A Proof Environment for the Development of Group Communication Systems

Christoph Kreitz Mark Hayden Jason Hickey

Department of Computer Science, Cornell University, Ithaca, NY 14853

Secure Group Communication Systems

Safety-critical Network Applications

Many Properties – Many Protocols

Modular Communication Toolkits — Protocol Layer Stacking

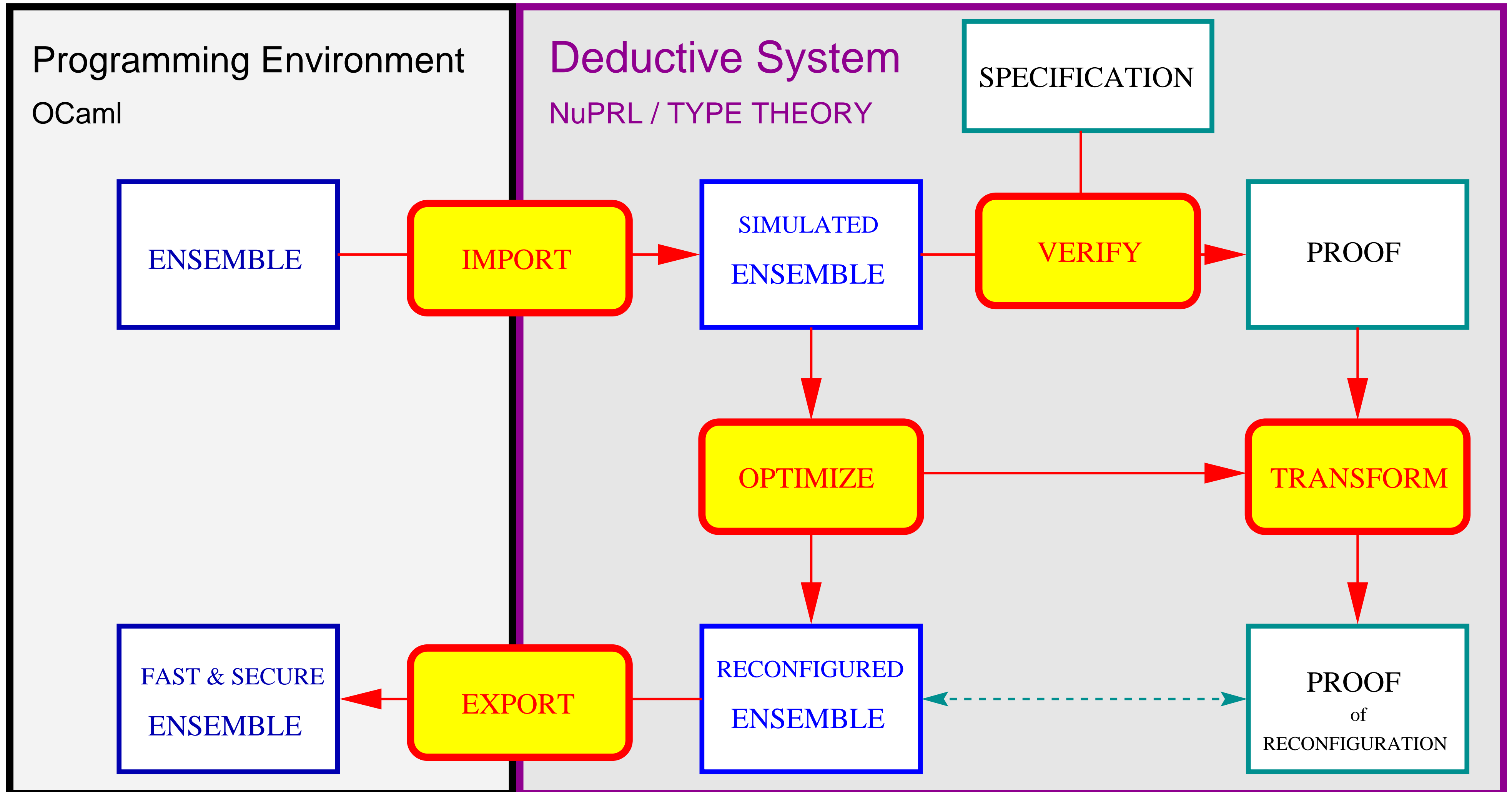
Building Networks by Stacking Protocol Layers

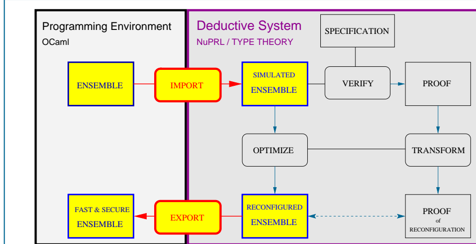
+ Flexible **+ Easy to design & maintain** **+ Construction by Composition** – Layers can be stacked arbitrarily – Many configurations possible – Properties add up **+ Verification possible** – Small modules	**? Performance Cost** – Internal communication overhead – Abstraction, redundant/extraneous code – Increased net load (message headers) **? Secure Implementation** – Distributed design difficult / error prone – No a priori verification **? Formalization Barrier** – Reasoning about real code? – Flexible tools?

