Le Centre de Recherche Commun INRIA-Microsoft Research

Jean-Jacques Lévy

INRIA Rocquencourt & MSR-INRIA Joint Centre

ENST
Mercredi 4 avril 2007

Plan

1. Context
2. Track A
3. Track B
4. Future

Context

INRIA

- Rocquencourt, Sophia-Antipolis, Rennes, Grenoble, Nancy, Futurs (Bordeaux, Lille, Saclay) $\simeq$ 500 chercheurs.
- premier institut de recherche européen en informatique.
  40 ans en 2007.
- Automatique et Informatique.

Jacques-Louis Lions

Microsoft Research

- Redmond, Cambridge, Pékin, Silicon Valley, Bangalore $\simeq$ 700 chercheurs.
- recherche ouverte (publications, logiciels) et principalement fondamentale, 15 ans en 2007.
- ouverture vers les universités/instituts de recherche (Trente, Aix-la-Chapelle, INRIA, Carnegie-Mellon)
- même directeur depuis 15 ans.

Rick Rashid

Politics

INRIA

Gilles Kahn

Michel Cosnard

MSR Cambridge

Roger Needham

Andrew Herbert

Joint Centre

Gérard Huet
↪ J.-J. Lévy

Stephen Emmott
Gérard Giraudon
Jean Vuillemin
Ken Wood

Strong points in french CS research

mathematics and theoretical CS

formal methods	b, coq, trusted logic
programming langages	ada, caml, lelisp, lustre, esterel
computer algebra	maple libraries, scilab
computer human interfaces	nextStep, Mac OS X interface
computational geometry	CGAL
vision	realviz
· · · INRIA · · ·	ilog, altavista · · · exalead
basic software (prototypes and real tools)	polyspace, astree, unison
	⋮

Strong points in french CS research

mathematics and theoretical CS

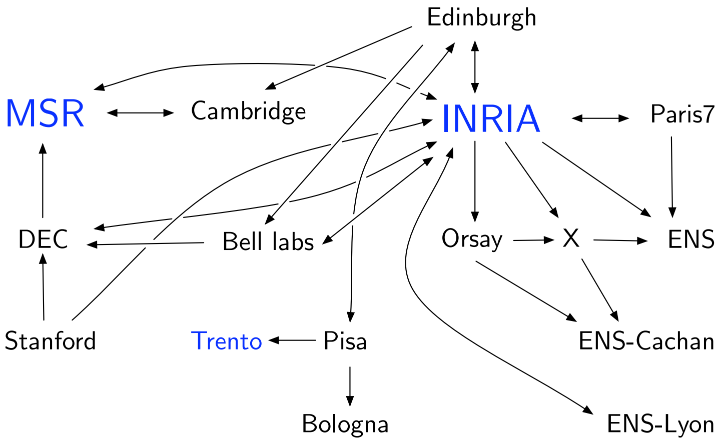
formal methods	b, coq, trusted logic
programming langages	ada, caml, lelisp, lustre, esterel
computer algebra	maple libraries, scilab
computer human interfaces	nextStep, Mac OS X interface
computational geometry	CGAL
vision	realviz
··· INRIA ···	ilog, altavista ··· exalead
basic software (prototypes and real tools)	polyspace, astree, unison ⋮

Strong points in french CS research

formal thinking = theory + *hacking*

formal methods	b, coq, trusted logic
programming langages	ada, caml, lelisp, lustre, esterel
computer algebra	maple libraries, scilab
computer human interfaces	nextStep, Mac OS X interface
computational geometry	CGAL
vision	realviz
··· INRIA ···	ilog, altavista ··· exalead
basic software (prototypes and real tools)	polyspace, astree, unison ⋮

Long cooperation between researchers



Track A

Software Security

Trustworthy Computing

Mathematical components

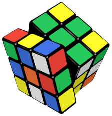
Georges Gonthier, MSR
Assia Mahboubi, INRIA-MSR
Enrico Tassi, Bologna
Y. Bertot, L. Rideau, INRIA Sophia
Sean McLaughlin, Carnegie Mellon
Benjamin Werner, INRIA Futurs
Roland Zumkeller, LIX

Computational proofs

- computer assistance for long formal proofs.
- Georges Gonthier proved 4-color theorem in 2001-2004 (60000 lines of Coq).

