Designing Reliable, High-Performance Networks

. . . with the Nuprl Proof Development System

Christoph Kreitz

Department of Computer Science, Cornell University

Ithaca, NY 14853

The Nuprl Project

- **Computational Formal Logics**

  = Extension of Martin-Löf's constructive Type Theory

  + Class theory + meta-reasoning + reflection + ...

  ⋮

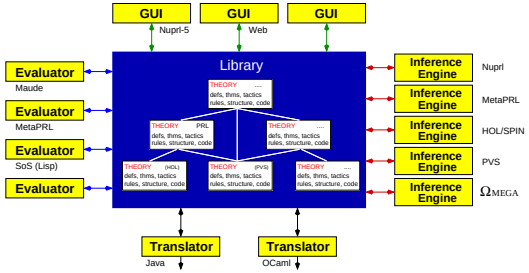
- **Proof & Program Development Systems**

  – Nuprl Logical Programming Environment

  – Proof search techniques + inference engines

  – Natural language generation

  ⋮



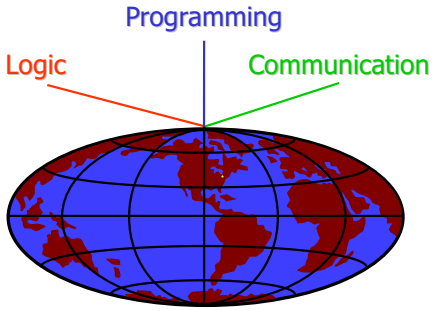
- **Application to Networked Systems**

  – Verification of communication protocols

  – Optimization of Ensemble protocol stacks

  – Formal design of adaptive systems

  ⋮



Secure software infrastructure

Features of Nuprl's Type Theory

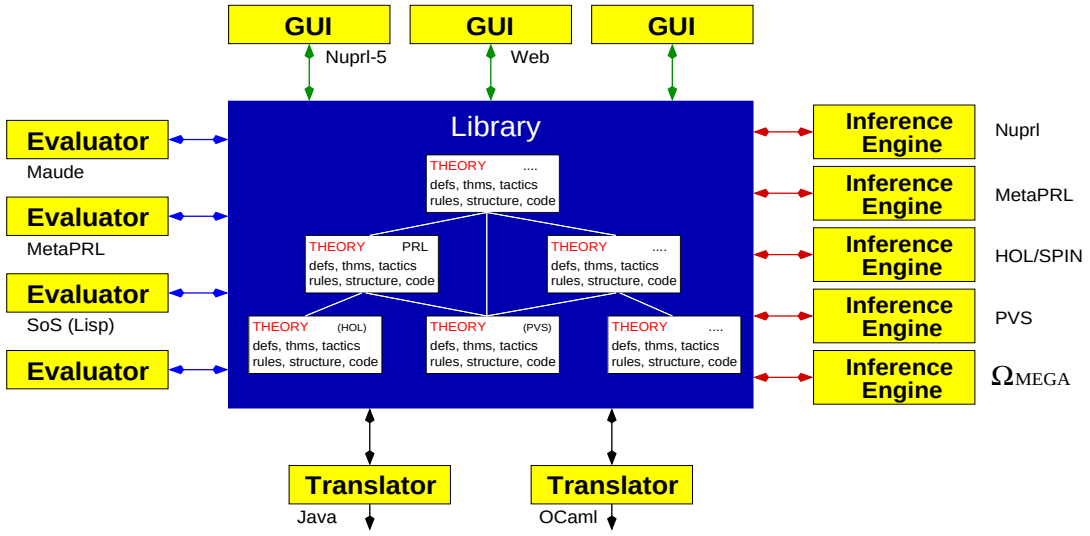
- **Open-ended, expressive type system**
  - Function, product, disjoint union, Π- & Σ-types, atoms ⇝ programming
  - Integers, lists, inductive types ⇝ inductive definition
  - Propositions as types, equality type, void, top, universes ⇝ logic
  - Subsets, subtyping, quotient types ⇝ mathematics
  - (Dependent) intersection, union, records ⇝ modules, program composition

  New types can be added as needed
- **Uniform internal notation**
  - No syntactical distinction between types, members, propositions ...
  - Independent term display allows "free syntax" ⇝ display forms
- **Expressions independent of types**
  - No restriction on expressions that can be defined ⇝ Y combinator
  - Expressions *in proofs* must be typeable ⇝ "total" functions
- **Refinement calculus**
  - Top-down sequent calculus ⇝ interactive proof development
  - Computation rules and extract terms ⇝ program development
- **User-defined extensions possible**
  - Language extensions (abstractions) + user-defined inference rules (tactics)

Features of Nuprl's Proof System

- Interactive proof editor ⇝ readable proofs
- Flexible definition mechanism ⇝ user-defined terms
- Customizable term display ⇝ flexible notation
- Structure editor for terms ⇝ no ambiguities
- Tactics & decision procedures ⇝ proof automation
- Program evaluation and extraction ⇝ program synthesis
- Library mechanism ⇝ large user-theories
- Formal documentation mechanism ⇝ LaTeX, HTML

