 1988: generalize Doeppner
  - preserve invariants of reduced program in original one
  - explicitly reason about control being external to reduced block
  - L. Lamport, F. Schneider: *Pretending atomicity.* Tech. Report, 1988.
    E. Cohen, L. Lamport: *Reduction in TLA.* Concur 1998.
- Back 1989: reduction for total correctness
  - requires commutativity hypotheses for statements outside block
  - R. Back: *A method for refining atomicity in parallel programs.* PARLE 1989.
- Zwiers 1990s: closed communication layers
  - reduction for synchronous message-passing algorithms
  - W. Janssen, J. Zwiers: *Protocol design by layered decomposition.* FTRTFT 1992.

Application to Peterson's Algorithm

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

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

||

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

Reduction justified by Doeppner's theorem for mutual exclusion

HO Executions: First Reduction

- Application to fine-grained executions of HO algorithms
  - send actions are left movers
  - receive actions are right movers

(assuming infinite network capacity)

HO Executions: First Reduction

- Application to fine-grained executions of HO algorithms
  - send actions are left movers
  - receive actions are right movers

(assuming infinite network capacity)

- 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

HO Executions: Second Reduction

- Processes execute rounds atomically

- Can we reduce even further?

HO Executions: Second Reduction

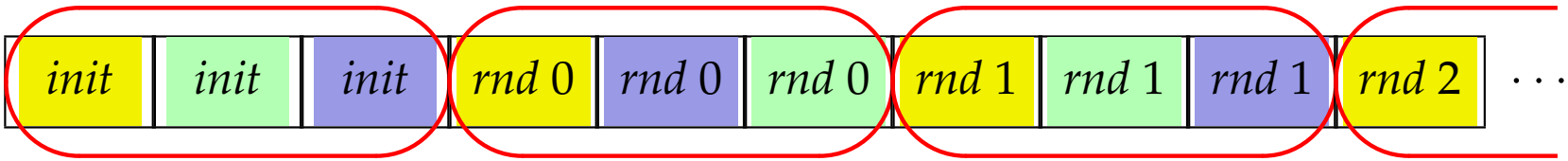
- Processes execute rounds atomically

- Can we reduce even further?
- Remember communication-closed rounds
  - round rnd_p^m left-commutes with rnd_q^n if $m < n$ (for fixed collection of HO sets)
  - messages sent during rnd_q^n did not influence rnd_p^m
- Rearrange execution so that executions of same round are adjacent

HO Executions: Second Reduction

- Processes execute rounds atomically

- Can we reduce even further?
- Remember communication-closed rounds
  - round rnd_p^m left-commutes with rnd_q^n if $m < n$ (for fixed collection of HO sets)
  - 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" Executions of HO Algorithms

- Sequence of configurations $\sigma_0\sigma_1\ldots$ $\quad(\sigma_i : Proc \to State)$
  - $\sigma_0(p) = s_{0,p}$
  - $\sigma_{r+1}(p) = T_p^r\big(\sigma_r(p), rcvd(p,r)\big)$

    where $\quad rcvd(p,r) = [q \in HO(p,r) \mapsto S_q^r(\sigma_r(q), p)]$

- Unit of atomicity: entire system rounds
  - all processes simultaneously perform transition for same round
  - no need for explicit representation of network or round numbers

"Coarse-Grained" Executions of HO Algorithms

- Sequence of configurations $\sigma_0\sigma_1\ldots$ $\quad (\sigma_i : Proc \to State)$
  - $\sigma_0(p) = s_{0,p}$
  - $\sigma_{r+1}(p) = T_p^r\big(\sigma_r(p), rcvd(p,r)\big)$

    where $\quad rcvd(p,r) = [q \in HO(p,r) \mapsto S_q^r(\sigma_r(q), p)]$

- Unit of atomicity: entire system rounds
  - all processes simultaneously perform transition for same round
  - no need for explicit representation of network or round numbers

- How do the two models of executions relate formally?
  - what properties are preserved by the reduction?