4-color

Appel-Haken

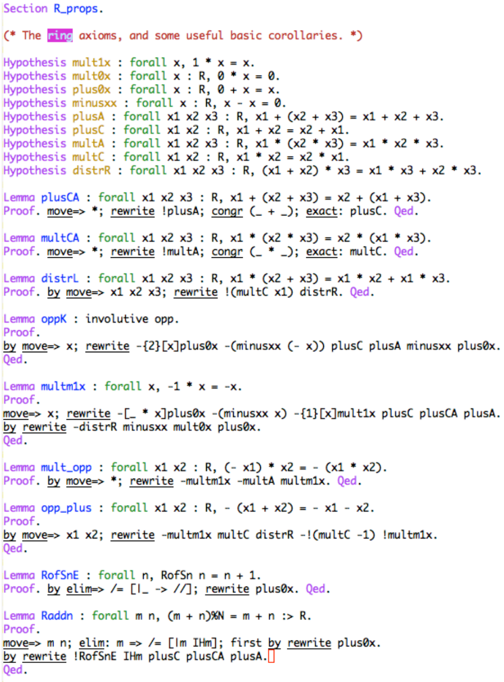


finite groups

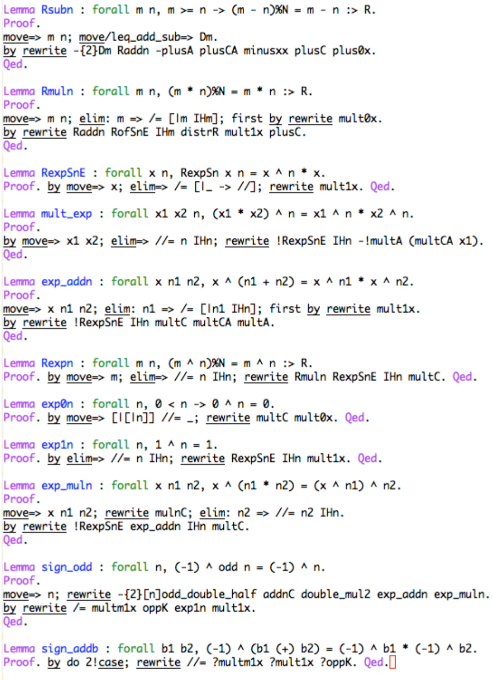
Feit-Thompson

Kepler

Hales

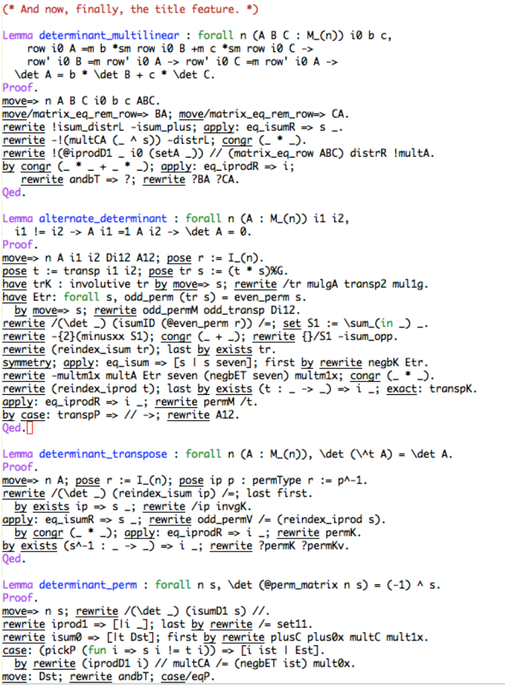


```
Section R_props.

(* The ring axioms, and some useful basic corollaries. *)

Hypothesis mult1x : forall x, 1 * x = x.
Hypothesis mult0x : forall x : R, 0 * x = 0.
Hypothesis plus0x : forall x : R, 0 + x = x.
Hypothesis minusxx : forall x : R, x - x = 0.
Hypothesis plusA : forall x1 x2 x3 : R, x1 + (x2 + x3) = x1 + x2 + x3.
Hypothesis plusC : forall x1 x2 : R, x1 + x2 = x2 + x1.
Hypothesis multA : forall x1 x2 x3 : R, x1 * (x2 * x3) = x1 * x2 * x3.
Hypothesis multC : forall x1 x2 : R, x1 * x2 = x2 * x1.
Hypothesis distrR : forall x1 x2 x3 : R, (x1 + x2) * x3 = x1 * x3 + x2 * x3.

Lemma plusCA : forall x1 x2 x3 : R, x1 + (x2 + x3) = x2 + (x1 + x3).
Proof. move=> *; rewrite !plusA; congr (_ + _); exact: plusC. Qed.

Lemma multCA : forall x1 x2 x3 : R, x1 * (x2 * x3) = x2 * (x1 * x3).