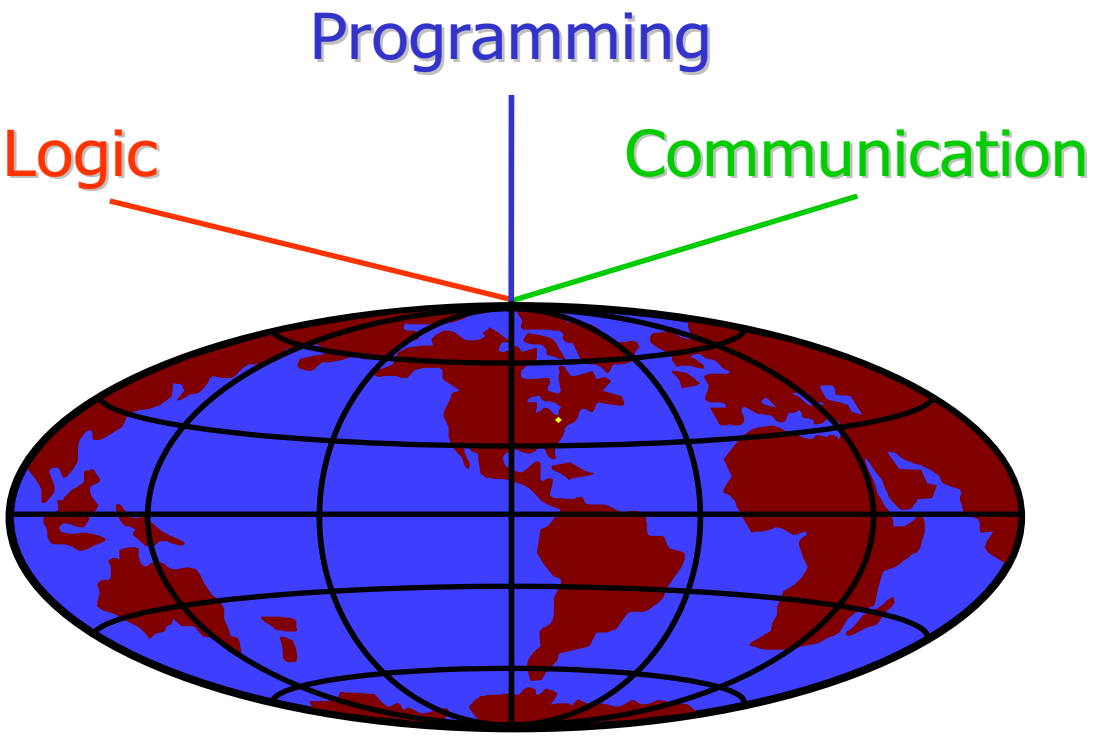
Open Architecture supports Cooperation



- **Collection of cooperating processes** ⇝ interoperability
  - Enables asynchronous, distributed & cooperative theorem proving
- **Centered around a common knowledge base**
  - Persistent data base, version control, dependency tracking ⇝ accountability
  - System structure designed within the library ⇝ customizability
- **Connected to external systems**
  - MetaPRL (fast rewriting, multiple logics) (Hickey & Nogin, 1999)
  - JProver (matrix-based intuitionistic theorem prover) (IJCAR 2001)
  - Multiple user interfaces ⇝ collaborative proving

⋮

Application: Reliable, High-Performance Networks



Secure software infrastructure

Link Ensemble communication system to Nuprl LPE

- Verify protocol components and system configurations (TACAS 1999)
- Optimize performance of configured systems (TACAS 1999, SOSP 1999)
- Formalize semantics of OCaml (CADE 1998, . . .)
- Formally design and verify new protocols (DISCEX 2001, TPHOLS 2001)

The Ensemble Group Communication Toolkit

Modular group communication system

– Developed by Cornell's System Group (Ken Birman)

– Used commercially (BBN, JPL, Segasoft, Alier, Nortel Networks)

Architecture: stack of micro-protocols

– Select from more than 60 micro-protocols for specific tasks

– Modules can be stacked arbitrarily

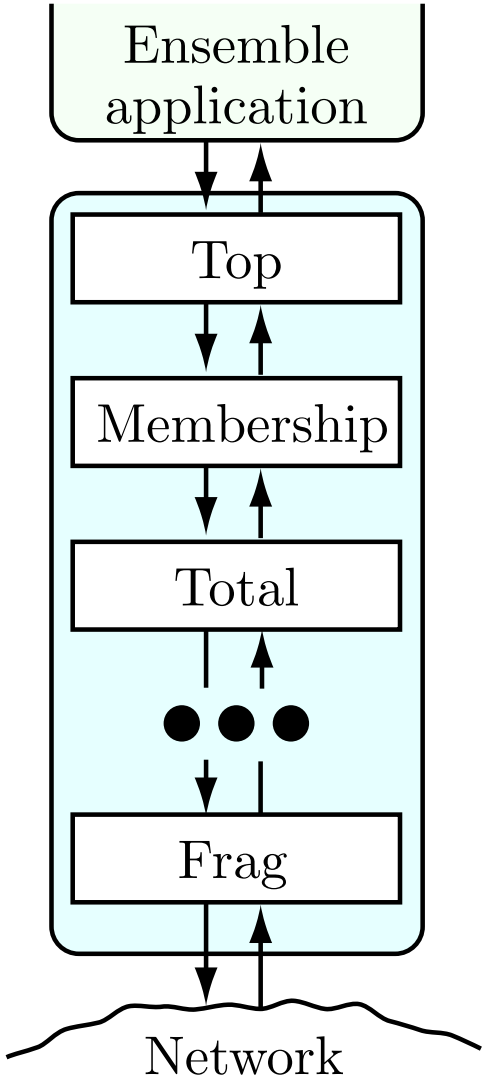
– Modeled as state/event machines

Implementation in Objective Caml (INRIA)