Ensemble **Group Communication Toolkit** (redesign of Isis)
- Well-defined execution model, functional partitioning, no global operations
- Reference implementation in Ocaml

+ NuPRL **Proof/Program Development System**
  - Formalism: Type Theory ≡ Logic + Programming language + Data Types
  - Deductive system: Proof editor + Tactics + Definitions + Metalanguage...

Logical Programming Environment





Importing and Exporting System Code

- **Type-theoretical Semantics of OCAML**
  - Conservative extension of NUPRL's type theory

  ↦ Supports formal reasoning about OCAML-code

  ↦ Supports symbolic evaluation of OCAML-code

- **Implementation by Abstractions & Display Forms**
  - NUPRL-terms represent formal semantics + original program text
  - Hyperlinks: function call → function definitions

  ↦ Supports interaction when reasoning about OCAML-code

- **Transformation Algorithms:** OCAML ↔ NUPRL-representation

  ↦ Actual ENSEMBLE implementation available for formal reasoning

  ↦ Formally optimized code available in application system

Embedding Ocaml into NuPRL's type theory

Ocaml subset required for implementing finite state-event systems

- **Functional core: similar to type theory**
  - Straightforward representations of predefined types and operations
  - Combination of several terms for variable-sized expressions

  $\mapsto$ Ocaml language constructs $\hat{=}$ meta-level term generators

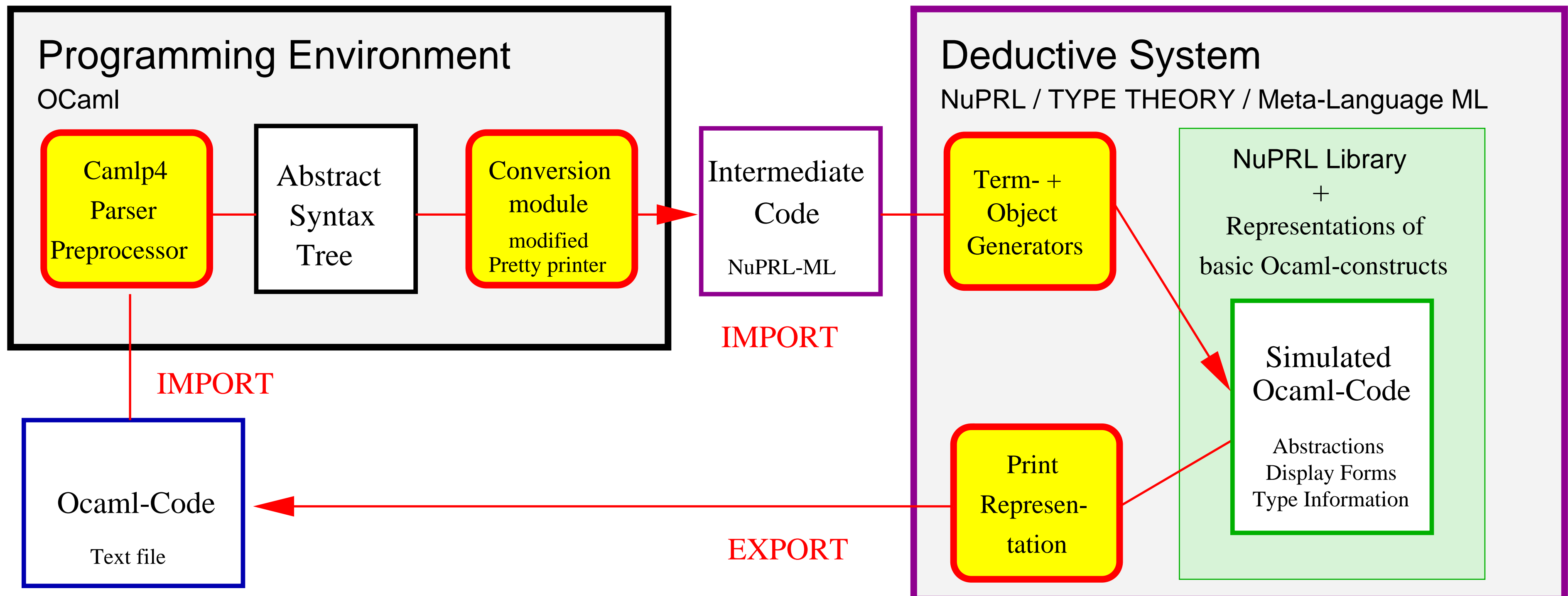
- **Patterns $\hat{=}$ matching functions**
  - Analyze an input expression, substitute variables in target