Proof. move=> *; rewrite !multA; congr (_ * _); exact: multC. Qed.

Lemma distrL : forall x1 x2 x3 : R, x1 * (x2 + x3) = x1 * x2 + x1 * x3.
Proof. by move=> x1 x2 x3; rewrite !(multC x1) distrR. Qed.

Lemma oppK : involutive opp.
Proof.
by move=> x; rewrite -{2}[x]plus0x -(minusxx (- x)) plusC plusA minusxx plus0x.
Qed.

Lemma multm1x : forall x, -1 * x = -x.
Proof.
move=> x; rewrite -[_ * x]plus0x -(minusxx x) -{1}[x]mult1x plusC plusCA plusA.
by rewrite -distrR minusxx mult0x plus0x.
Qed.

Lemma mult_opp : forall x1 x2 : R, (- x1) * x2 = - (x1 * x2).
Proof. by move=> *; rewrite -multm1x -multA multm1x. Qed.

Lemma opp_plus : forall x1 x2 : R, - (x1 + x2) = - x1 - x2.
Proof.
by move=> x1 x2; rewrite -multm1x multC distrR -!(multC -1) !multm1x.
Qed.

Lemma RofSnE : forall n, RofSn n = n + 1.
Proof. by elim=> /= [|_ -> //]; rewrite plus0x. Qed.

Lemma Raddn : forall m n, (m + n)%N = m + n :> R.
Proof.
move=> m n; elim: m => /= [|m IHm]; first by rewrite plus0x.
by rewrite !RofSnE IHm plusC plusCA plusA.
Qed.
```