Relating Fine- and Coarse-Grained Executions

- 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 identical up to stuttering, for every process

Reduction Theorem

Theorem

Given a HO collection and a compatible fine-grained execution τ there exists a compatible coarse-grained execution σ such that $\sigma \approx \xi$.

Reduction Theorem

Theorem

Given a HO collection and a compatible fine-grained execution τ there exists a compatible coarse-grained execution σ such that $\sigma \approx \xi$.

Corollary

Any stuttering invariant property that depends only on local views can be verified over coarse-grained executions.

- "Local" LTL-X formulas (sufficient syntactic criterion)
  - any formula containing solely state variables of a single process
  - first-order combinations of local properties
  - non-local: variables of different processes below temporal operator

$$\forall p, q \in Proc : \Box(rnd[p] = rnd[q]) \qquad rnd[p]\text{: current round of } p$$

Consensus as a Local Property

- Integrity

$$\forall p \in Proc, v \in Val : \big(\Diamond(decide[p] = v) \Rightarrow \exists q \in Proc : x[q] = v\big)$$

- Irrevocability

$$\forall p \in Proc, v \in Val : \Box\big(decide[p] = v \Rightarrow \Box(decide[p] = v)\big)$$

- Agreement

$$\forall p, q \in Proc, v, w \in Val : \Diamond(decide[p] = v) \wedge \Diamond(decide[q] = w) \Rightarrow v = w$$

- Termination

$$\forall p \in Proc : \Diamond(decide[p] \in Val)$$

Outline

1. Fault-Tolerant Distributed Algorithms in the Heard-Of Model
2. Representing Executions of HO Algorithms
3. Formal Proofs in Isabelle/HOL
4. Summing Up

Representation of HO Algorithms in Isabelle (1)

- Uniform encoding of algorithms and their executions
  - generic definition: coordinated algorithms, Byzantine failures
  - other variants derived as particular instances

record $('proc, 'pst, 'msg)$ CHOAlgorithm $=$
  CinitState :: $['proc, 'pst, 'proc] \Rightarrow$ bool
  sendMsg :: $[\text{nat}, 'proc, 'proc, 'pst] \Rightarrow 'msg$
  CnextState :: $[\text{nat}, 'proc, 'pst, 'proc \Rightarrow 'msg\ \text{option}, 'proc, 'pst] \Rightarrow$ bool

definition CSHOnextConfig **where** CSHOnextConfig *alg r cfg HO SHO coord cfg'* $\equiv$
  $\forall p.\ \exists \mu \in$ SHOmsgVectors *alg r p cfg* $(HO\ p)$ $(SHO\ p)$.
    CnextState *alg r p* $(cfg\ p)$ μ $(coord\ p)$ $(cfg'\ p)$

definition CSHORun **where** CSHORun *alg rho HOs SHOs coords* $\equiv$
  CHOinitConfig *alg* $(rho\ 0)$ $(coords\ 0)$ $\wedge$
  $(\forall r.$ CSHOnextConfig *alg r* $(rho\ r)$ $(HOs\ r)$ $(SHOs\ r)$ $(coords\ (Suc\ r))$ $(rho\ (Suc\ r)))$

Representation of HO Algorithms in Isabelle (2)

- Concrete algorithms defined as instances

record $'val\ pstate\ =\ (\mathrm{x} :: 'val)\ (\mathrm{dec} :: 'val\ \mathrm{option})$

definition initState **where** $\mathrm{initState}\ p\ st\ \equiv\ \mathrm{dec}\ st\ =\ \mathrm{None}$

definition sendMsg **where** $\mathrm{sendMsg}\ r\ st\ \equiv\ \mathrm{x}\ st$

definition HOV **where** $\mathrm{HOV}\ \mu\ v\ \equiv\ \{\, q\ .\ \mu\ q\ =\ \mathrm{Some}\ v\,\}$