- **Imperative features $\hat{=}$ update functions**
  - Copy semantics (reasoning only): update layer state and event queue

- **Declarations $\hat{=}$ object generators**
  - Programmed functions and types must be stored and referenced

- **Modules $\hat{=}$ name space management**
  - Support code structuring and re-use of names

Transformation Algorithms

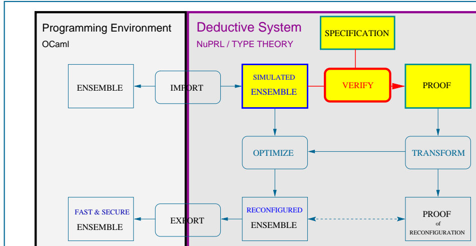


Import: – Parse with original Camlp4 parser-preprocessor

– Convert syntax tree into term- & object generators

– Generators perform second pass and create NuPRL library objects

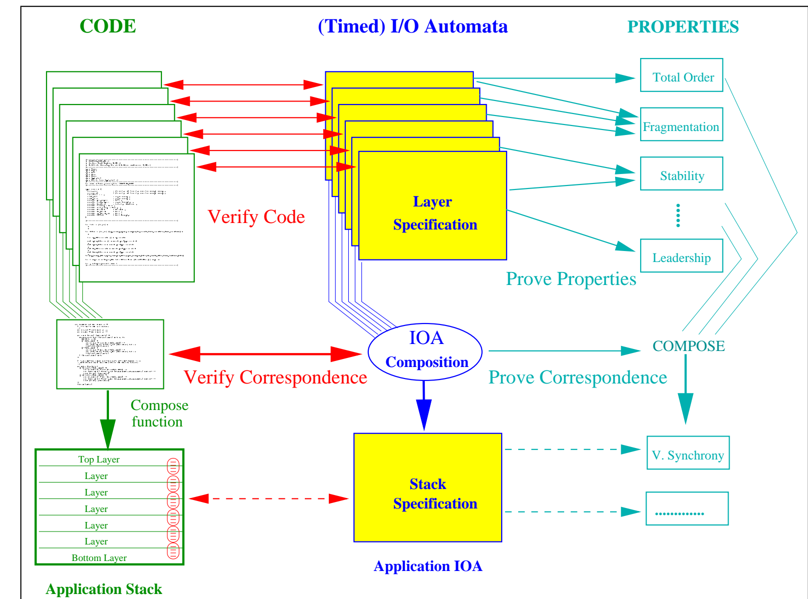
Export: – Generate print-representation of display (genuine OCaml-code)