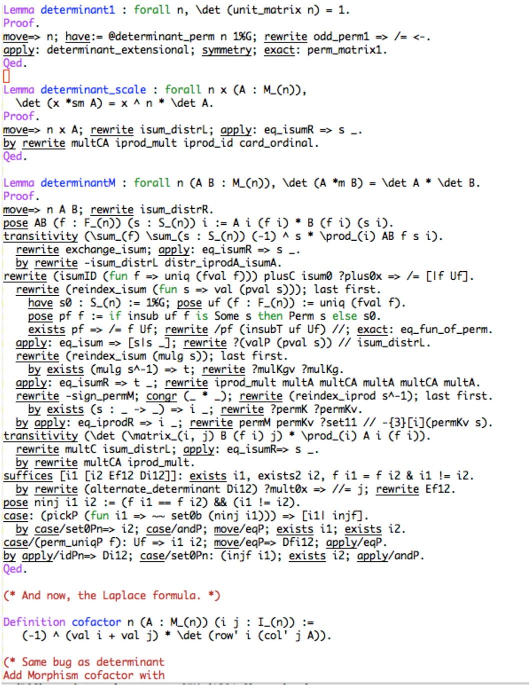


```
Lemma Rsubn : forall m n, m >= n -> (m - n)%N = m - n :> R.
Proof.
move=> m n; move/leq_add_sub=> Dm.
by rewrite -{2}Dm Raddn -plusA plusCA minusxx plusC plus0x.
Qed.

Lemma Rmuln : forall m n, (m * n)%N = m * n :> R.
Proof.
move=> m n; elim: m => /= [|m IHm]; first by rewrite mult0x.
by rewrite Raddn RofSnE IHm distrR mult1x plusC.
Qed.

Lemma RexpSnE : forall x n, RexpSn x n = x ^ n * x.
Proof. by move=> x; elim=> /= [|_ -> //]; rewrite mult1x. Qed.

Lemma mult_exp : forall x1 x2 n, (x1 * x2) ^ n = x1 ^ n * x2 ^ n.
Proof.
by move=> x1 x2; elim=> //= n IHn; rewrite !RexpSnE IHn -!multA (multCA x1).
Qed.

Lemma exp_addn : forall x n1 n2, x ^ (n1 + n2) = x ^ n1 * x ^ n2.
Proof.
move=> x n1 n2; elim: n1 => /= [|n1 IHn]; first by rewrite mult1x.
by rewrite !RexpSnE IHn multC multCA multA.
Qed.

Lemma Rexpn : forall m n, (m ^ n)%N = m ^ n :> R.
Proof. by move=> m; elim=> //= n IHn; rewrite Rmuln RexpSnE IHn multC. Qed.

Lemma exp0n : forall n, 0 < n -> 0 ^ n = 0.
Proof. by move=> [|[|n]] //= _; rewrite multC mult0x. Qed.

Lemma exp1n : forall n, 1 ^ n = 1.
Proof. by elim=> //= n IHn; rewrite RexpSnE IHn mult1x. Qed.

Lemma exp_muln : forall x n1 n2, x ^ (n1 * n2) = (x ^ n1) ^ n2.
Proof.
move=> x n1 n2; rewrite mulnC; elim: n2 => //= n2 IHn.
by rewrite !RexpSnE exp_addn IHn multC.
Qed.

Lemma sign_odd : forall n, (-1) ^ odd n = (-1) ^ n.
Proof.
move=> n; rewrite -{2}[n]odd_double_half addnC double_mul2 exp_addn exp_muln.
by rewrite /= multm1x oppK exp1n mult1x.
Qed.

Lemma sign_addb : forall b1 b2, (-1) ^ (b1 (+) b2) = (-1) ^ b1 * (-1) ^ b2.
Proof. by do 2!case; rewrite //= ?multm1x ?mult1x ?oppK. Qed.
```

```
Lemma matrix_transpose_mul : forall m n p (A : M_(m, n)) (B : M_(n, p)),
   \^t (A *m B) =m \^t B *m \^t A.
Proof. split=> k i; apply: eq_isumR => j _; exact: multC. Qed.

Lemma matrix_multx1 : forall m n (A : M_(m, n)), A *m \1m =m A.
Proof.
move=> m n A; apply: matrix_transpose_inj.
by rewrite matrix_transpose_mul matrix_transpose_unit matrix_mult1x.
Qed.

Lemma matrix_distrR : forall m n p (A1 A2 : M_(m, n)) (B : M_(n, p)),
  (A1 +m A2) *m B =m A1 *m B +m A2 *m B.
Proof.
move=> m n p A1 A2 B; split=> i k /=; rewrite -isum_plus.
by apply: eq_isumR => j _; rewrite -distrR.
Qed.

Lemma matrix_distrL : forall m n p (A : M_(m, n)) (B1 B2 : M_(n, p)),
  A *m (B1 +m B2) =m A *m B1 +m A *m B2.
Proof.
move=> m n p A B1 B2; apply: matrix_transpose_inj.
rewrite matrix_transpose_plus !matrix_transpose_mul.
by rewrite -matrix_distrR -matrix_transpose_plus.
Qed.

Lemma matrix_multA : forall m n p q
   (A : M_(m, n)) (B : M_(n, p)) (C : M_(p, q)),
  A *m (B *m C) =m A *m B *m C.
Proof.
move=> m n p q A B C; split=> i l /=.
transitivity (\sum_(k) (\sum_(j) (A i j * B j k * C k l))).
  rewrite exchange_isum; apply: eq_isumR => j _; rewrite isum_distrL.