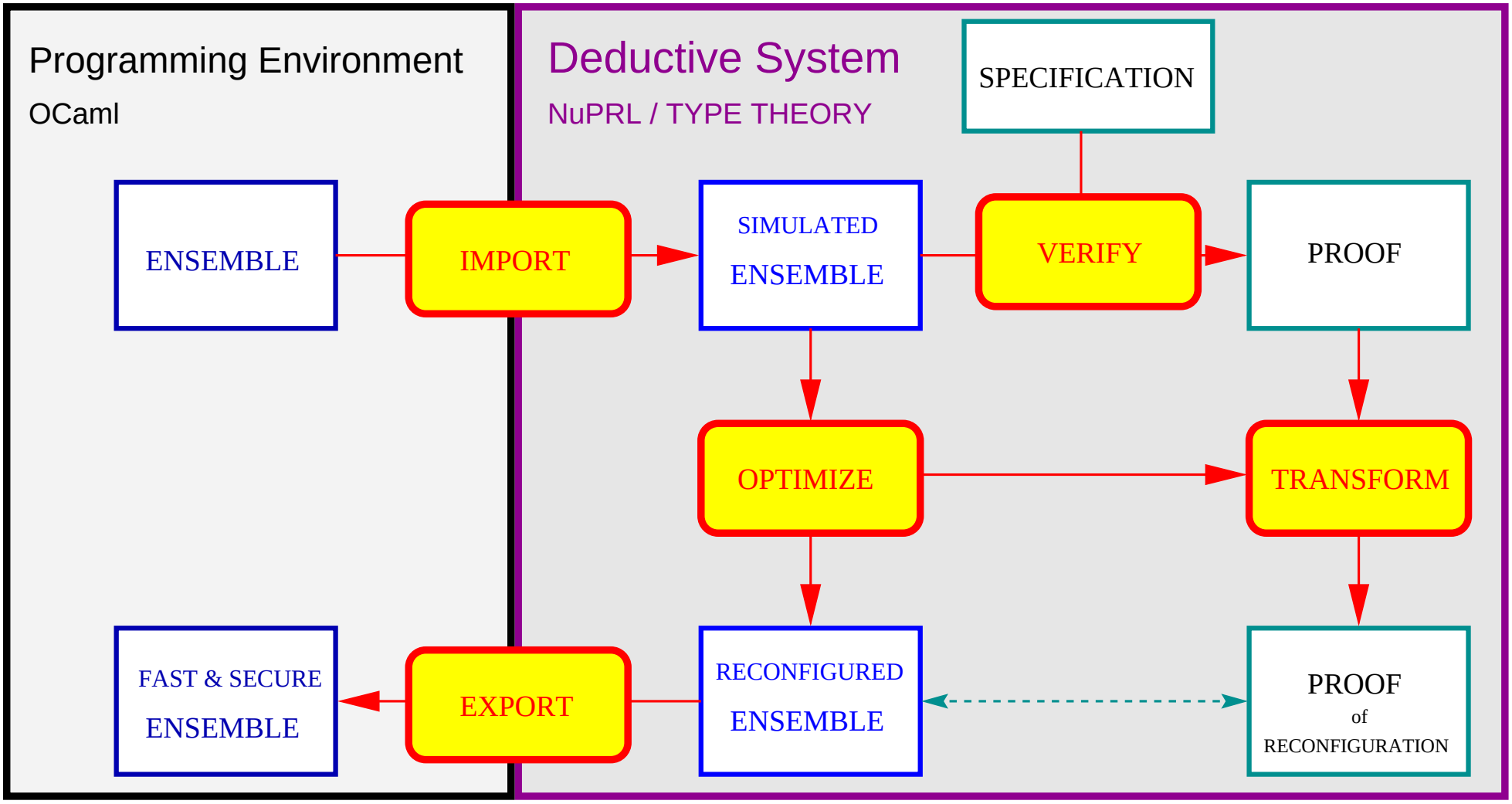
– Easy maintenance (small code, good data structures)

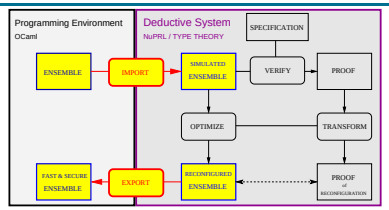
– Mathematical semantics, strict data type concepts

– Efficient compilers and type checkers



Linking Ensemble and the Nuprl LPE





Embedding Ensemble's code into Nuprl

- **Type-theoretical semantics of OCaml**
  - Functional core, pattern matching, exceptions, references, modules, ...
  - Evaluation may update store, uses environment, returns value or exception
  - Nuprl's Type theory has only β-reduction

  $\rightsquigarrow$ Represent as functions in $\texttt{STORE} \longrightarrow \texttt{ENV} \longrightarrow (\texttt{EXCEPTION} + T) \times \texttt{STORE}$

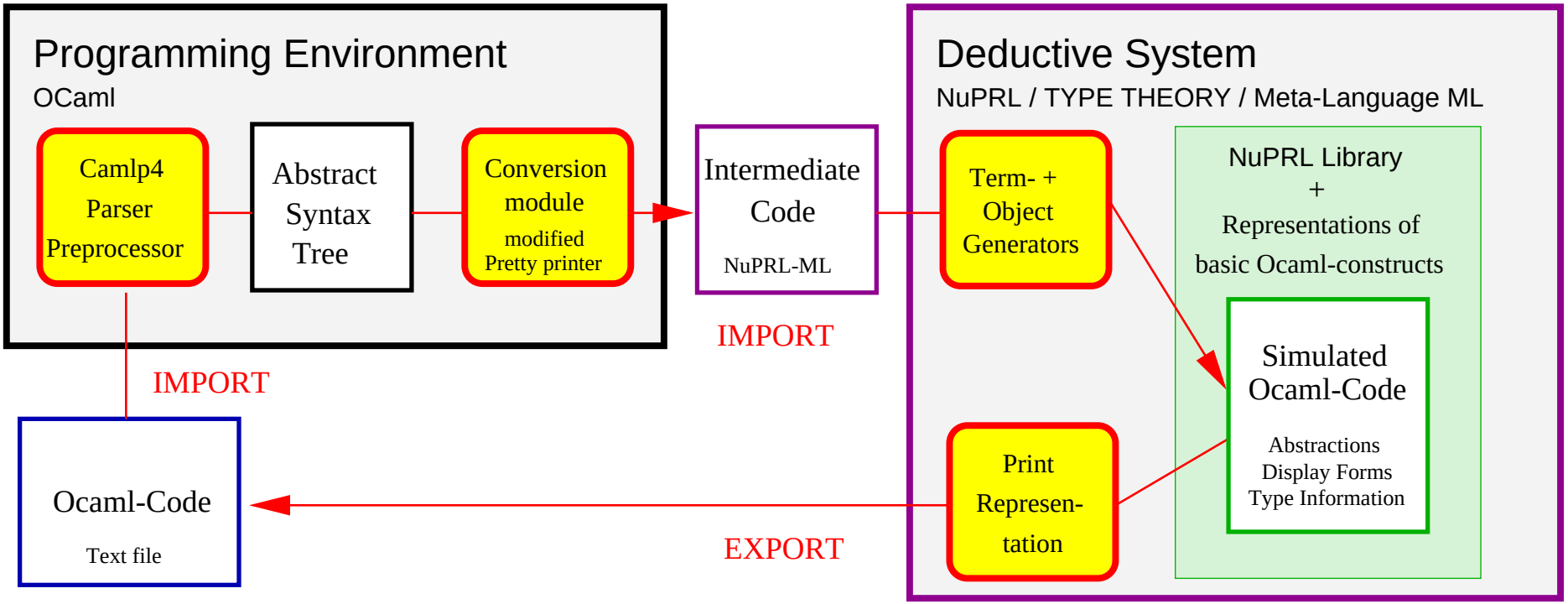
- **Implementation through Nuprl definitions**
  - Representation of semantics (abstractions) + OCaml syntax (display forms)
  - Many predefined data types, expressions, and patterns must be formalized