Verification of System Properties

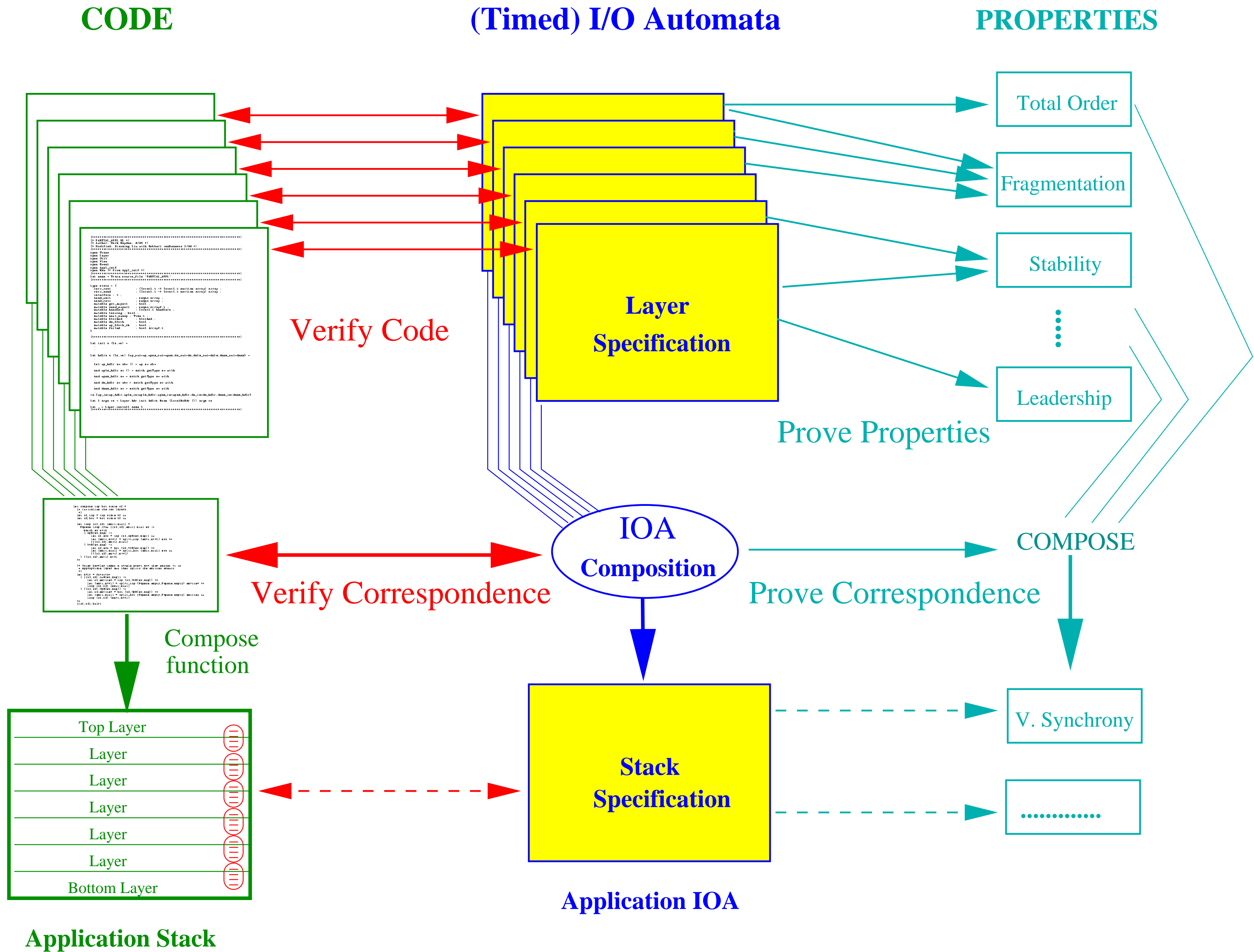
— Separation of Concerns —

(in Progress)



- **Verify Protocol Layers**
  - Specify layers as I/O automata
    (Timed I/O automata for synchronization, liveness,. . .)
  - Verify code wrt. specification (higher-order)
  - Prove properties from specification (first-order)
- **Verify Protocol Stacks**
  - Verify code of layer composition wrt. IOA composition
  - Prove that IOA composition leads to property composition
  - ↦ Automatic derivation of application stack specification & properties
  - + Easier proofs of global system invariants
- **Make automated proofs comprehensible**
  - Use successful proofs for documentation, failed proofs for debugging

Verification Methodology



Fundamental Techniques for Code Verification

- **Programming Logic for Ocaml**
  - Derived inference/evaluation rules for each Ocaml language construct

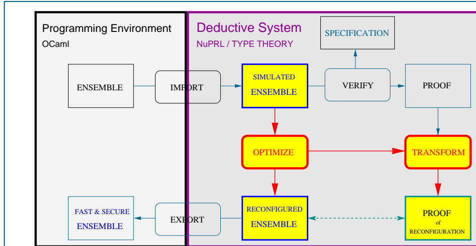
  $\mapsto$ Formal reasoning on the level of the programming language

- **Extended Type Inference**
  - Conventional type checking + constraint checking

    (array bounds, division errors, boolean annotations, ...)

- **Standard Reasoning Techniques**
  - Function evaluation / definition unfolding
  - Lemma application, equality substitution, arithmetic decision procedures
  - Constructive first-order proof search (Connection method)
  - Induction techniques (Rippling)
  - Theory modules

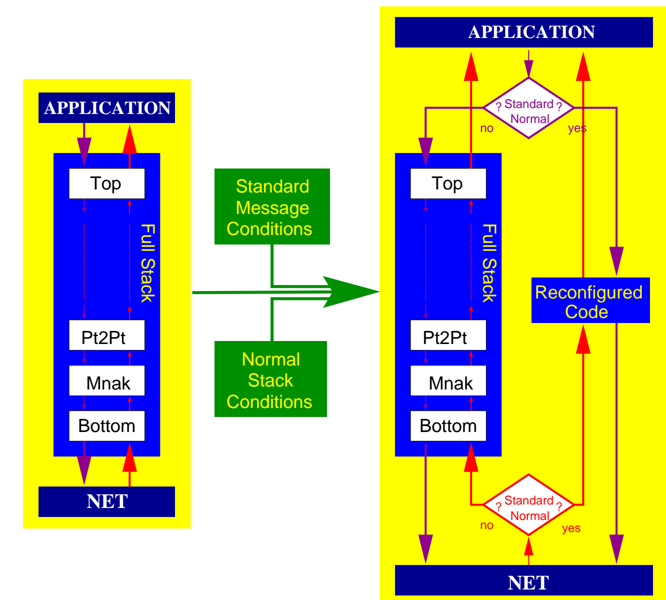
- **Specialized Proof Tactics**
  - Reasoning follows global structure of layer code
  - Tactics for Ensemble's infrastructure code (composition, ...)