  by apply: eq_isumR => k _; rewrite multA.
by apply: eq_isumR => j _; rewrite isum_distrR.
Qed.

Lemma perm_matrixM : forall n (s t : S_(n)),
  perm_matrix (s * t)%G =m perm_matrix s *m perm_matrix t.
Proof.
move=> n; split=> i j /=; rewrite (isumD1 (s i)) // set11 mult1x -permM.
rewrite isum0 => [|j']; first by rewrite plusC plus0x.
by rewrite andbT; move/negbET->; rewrite mult0x.
Qed.

Lemma matrix_trace_plus : forall n (A B : M_(n)), \tr (A +m B) = \tr A + \tr B.
Proof. by move=> n A B; rewrite -isum_plus. Qed.
```



```
(* And now, finally, the title feature. *)

Lemma determinant_multilinear : forall n (A B C : M_(n)) i0 b c,
    row i0 A =m b *sm row i0 B +m c *sm row i0 C ->
    row' i0 B =m row' i0 A -> row' i0 C =m row' i0 A ->
  \det A = b * \det B + c * \det C.
Proof.
move=> n A B C i0 b c ABC.
move/matrix_eq_rem_row=> BA; move/matrix_eq_rem_row=> CA.
rewrite !isum_distrL -isum_plus; apply: eq_isumR => s _.
rewrite -!(multCA (_ ^ s)) -distrL; congr (_ * _).
rewrite !(@iprodD1 _ i0 (setA _)) // (matrix_eq_row ABC) distrR !multA.
by congr (_ * _ + _ * _); apply: eq_iprodR => i;
   rewrite andbT => ?; rewrite ?BA ?CA.
Qed.

Lemma alternate_determinant : forall n (A : M_(n)) i1 i2,
  i1 != i2 -> A i1 =1 A i2 -> \det A = 0.
Proof.
move=> n A i1 i2 Di12 A12; pose r := I_(n).
pose t := transp i1 i2; pose tr s := (t * s)%G.
have trK : involutive tr by move=> s; rewrite /tr mulgA transp2 mul1g.
have Etr: forall s, odd_perm (tr s) = even_perm s.
  by move=> s; rewrite odd_permM odd_transp Di12.
rewrite /(\det _) (isumID (@even_perm r)) /=; set S1 := \sum_(in _) _.
rewrite -{2}(minusxx S1); congr (_ + _); rewrite {}/S1 -isum_opp.
rewrite (reindex_isum tr); last by exists tr.
symmetry; apply: eq_isum => [s | s seven]; first by rewrite negbK Etr.
rewrite -multm1x multA Etr seven (negbET seven) multm1x; congr (_ * _).
rewrite (reindex_iprod t); last by exists (t : _ -> _) => i _; exact: transpK.
apply: eq_iprodR => i _; rewrite permM /t.
by case: transpP => // ->; rewrite A12.
Qed.

Lemma determinant_transpose : forall n (A : M_(n)), \det (\^t A) = \det A.
Proof.
move=> n A; pose r := I_(n); pose ip p : permType r := p^-1.
rewrite /(\det _) (reindex_isum ip) /=; last first.
  by exists ip => s _; rewrite /ip invgK.
apply: eq_isumR => s _; rewrite odd_permV /= (reindex_iprod s).
  by congr (_ * _); apply: eq_iprodR => i _; rewrite permK.
by exists (s^-1 : _ -> _) => i _; rewrite ?permK ?permKv.
Qed.

Lemma determinant_perm : forall n s, \det (@perm_matrix n s) = (-1) ^ s.
Proof.
move=> n s; rewrite /(\det _) (isumD1 s) //.
rewrite iprod1 => [|i _]; last by rewrite /= set11.
rewrite isum0 => [|t Dst]; first by rewrite plusC plus0x multC mult1x.
case: (pickP (fun i => s i != t i)) => [i ist | Est].
  by rewrite (iprodD1 i) // multCA /= (negbET ist) mult0x.
move: Dst; rewrite andbT; case/eqP.
```