- **Programming logic for OCaml**
  - (Derived) rules for formal reasoning about OCaml code

$\Downarrow$

Formal reasoning on level of programming language

Importing and Exporting System Code

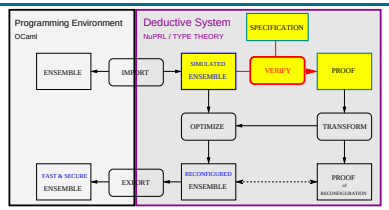


Import: – Parse with Camlp4 parser-preprocessor
– Convert abstract syntax tree into term- & object generators
– Generators perform second pass and create NuPRL library objects

Export: – Print-representation is genuine OCaml-code

⇓

Actual Ensemble code available for formal reasoning



Specifications and Correctness

- **System properties**

  e.g. FIFO: *"Messages are received in the same order in which they were sent"*

  – $\forall i,j,k,l{<}|tr|.\,(i{<}j \wedge tr[i]{\downarrow}tr[k] \wedge tr[j]{\downarrow}tr[l]) \Rightarrow k{<}l$

- **Abstract (global) behavioral specification**

  "Messages may be appended to global event queue and removed from its beginning"

  – Represented as formal nondeterministic I/O Automaton

- **Concrete (local) behavioral specification**

  "Messages whose sequence number is too big will be buffered"

  – Represented as deterministic I/O Automaton

- **Implementation**

  – Ensemble module `Pt2pt.ml`: 250 lines of OCaml code

All formalisms are represented in Nuprl's type theory

IOA Specifications of a FIFO network

Abstract behavioral specification

Specification	FifoNetwork()	
Variables	in-transit: queue of $\langle Address, Message \rangle$	
Actions	$Send(dst : Address; msg : Message)$	
	condition: true	{in-transit.append($\langle dst, msg \rangle$)}
	$Deliver(dst : Address; msg : Message)$	
	condition: in-transit.head() = $\langle dst, msg \rangle$	{in-transit.dequeue()}

Concrete behavioral specification

Specification	FifoProtocol($\boldsymbol{p}$: $Address$)
Variables	send-window, recv-window, ...
Actions	$Above.Send(dst : Address; msg : Message)$
	{ ...list of individual sub-actions ... }
	$Below.Send(dst : Address; \langle hdr, msg \rangle : \langle Header, Message \rangle)$
	$Below.Deliver(dst : Address; \langle hdr, msg \rangle : \langle Header, Message \rangle)$
	$Above.Deliver(dst : Address; msg : Message)$
	$Timer()$

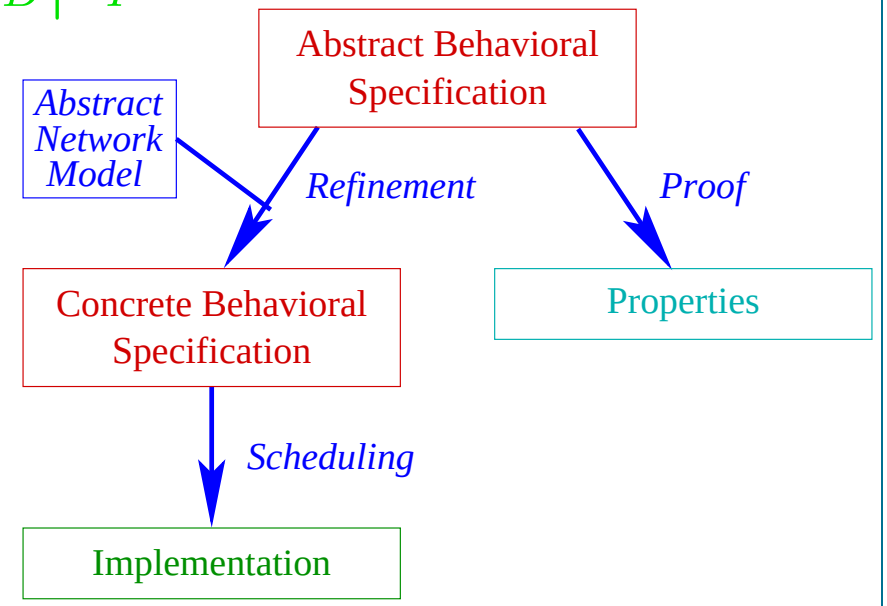
I/O-automata represented as dependent product types