Fast-Track Reconfiguration of Protocol Stacks

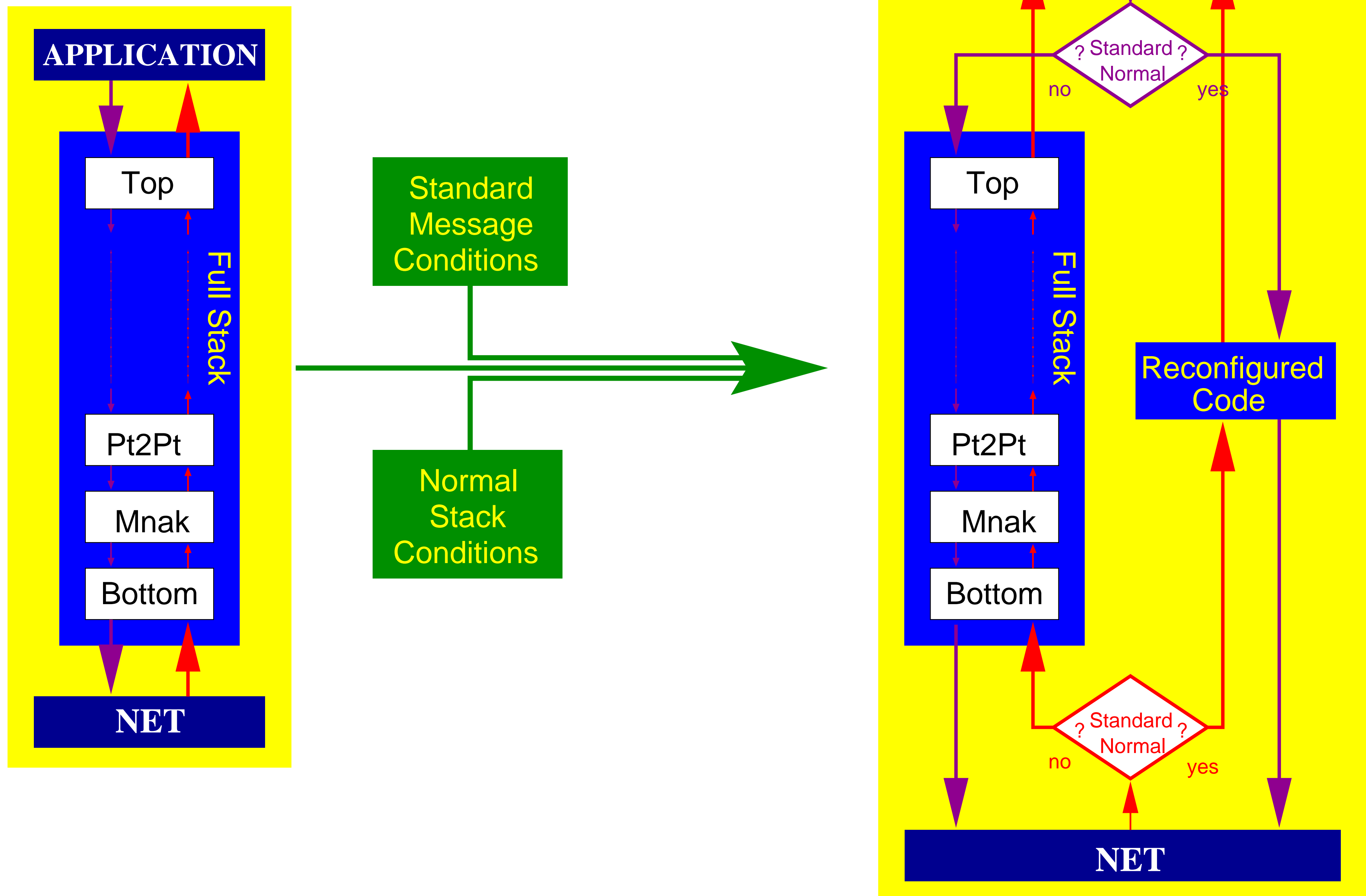
- **Redundancies in Application Stacks**
  - Dead code, error handling, intra-stack communication, large headers...
- **Optimize Code of Protocol Stacks**
  - State assumptions about common events / states
  - Analyze path of events through stack & rewrite code
  - Compress headers of common messages