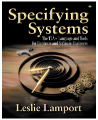
```
Lemma determinant1 : forall n, \det (unit_matrix n) = 1.
Proof.
move=> n; have:= @determinant_perm n 1%G; rewrite odd_perm1 => /= <-.
apply: determinant_extensional; symmetry; exact: perm_matrix1.
Qed.

Lemma determinant_scale : forall n x (A : M_(n)),
  \det (x *sm A) = x ^ n * \det A.
Proof.
move=> n x A; rewrite isum_distrL; apply: eq_isumR => s _.
by rewrite multCA iprod_mult iprod_id card_ordinal.
Qed.

Lemma determinantM : forall n (A B : M_(n)), \det (A *m B) = \det A * \det B.
Proof.
move=> n A B; rewrite isum_distrR.
pose AB (f : F_(n)) (s : S_(n)) i := A i (f i) * B (f i) (s i).
transitivity (\sum_(f) \sum_(s : S_(n)) (-1) ^ s * \prod_(i) AB f s i).
  rewrite exchange_isum; apply: eq_isumR => s _.
  by rewrite -isum_distrL distr_iprodA_isumA.
rewrite (isumID (fun f => uniq (fval f))) plusC isum0 ?plus0x => /= [|f Uf].
  rewrite (reindex_isum (fun s => val (pval s))); last first.
    have s0 : S_(n) := 1%G; pose uf (f : F_(n)) := uniq (fval f).
    pose pf f := if insub uf f is Some s then Perm s else s0.
    exists pf => /= f Uf; rewrite /pf (insubT uf Uf) //; exact: eq_fun_of_perm.
  apply: eq_isum => [s|s _]; rewrite ?(valP (pval s)) // isum_distrL.
  rewrite (reindex_isum (mulg s)); last first.
    by exists (mulg s^-1) => t; rewrite ?mulKgv ?mulKg.
  apply: eq_isumR => t _; rewrite iprod_mult multA multCA multA multCA multA.
  rewrite -sign_permM; congr (_ * _); rewrite (reindex_iprod s^-1); last first.
    by exists (s : _ -> _) => i _; rewrite ?permK ?permKv.
  by apply: eq_iprodR => i _; rewrite permM permKv ?set11 // -{3}[i](permKv s).
transitivity (\det (\matrix_(i, j) B (f i) j) * \prod_(i) A i (f i)).
  rewrite multC isum_distrL; apply: eq_isumR=> s _.
  by rewrite multCA iprod_mult.
suffices [i1 [i2 Ef12 Di12]]: exists i1, exists2 i2, f i1 = f i2 & i1 != i2.
  by rewrite (alternate_determinant Di12) ?mult0x => //= j; rewrite Ef12.
pose ninj i1 i2 := (f i1 == f i2) && (i1 != i2).
case: (pickP (fun i1 => ~~ set0b (ninj i1))) => [i1| injf].
  by case/set0Pn=> i2; case/andP; move/eqP; exists i1; exists i2.
case/(perm_uniqP f): Uf => i1 i2; move/eqP=> Dfi12; apply/eqP.
by apply/idPn=> Di12; case/set0Pn: (injf i1); exists i2; apply/andP.
Qed.

(* And now, the Laplace formula. *)

Definition cofactor n (A : M_(n)) (i j : I_(n)) :=
   (-1) ^ (val i + val j) * \det (row' i (col' j A)).

(* Same bug as determinant
Add Morphism cofactor with
```