Ensemble code for a FIFO protocol

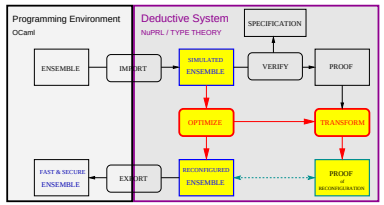
```
let name = Trace.source_file "PT2PT"
type header = NoHdr | Data of seqno | Ack of seqno  | Nak of seqno * seqno
type 'abv state = {sweep: Time.t; sends: 'abv Iq.t Arraye.t ; recvs ... }
let  init _ (ls,vs) = {sweep = Param.time vs.params "pt2pt_sweep" ; .........}
let  hdlrs s (ls,vs) {up_out=up;upnm_out=upnm; dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
 = let up_hdlr ev abv hdr = ...
   and uplm_hdlr ev hdr   = ...
   and upnm_hdlr ev       = ...
   and dn_hdlr ev abv     =
    match getType ev with
    | ESend ->
       let dest = getPeer ev in
       if dest = ls.rank then (eprintf "PT2PT:%s\nPT2PT:%s\n"
                               (Event.to_string ev) (View.string_of_full (ls,vs));
                               failwith "send to myself" ) ;
       let sends = Arraye.get s.sends dest in
       let seqno = Iq.hi sends in
       let iov = getIov ev in
       Arraye.set s.sends dest (Iq.add sends iov abv) ;
       dn ev abv (Data seqno)
    | _ -> dn ev abv NoHdr
   and dnnm_hdlr ev       = dnnm
 in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}
let l args vs = Layer.hdr init hdlrs args vs
Layer.install name l
```

Verification Methodology

- **Verify IOA-specifications of micro-protocols**
  - Concrete specification $\leftrightarrow$ abstract specification $\longrightarrow$ system properties
  - Easy for benign networks $\rightsquigarrow$ **subtle bug discovered**
- **Verify protocol stacks by IOA-composition**
  - IOA-composition represented as automata intersection
  - Preserves safety properties: $A \models P \Rightarrow A \cap B \models P$
- **Weave aspects** (ongoing)
  - Transformations add tolerance against network failures or security attacks
- **Verify code** (ongoing)
  - Micro-protocols $\leftrightarrow$ IOA-specifications
  - Layer composition $\leftrightarrow$ IOA-composition