  ↦ Efficiency improvements of factor 50 possible

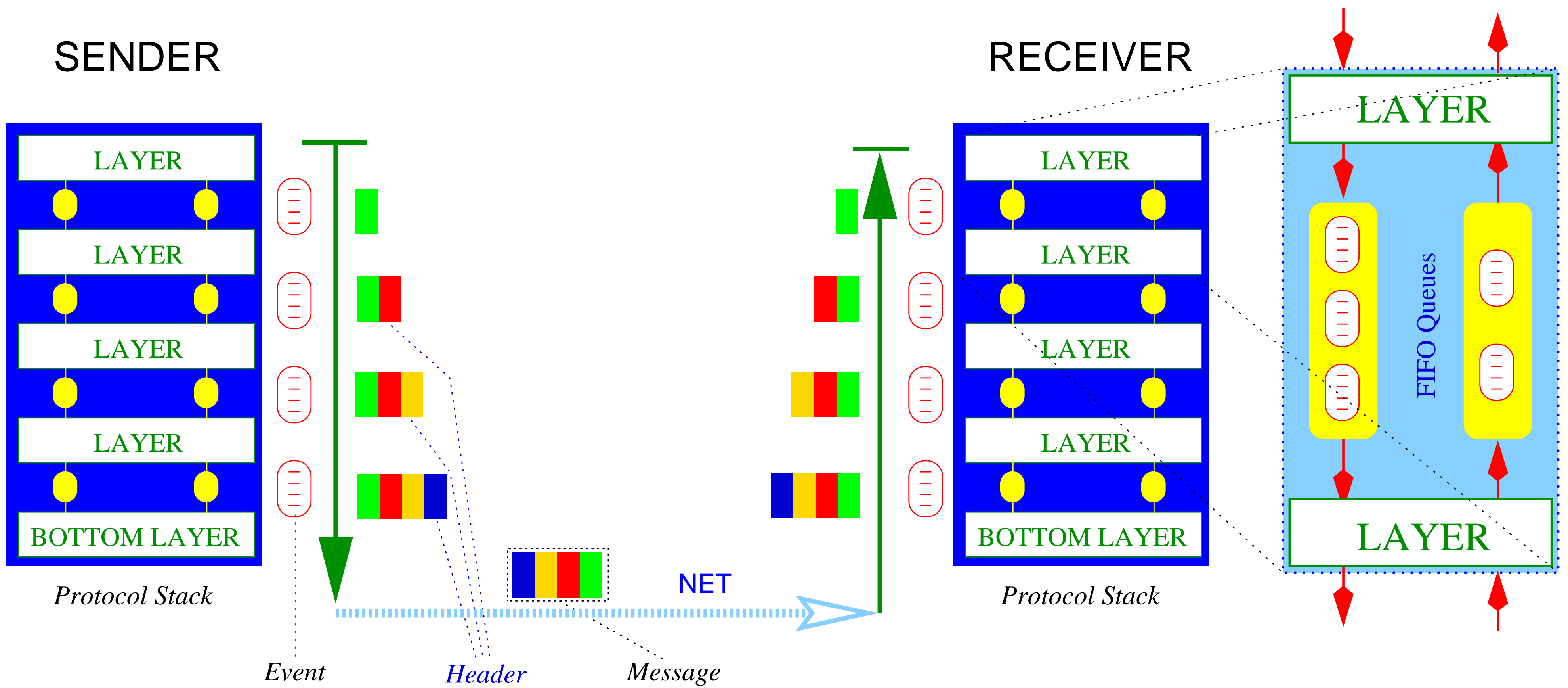


- **Reconfiguration by hand error-prone**
  - No a priori reconfiguration (too many configurations)
- **Provide Formal Reconfiguration Tools**
  - General optimization strategies for OCaml programs
  - Specialized strategies for Ensemble layers and composition
  - Automatic data compression

Stack Reconfiguration: Optimized System



Protocol Layering Architecture



Fundamental Reconfiguration Techniques

- **Symbolic Computation**
  - Derived evaluation rules for each OCAML language construct
  - β-reduction, η-reduction, function inlining
  - General strategies for partial evaluation ($\mapsto$ optimizing compilers)
  - Specialized strategies follow ENSEMBLE's code structure
- **Knowledge-based Simplification**
  - Equality substitution (assumptions + lemmata) & equality resoning
  - Strategies for (conditional) rewriting
- **Interactive Reconfiguration of Individual Layers**
  - User guides macro steps and decides when to end reconfiguration
- **Automatic Verification of Reconfigured Code**
  - Provide proof tactics that mirror transformation steps
  - Verification follows reconfiguration script (no search, success guaranteed)