Tools for formal proofs

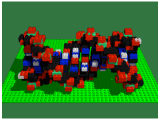
Damien Doligez, INRIA Rocq.
Leslie Lamport, MSR
Stephan Merz, INRIA Lorraine

Natural proofs

- first-order set theory + temporal logic
- specifications/verification of concurrent programs.
- tools for automatic theorem proving



TLA+



tools for proofs

Zenon

TLA$^+$ example: a clock

$$\begin{array}{rcl}
Clockini & \triangleq & hr \in \{0, ..., 23\} \ \wedge \ min \in \{0, ..., 59\} \\
 & & \\
Hr & \triangleq & hr' = hr \ \wedge \ min' = min + 1 \\
Min & \triangleq & hr' = hr + 1 \ \wedge \ min' = 0 \\
Clocknxt & \triangleq & min < 59 \ \wedge \ Hr \\
 & & \vee \ min = 59 \ \wedge \ Min \\
 & & \\
Clock & \triangleq & Clockini \wedge \Box[Clocknxt]_{<hr,min>}
\end{array}$$

- $Clockini$ is a predicate that describes the possible initial states.
- $Clocknxt$ is a predicate of two states: the current state, described by unprimed variables, and the next state, described by primed variables.
- $Clock$ is a temporal formula that specifies all the possible behaviours of our clock: they start in a state that satisfies $Clockini$ and every step they take must is $Clocknxt$ step.

Secure Distributed Computations and their Proofs

Cédric Fournet, MSR
Karthik Bhargavan, MSR
Ricardo Corin, INRIA-MSR
Pierre-Malo Deniélou, INRIA Rocq.
James Leifer, INRIA Rocq.
Jean-Jacques Lévy, INRIA Rocq.
Tamara Rezk, INRIA-MSR
Francesco Zappa Nardelli, INRIA Rocq.
G. Barthe, B. Grégoire, S. Zanella, INRIA Sophia

Distributed computations + Security

- programming with secured communications
- certified compiler from high level primitives to low level crypto-protocols
- formal proofs of probabilistic protocols

Secure Distributed Computations and their Proofs

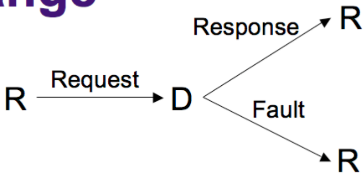
Cédric Fournet, MSR	James Leifer, INRIA Rocq.
Karthik Bhargavan, MSR	Jean-Jacques Lévy, INRIA Rocq.
Ricardo Corin, INRIA-MSR	Tamara Rezk, INRIA-MSR
Pierre-Malo Deniélou, INRIA Rocq.	Francesco Zappa Nardelli, INRIA Rocq.

G. Barthe, B. Grégoire, S. Zanella, INRIA Sophia

Distributed computations + Security

- programming with secured communications
- certified compiler from high level primitives to low level crypto-protocols
- formal proofs of probabilistic protocols

Simple Exchange



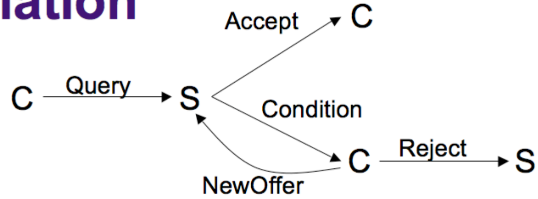
```
session S =
  role requester : int =
   !Request:string ;
   ?(Response:int + Fault:unit)

  role directory : string =
   ?Request:string;
   !(Response:int + Fault:unit)
```

Session declaration

```
let lookup name =
 S.requester ["client";"server"]
     (Request
           (name,
             {hResponse = (fun _ q → q) ;
              hFault = (fun _ x → failwith "Failed")
             }))
in lookup "Ricardo"
```

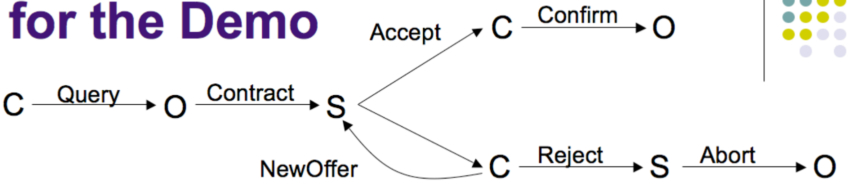
User code

Negotiation



```
session S2 =
   role customer : string =
    !Query:int;
      mu start.?(Accept:unit +
                 Condition:unit;!(NewOffer:int;start + Reject:unit))

     role store : string=
    ?Query:int;
      mu start.!(Accept:unit +
                 Condition:unit;?(NewOffer:int;start + Reject:unit))
```