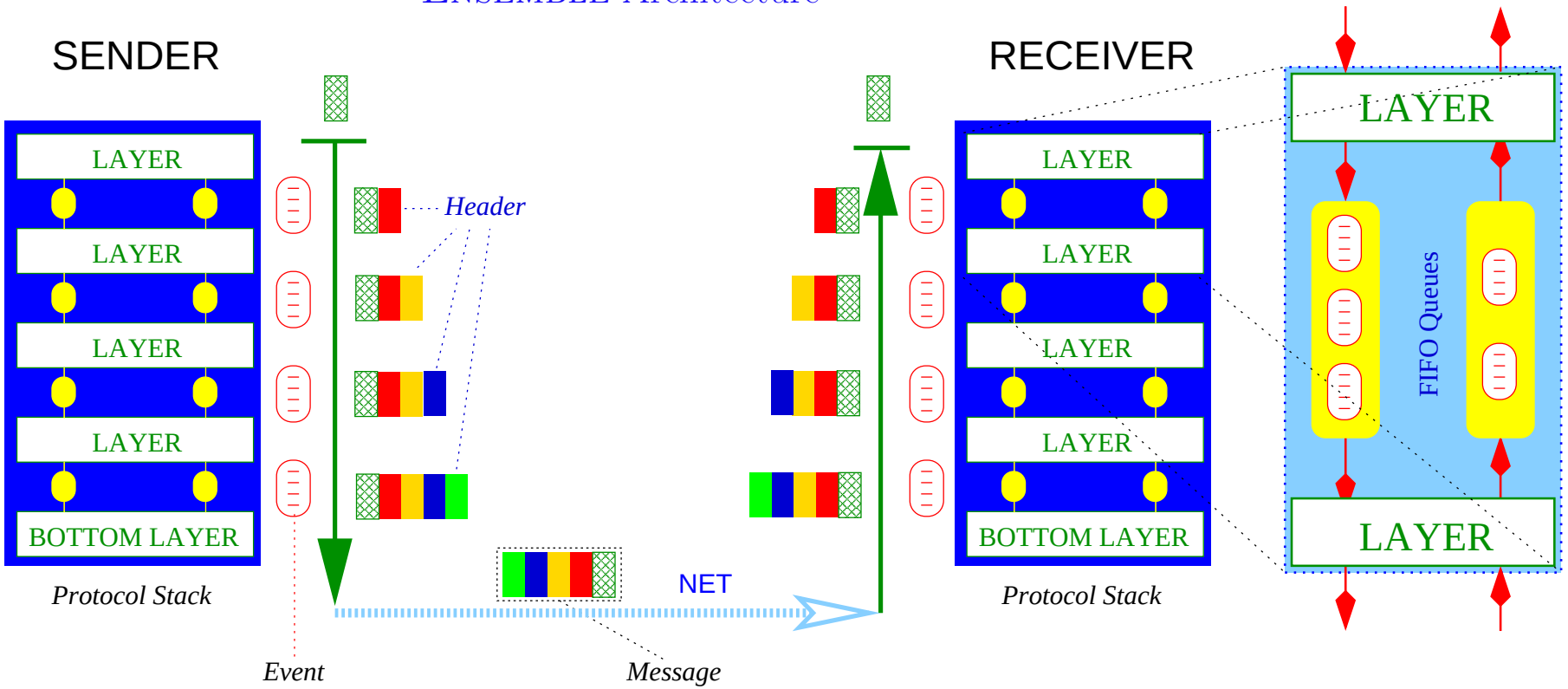


Verification process can be reversed into network synthesis



Optimization of Protocol Stacks

Ensemble Architecture



Performance loss: redundancies, internal communication, large message headers

Optimizations: bypass-code for common execution sequences, header compression

Need formal methods to do this correctly

Example Protocol Stack Bottom::Mnak::Pt2pt

Trace downgoing Send events and upgoing Cast events

Bottom (200 lines)

```
let name = Trace.source_file "BOTTOM"

type header = NoHdr | ... | ...

type state = {mutable all_alive : bool ; ... }

let  init _ (ls,vs) = {.........}

let  hdlrs s (ls,vs)
     {up_out=up;upnm_out=upnm;
      dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm}
 = ...
 let up_hdlr ev abv hdr =
  match getType ev, hdr with
  | (ECast|ESend), NoHdr ->
    if s.all_alive  or not (s_bottom.failed.(getPeer ev))
       then up ev abv
       else free name ev
  | :
 and uplm_hdlr ev hdr    = ...
 and upnm_hdlr ev        = ...
 and dn_hdlr ev abv      =
  if s.enabled then
    match getType ev with
     | ECast          -> dn ev abv NoHdr
     | ESend          -> dn ev abv NoHdr
     | ECastUnrel     -> dn (set name ev[Type ECast]) abv Unrel
     | ESendUnrel     -> dn (set name ev[Type ESend]) abv Unrel
     | EMergeRequest -> dn ev abv MergeRequest
     | EMergeGranted -> dn ev abv MergeGranted
     | EMergeDenied  -> dn ev abv MergeDenied
     | _ -> failwith "bad down event[1]"
   else (free name ev)
 and dnnm_hdlr ev        = ...
 in {up_in=up_hdlr;uplm_in=uplm_hdlr;upnm_in=upnm_hdlr;
     dn_in=dn_hdlr;dnnm_in=dnnm_hdlr}

let l args vs = Layer.hdr init hdlrs args vs

Layer.install name (Layer.init l)
```

Mnak (350 lines)

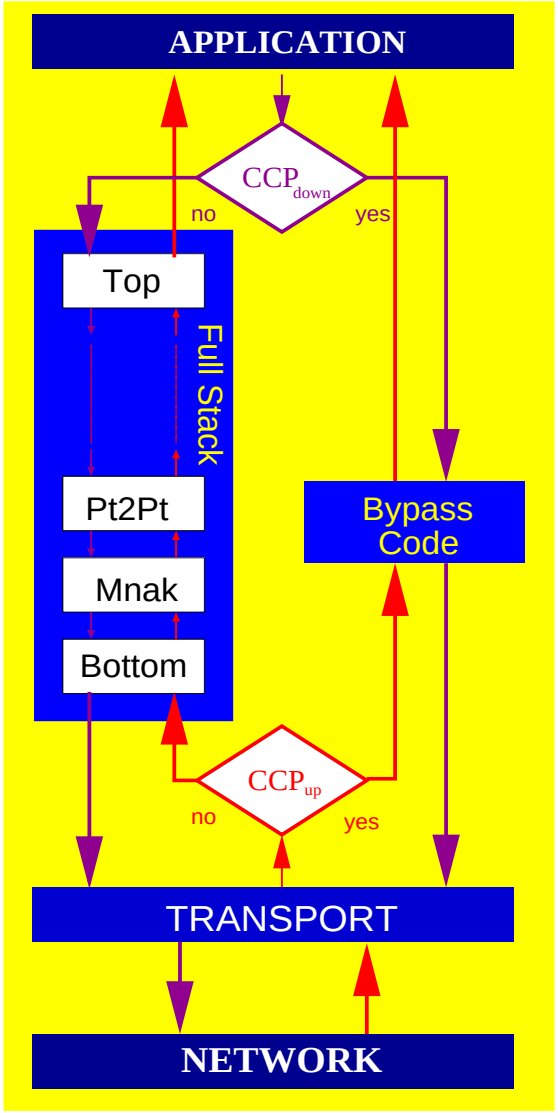
```
let init ack_rate (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
   match getType ev with
    | ECast ->
       let iov = getIov ev in
       let buf = Arraye.get s.buf ls.rank in
       let seqno = Iq.hi buf in
       assert (Iq.opt_insert_check buf seqno) ;
       Arraye.set s.buf ls.rank
             (Iq.opt_insert_doread buf seqno iov abv) ;
       s.acct_size <- s.acct_size + getIovLen ev ;
       dn ev abv (Data seqno)
    | _ -> dn ev abv NoHdr
  :
```

Pt2pt (250 lines)

```
let init _ (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv      =
  match getType ev with
   | ESend ->
      let dest = getPeer ev in
      if dest = ls.rank then (
        eprintf "PT2PT:%s\nPT2PT:%s\n"
         (Event.to_string ev) (View.string_of_full (ls,vs));
        failwith "send to myself" ;
       ) ;
      let sends = Arraye.get s.sends dest in
      let seqno = Iq.hi sends in
      let iov = getIov ev in
      Arraye.set s.sends dest (Iq.add sends iov abv) ;
      dn ev abv (Data seqno)
   | _ -> dn ev abv NoHdr
   :
```