  $\mapsto$ Proof: reconfigured code $\equiv$ original code

High-Level Reconfiguration Techniques

- **Optimization tactics alone are too slow**
  - Many rewrite steps + extremely large terms
  - But messages pass through most layers unchanged
- **Prove reconfiguration theorems for individual layers**
  - Conditions about typical event structure and normal layer state
  - For up- & downgoing events, sending & broadcasting
  - Layer reconfiguration with some user interaction (once and for all)
  - Verification of results created automatically
- **Prove higher-order theorems about stack operations**
  - Composition of reconfigured layers: up/down, linear/bouncing/splitting trace
  - Wrapping stacks with header compression & expansion

  $\mapsto$ Derived inference rules (logical laws of protocol stacks)
- **Reconfigure stacks by composing theorems**
  - Apply composition theorems to individual layer reconfiguration theorems
  - Generate code for header compression/expansion of common messages
  - Apply wrapping theorems to reconfigured stack
  - Generate code for fast-track through protocol stack

  $\mapsto$ Efficient reconfiguration of applications protocol stacks (linear time)

Reconfiguration of Protocol Layers — Examples

```
THM Pt2ptReconfDnMESend_verif
  ∀s_pt2pt:Pt2pt.state. ∀ls:View.local. ∀ev:Event.t. ∀msg:Messages.
    RECONFIGURING LAYER Pt2pt
     FOR EVENT     DnM(ev, msg)
     AND STATE     s_pt2pt
     ASSUMING      getType ev = ESend  ∧  getPeer ev ≠ ls.rank
    YIELDS EVENTS [:DnM(ev, Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)), msg)):]
       AND STATE  s_pt2pt[.sends.(getPeer ev)
                    ← Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) msg]

THM MnakReconfDnMESend_verif
  ∀s_mnak:Mnak.state. ∀ls:View.local. ∀ev:Event.t. ∀msg:Messages.
    RECONFIGURING LAYER Mnak
     FOR EVENT     DnM(ev, msg)
     AND STATE     s_mnak
     ASSUMING      getType ev = ESend
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, msg)):]
       AND STATE  s_mnak

THM BottomReconfDnMESend_verif
 ∀s_bottom:Bottom.state. ∀ls:View.local. ∀ev:Event.t. ∀msg:Messages.
    RECONFIGURING LAYER Bottom
     FOR EVENT     DnM(ev, msg)
     AND STATE     s_bottom
     ASSUMING      getType ev = ESend  ∧  s_bottom.enabled
    YIELDS EVENTS [:DnM(ev, Full(NoHdr, msg)):]
       AND STATE  s_bottom
```

Reconfiguration of Protocol Stacks — Examples

- **Composition Theorem for Downgoing Events**

```
THM ComposeDnLinear
 ∀Upper,Lower,s_lower,s1_lower,s_upper,s1_upper, ev, msg, msg1, msg2
   RECONFIGURING LAYER Upper FOR EVENT    DnM(ev, msg)    AND STATE s_upper
                            YIELDS EVENTS [:DnM(ev, msg1):] AND STATE s1_upper
 ∧ RECONFIGURING LAYER Lower  FOR EVENT    DnM(ev, msg1)   AND STATE s_lower
                            YIELDS EVENTS [:DnM(ev, msg2):] AND STATE s1_lower
  ⇒ RECONFIGURING LAYER compose Upper Lower
       FOR EVENT    DnM(ev, msg)    AND STATE (s_upper, s_lower)
    YIELDS EVENTS [:DnM(ev, msg2):] AND STATE (s1_upper, s1_lower)
```

- **Generated Reconfiguration Theorem for Application Stack**

```
THM ReconfStackDnMESend_Pt2ptMnakBottom
 ∀s_pt2pt:Pt2pt.state.∀s_mnak:Mnak.state.∀s_bottom:Bottom.state. ∀ls, ev, msg
  RECONFIGURING LAYER Pt2pt::Mnak::Bottom
     FOR EVENT  DnM(ev, msg)
     AND STATE  (s_pt2pt, s_mnak, s_bottom)
     ASSUMING   getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
  YIELDS EVENTS [:DnM(ev, Full(NoHdr, Full(NoHdr,
                      Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),msg)))):]
     AND STATE  ( s_pt2pt[.sends.(getPeer ev)
                    ←Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) msg]
                , s_mnak
                , s_bottom
                )
```