definition MFR **where** $\mathrm{MFR}\ \mu\ v\ \equiv\ \forall w.\ \mathrm{card}\ (\mathrm{HOV}\ \mu\ w)\ \leq\ \mathrm{card}\ (\mathrm{HOV}\ \mu\ v)$

definition TwoThirds **where** $\mathrm{TwoThirds}\ \mu\ v\ \equiv\ (2 * \mathrm{N})\ \mathrm{div}\ 3\ <\ \mathrm{card}\ (\mathrm{HOV}\ \mu\ v)$

definition nextState **where** $\mathrm{nextState}\ r\ p\ st\ \mu\ st'\ \equiv$

$\quad \mathrm{if}\ (2 * \mathrm{N})\ \mathrm{div}\ 3\ <\ \mathrm{card}\ \{\, q\ .\ \mu\ q\ \neq\ \mathrm{None}\,\}$

$\quad \mathrm{then}\ st'\ =\ (\!|\, \mathrm{x}\ =\ \mathrm{Min}\ \{\, v\ .\ \mathrm{MFR}\ \mu\ v\,\},$

$\qquad\qquad \mathrm{dec}\ =\ (\mathrm{if}\ (\exists v.\ \mathrm{TwoThirds}\ \mu\ v)$

$\qquad\qquad\qquad\qquad \mathrm{then\ Some}\ (\epsilon v.\ \mathrm{TwoThirds}\ \mu\ v)$

$\qquad\qquad\qquad\qquad \mathrm{else\ dec}\ st)\ |\!)$

$\quad \mathrm{else}\ st'\ = st$

Results: Meta-level

- Formal proof of reduction theorem

theorem reduction :
assumes fg-run *alg* ρ *HOs SHOs coords*
shows CSHORun *alg* (coarse-run ρ) *HOs SHOs coords*

theorem coarse-run-locally-similar :
assumes fg-run *alg* ρ *HOs SHOs coords*
shows locally-similar (state $\circ$ ρ) (coarse-run ρ)

- Corollary on verification of local properties

theorem local-property-reduction :
assumes fg-run *alg* ρ *HOs SHOs coords* **and** local-property P
and $\forall\sigma$. CSHORun *alg* σ *HOs SHOs coords* $\wedge$ $\sigma\ 0 = state\ (\rho\ 0) \longrightarrow P\ \sigma$
shows P (state $\circ$ ρ)

theorem consensus-local : local-property (consensus *vals dec*)

Results: Algorithms

- Verification of six algorithms

Algorithm	synchrony	failures	coordinated
OneThirdRule	partial	benign	no
UniformVoting	partial	benign	no
LastVoting	partial	benign	yes
$\mathcal{U}_{T,E}$	partial	Byzantine	no
$\mathcal{A}_{T,E}$	partial	Byzantine	no
EIGbyz$_f$	full	Byzantine	no

- Precision of failure hypotheses
  - identification of properties established by hypotheses
  - default value for agreement (EIGbyz$_f$) or termination ($\mathcal{U}_{T,E}$)
  - EIGbyz$_f$ tolerates certain dynamic faults even when $N \leq 3f$

Outline

1. Fault-Tolerant Distributed Algorithms in the Heard-Of Model
2. Representing Executions of HO Algorithms
3. Formal Proofs in Isabelle/HOL
4. Summing Up

Lessons Learnt / Perspectives

- Choose suitable computational model before verification
  - take advantage of fundamental concept of algorithm design
  - significant ($\sim$ 1 order of magnitude) reduction of proof effort
  - general framework: uniform encoding, comparison, proof reuse
  - also useful for model checking: reduction to finite-state model (for fixed numbers of processes)
- Ongoing / future work
  - extension to probabilistic algorithms or executions
  - explicit representation of transition faults
  - improved proof automation
- Available at `afp.sf.net/entries/Heard_Of.shtml`