Formal Optimization in the Nuprl LPE

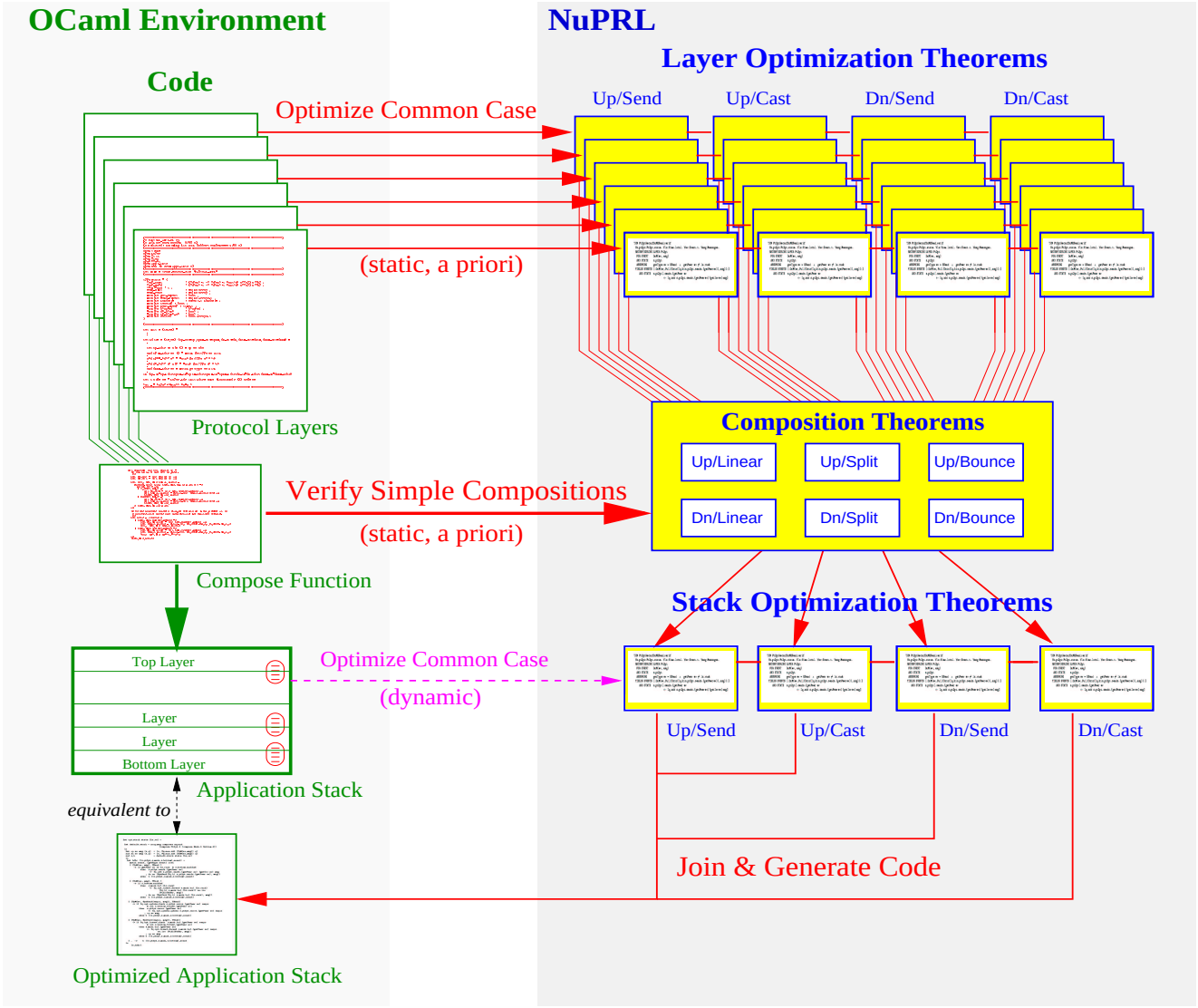
- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as Common Case Predicate
- **Analyze path of events through stack**
- **Isolate code for fast-path**
- **Integrate code for compressing headers of common messages**
- **Generate bypass-code**
  - Insert CCP as runtime switch



Methodology: compose formal optimization theorems

Fast, error-free, independent of programming language, **speedup factor 3-10**

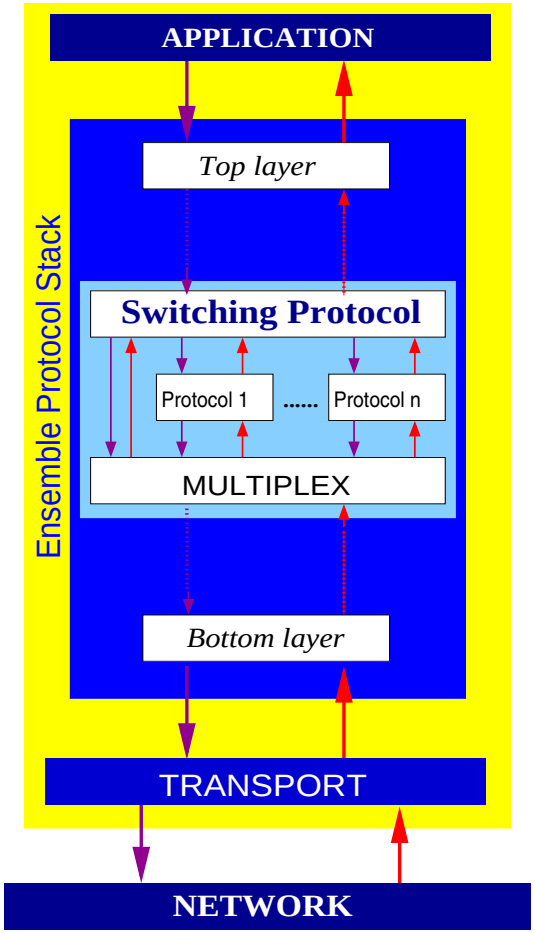
Methodology: Compose Optimization Theorems



1. Use known optimizations of micro-protocols — A priori: Ensemble + Nuprl experts
2. Compose into optimizations of protocol stacks — automatic: application designer
3. Integrate message header compression — automatic: ⋮
4. Generate code from optimization theorems and reconfigure system — automatic: ⋮

Formal Design of Adaptive Systems

- **Make systems adapt safely to run-time dynamics**
  - On-line upgrading, security, performance
  - Difficult to design correctly *(distributed migration?)*



- **Generic switching protocol**
  - Construct hybrid protocols from simpler ones
  - Normal mode: interact with one protocol
  - Switching mode: deliver old messages, buffer new ones

- **Correctness issues**
  - *What kind of protocols are switchable at all?*
    - Reliability? Integrity? Confidentiality? Total Order? ...
  - *What code invariant guarantees that switchable properties are preserved?*

LPE verification answers both questions

A Formal Model of Communication

- **Communication property P**
  - Predicate on traces, i.e. lists of Send(p,m) and Deliver(p,m) events

    e.g. $\texttt{Reliable}(tr) \equiv \forall \texttt{p,q:PID}.\forall \texttt{m:Msg}.\ \texttt{Send(p,m)} \in tr \Rightarrow \texttt{Deliver(q,m)} \in tr$