Header Compression — Examples

- **Generated Code for Compression and Expansion**

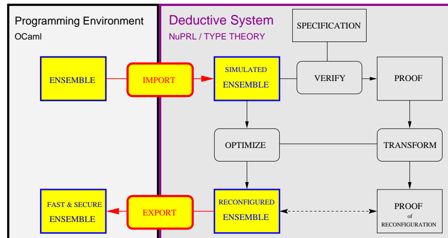
```
let compress msg = match msg with
   Full(NoHdr, Full(NoHdr, Full(Data(seqno), msg))) -> OptSend(seqno, msg)
 | Full(NoHdr, Full(Data(seqno), Full(NoHdr, msg))) -> OptCast(seqno, msg)
 | msg                                               -> Normal(msg)

let expand msg = match msg with
   OptSend(seqno, msg) -> Full(NoHdr, Full(NoHdr, Full(Data(seqno), msg)))
 | OptCast(seqno, msg) -> Full(NoHdr, Full(Data(seqno), Full(NoHdr, msg)))
 | Normal(msg)         -> msg
```

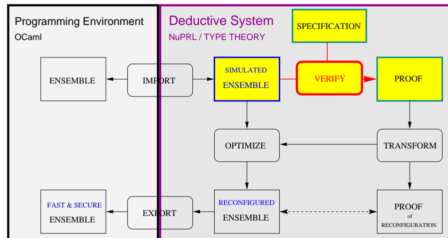
- **Reconfiguration Theorem for Stack with Compression**

```
THM ReconfStackDnMESend_Pt2ptMnakBottom
 ∀s_pt2pt:Pt2pt.state.∀s_mnak:Mnak.state.∀s_bottom:Bottom.state. ∀ls, ev, msg
  RECONFIGURING LAYER Pt2pt::Mnak::Bottom  WRAPPED WITH COMPRESSION
     FOR EVENT  DnM(ev, msg)
     AND STATE  (s_pt2pt, s_mnak, s_bottom)
     ASSUMING   getType ev = ESend ∧ getPeer ev ≠ ls.rank ∧ s_bottom.enabled
  YIELDS EVENTS [:DnM(ev,OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), msg)):]
     AND STATE  ( s_pt2pt[.sends.(getPeer ev)
                    ←Iq.add s_pt2pt.sends.(getPeer ev) (getIov ev) msg]
                , s_mnak
                , s_bottom
                )
```

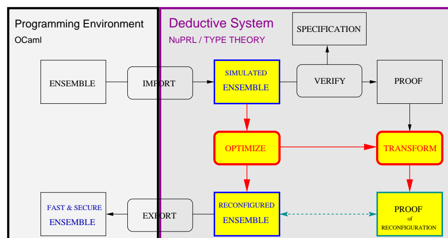
Results



- √ Formal semantics for Ocaml-subset
- √ Import/Export Algorithms
- √ Ocaml-Libraries available in NuPRL (1200 objects, 20000 lines)
- √ Ensemble-Code available in NuPRL (1500 objects, 30000 lines)



- √ Inference Calculus for Ocaml
- √ Example Verification of simplified layer code
- √ IOA-specification + paper verification of total order layer



- √ Symbolic Evaluation Strategies for Ocaml
- √ Reconfiguration Theorems for 30 System Layers
- √ Reconfiguration Tactics for Protocol Stacks
- √ Automatic Reconfiguration/Compression of 25-Layer Stack

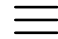
Future Work

- **Formalization**
  - Full imperative behavior,
  - Full module system: inheritance, functors, abstract types . . .
- **Modularization**
  - Theory & tactic modules corresponding to Ensemble modules
  - Modular formal database of verified domain knowledge
- **Applications**
  - Specify and verify important protocols (total order, virtual synchrony, . . .)
  - Integrate reconfigured protocol stack into application code (Ensemble Jukebox)
- **Synthesis**
  - Derive IOA-specification from layer code
  - Derive Ensemble-like implementations from IO-automata
  - Data type refinement
- **Tools**
  - Extended static typechecking
  - Theorem prover for first-order logic with induction
  - Professionalize existing tools (more automation/robustness)

⋮

Conclusion

Logical programming environment for communication systems



- **Improving the efficiency of modular systems**
  - Automatic reconfiguration for concrete applications
- **Hardening the security of distributed systems**
  - Verification of protocol properties and system invariants
- **Proof tools as debugging support**
  - Failed proofs provide insights
- **Proof as formal documentation**
  - Explanation of system's behavior

High-assurance design paradigm for distributed systems