Three-party session for the Demo



C Query → O Contract → S

S Accept → C Confirm → O

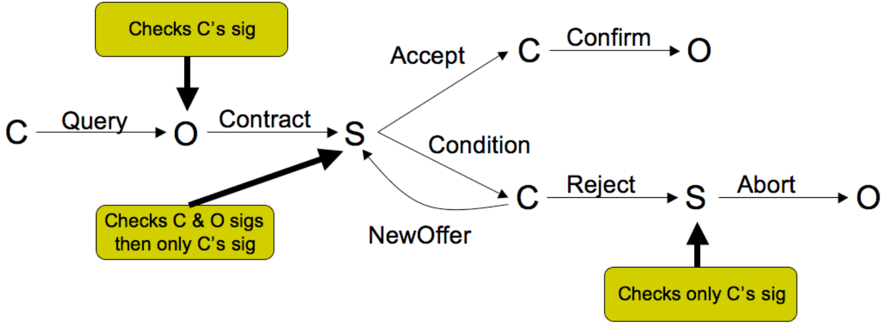
S → C Reject → S Abort → O

C NewOffer → S

```
session S3 =
   role customer :string =
    !Query:int;
     mu start.?(Accept:unit;!Confirm:unit +
                      Condition:unit; !(Newoffer:int;start + Reject:unit;))

   role store :string=
    ?Contract:int;
     mu start.!(Accept:unit +
                      Condition:unit; ?(Newoffer:int;start + Reject:unit;!Abort:unit))

   role officer :string=
    ?Query:int;!Contract:int;?(Confirm:unit + Abort:unit)
```

Visibility

- Minimal sequence of signatures that guarantee session compliance.
- Example:



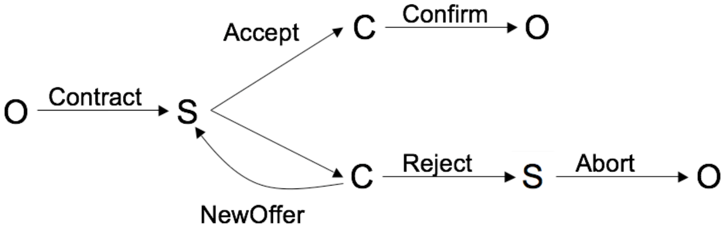
Checks C's sig

Checks C & O sigs then only C's sig

Checks only C's sig

No Fork

- Some forks in protocols represent a security threat.



Accept

C

Confirm

O

O

Contract

S

C

Reject

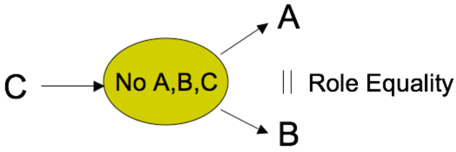
S

Abort

O

NewOffer

- Property



A

C

No A,B,C

|| Role Equality

B

Track B

Computational Sciences

CS research in use for other Sciences/Scientists

Current proposals

- Information interaction
  - dynamic encyclopedia of mathematics
    (Bruno Salvy, Alin Bostan, Frédéric Chyzak)
  - management of scientific workflows
    (Wendy Mackay, J.-D. Fekete, Mary Czerwinski, George Robertson)

- Scientific data visualisation
  - image and video analysis for environmental sciences
    (Patrick Perez, Andrew Blake)
  - geometric methods for data analysis
    (J.-D. Boissonnat, F. Chazal, F. Cazals, D. Cohen-Steiner)

Future

Future

- install Track B in 2007
- 30 researchers
- tight links with french academia (phD, post-doc)
- develop useful research for scientific community
- provide public tools (BSD licence)
- become a new and attractive pole in CS research