- **Characterize switchable properties by meta-properties**
  - Predicates on communication properties
  - Expressed by relation R between traces tr_u, tr_l above/below a protocol

    R preserves $P \equiv \forall \texttt{tr}_u,\texttt{tr}_l\texttt{:Trace}.\ (P(\texttt{tr}_l) \wedge \texttt{tr}_u\, R\, \texttt{tr}_l) \Rightarrow P(\texttt{tr}_u)$

  Examples of meta-properties:

tr_u R_safety tr_l $\equiv$ $tr_u \sqsubseteq tr_l$

tr_u R_asynchrony tr_l $\equiv$ tr_u $\texttt{swap-adjacent}_{[\texttt{loc(e)}\neq\texttt{loc(e')}]}$ tr_l

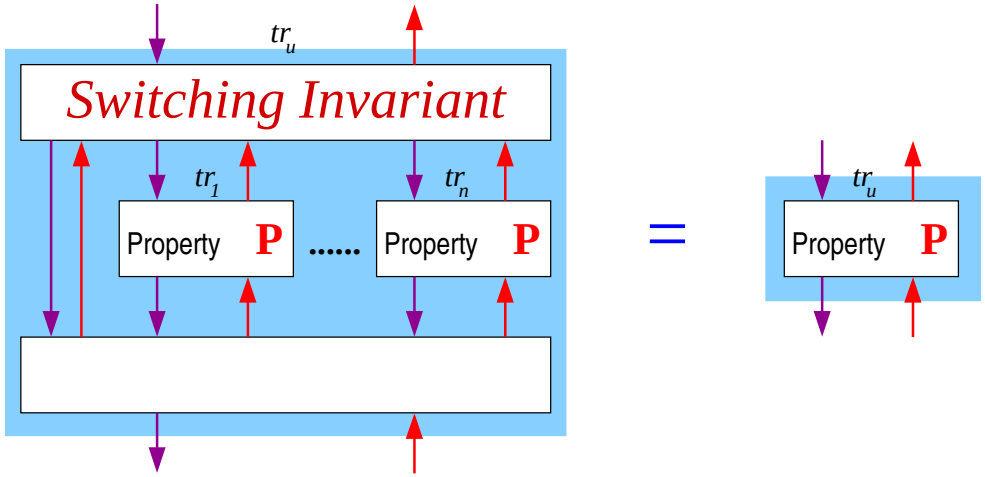
tr_u R_delayable tr_l $\equiv$ tr_u $\texttt{swap-adjacent}_{[\texttt{msg(e)}\neq\texttt{msg(e')} \wedge \texttt{is-send(e)}\neq\texttt{is-send(e')}]}$ tr_l

tr_u R_send-enabled tr_l $\equiv$ $\exists \texttt{e:Events}.\ \texttt{is-send(e)} \wedge tr_u = tr_l\texttt{@[e]}$

tr_u R_memoryless tr_l $\equiv$ $\exists \texttt{e:Events}.\ tr_u = [\,\texttt{e}_1 \in tr_l \mid \texttt{msg(e)} \neq \texttt{msg(e}_1\texttt{)}\,]$

R_composable(tr_u, tr_1, tr_2) $\equiv$ $tr_u = tr_1\texttt{@}tr_2 \wedge \forall \texttt{e}_1 \in tr_1.\forall \texttt{e}_2 \in tr_2.\ \texttt{msg(e}_1\texttt{)} \neq \texttt{msg(e}_2\texttt{)}$

Verifying the Correctness of Switching



switchable(P)

$\equiv$ P refines Causality
$\wedge$ P refines No-replay
$\wedge$ R_safety preserves P
$\wedge$ R_async preserves P
$\wedge$ R_delayable preserves P
$\wedge$ R_send-enabled preserves P
$\wedge$ R_memoryless preserves P
$\wedge$ R_composable preserves_3 P

- **Characterize switch invariant between tr_u and $tr_1,..,tr_n$**
  - tr_u results from joint trace by swapping events with different origin
  - Messages sent by different protocols must be delivered in same order
- **Prove that switchable properties will be preserved**

$\vdash \forall P{:}\text{TraceProperty}.\ \forall \text{tr}_u, \text{tr}_1, \ldots \text{tr}_n{:}\text{Trace}.$
$\quad \text{switchable}(P) \wedge \text{switch\_invariant}(\text{tr}_u; \text{tr}_1, .., \text{tr}_n)$
$\Rightarrow\ (\ \forall i \leq n.\ P(\text{tr}_i) \ \Rightarrow\ P(\text{tr}_u)\ )$

Abstract verification affects implementation and use of switch

Lessons learned

- **Results**
  - Type theory expressive enough to formalize today's software systems
  - Nuprl LPE capable of supporting real design at reasonable pace
  - Formal verification reveals errors even in well-explored designs
  - Formal optimization can significantly improve practical performance
  - Formal design reveals hidden assumptions and limitations for use of protocols
- **Ingredients for success . . .**
  - Collaboration between systems and formal reasoning groups
  - Implementation language with precise semantics
  - Employing formal methods at every design stage
  - Formal models of: communication, I/O-automata, programming language
  - Knowledge-based approach: large library of algorithmic knowledge
  - Great colleagues! Stuart Allen, Mark Bickford, Ken Birman, Robert Constable, Richard Eaton, Xiaming Liu, Lori Lorigo, Robbert van Renesse

Future Challenges

- **Better reasoning tools**
  - Build interactive library of formal algorithmic knowledge (ONR project)
  - Deploy new reflection mechanism
  - Connect more external systems
  - **Improve cooperation between research groups**
- **Learn more from applications**
  - Build support for aspect-oriented programming
  - Support reasoning about real-time & embedded systems
    - reason about probabilistic protocols
  - Support programming languages with less clean semantics
  - Invert reasoning direction from verification to synthesis