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  \(\leadsto\) programming
  - Integers, lists, inductive types  \(\leadsto\) inductive definition
  - Propositions as types, equality type, void, top, universes  \(\leadsto\) logic
  - Subsets, subtyping, quotient types  \(\leadsto\) mathematics
  - (Dependent) intersection, union, records  \(\leadsto\) 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”  \(\leadsto\) display forms

- **Expressions independent of types**
  - No restriction on expressions that can be defined  \(\leadsto\) Y combinator
  - Expressions in proofs must be typeable  \(\leadsto\) “total” functions

- **Refinement calculus**
  - Top-down sequent calculus  \(\leadsto\) interactive proof development
  - Computation rules and extract terms  \(\leadsto\) program development

- **User-defined extensions possible**
  - Language extensions (abstractions) + user-defined inference rules (tactics)
Features of Nuprl’s Proof System

- Interactive **proof editor**  \(\leadsto\) readable proofs
- Flexible **definition** mechanism  \(\leadsto\) user-defined terms
- Customizable **term display**  \(\leadsto\) flexible notation
- **Structure editor** for terms  \(\leadsto\) no ambiguities
- **Tactics & decision procedures**  \(\leadsto\) proof automation
- **Program evaluation and extraction**  \(\leadsto\) program synthesis
- **Library** mechanism  \(\leadsto\) large user-theories
- **Formal documentation** mechanism  \(\leadsto\) LaTeX, HTML
- **Collection of cooperating processes**
  - Enables asynchronous, distributed & cooperative theorem proving

- **Centered around a common knowledge base**
  - Persistent data base, version control, dependency tracking
  - System structure designed within the library

- **Connected to external systems**
  - MetaPRL (fast rewriting, multiple logics) (Hickey & Nogin, 1999)
  - JProver (matrix-based intuitionistic theorem prover) (IJCAR 2001)
  - Multiple user interfaces — interoperability

- Accountability

- Customizability

- Collaborative proving
Link ENSSEMBLE 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)
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
Designing Reliable, High-Performance Networks...
**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 $\beta$-reduction

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

- **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**

**Programming Environment**
- OCaml
  - Camlp4 Parser Preprocessor
  - Abstract Syntax Tree
  - Conversion module modified Pretty printer

**Deductive System**
- NuPRL / TYPE THEORY / Meta-Language ML
  - Intermediate Code NuPRL-ML
  - Term + Object Generators
  - NuPRL Library + Representations of basic Ocaml-con structs
  - Simulated Ocaml-Code
    - Abstractions Display Forms Type Information

**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**

![Diagram showing the process from Programming Environment to Deductive System]

---

**Actual Ensemble code available for formal reasoning**
● System properties
  e.g. FIFO: “Messages are received in the same order in which they were sent”
  \[ \forall i, j, k, l \in |\text{tr}|. (i < j \land \text{tr}[i] \downarrow \text{tr}[k] \land \text{tr}[j] \downarrow \text{tr}[l]) \Rightarrow k < l \]

● 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

<table>
<thead>
<tr>
<th>Specification</th>
<th>FifoNetwork()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>in-transit: queue of (\langle Address, Message\rangle)</td>
</tr>
<tr>
<td>Actions</td>
<td>(\text{Send}(dst : Address; msg : Message))</td>
</tr>
<tr>
<td></td>
<td>(\text{condition: true}) {in-transit.append((\langle dst, msg\rangle))}</td>
</tr>
<tr>
<td></td>
<td>(\text{Deliver}(dst : Address; msg : Message))</td>
</tr>
<tr>
<td></td>
<td>(\text{condition: in-transit.head()} = \langle dst, msg\rangle) {in-transit.dequeue()}</td>
</tr>
</tbody>
</table>

#### Concrete behavioral specification

<table>
<thead>
<tr>
<th>Specification</th>
<th>FifoProtocol((p : Address))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>send-window, recv-window, ...</td>
</tr>
<tr>
<td>Actions</td>
<td>(\text{Above.Send}(dst : Address; msg : Message))</td>
</tr>
<tr>
<td></td>
<td>{ ... list of individual sub-actions ... }</td>
</tr>
<tr>
<td></td>
<td>(\text{Below.Send}(dst : Address; \langle hdr, msg\rangle : \langle Header, Message\rangle))</td>
</tr>
<tr>
<td></td>
<td>(\text{Below.Deliver}(dst : Address; \langle hdr, msg\rangle : \langle Header, Message\rangle))</td>
</tr>
<tr>
<td></td>
<td>(\text{Above.Deliver}(dst : Address; msg : Message))</td>
</tr>
<tr>
<td></td>
<td>(\text{Timer()})</td>
</tr>
</tbody>
</table>

---

I/O-automata represented as dependent product types
let name = Trace.source_file "PT2PT"

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 $\rightarrow$ system properties
  - Easy for benign networks $\sim$ 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

**Sender**

- Layer
- Layer
- Layer
- Layer
- Bottom Layer

**Receiver**

- Layer
- Layer
- Layer
- Layer
- Bottom Layer

Protocol Stack

Event

Message

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)

```hs
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;dl1m_out=dl1m;dlnm_out=dlnm}
  = ...

  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 abv hdr = ...
    and upnm_hdlr ev = ...
    and dn_hdlr ev abv =
      if s.enabled then
        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
    : ...

Mnak (350 lines)

```hs

```hs
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)

```hs

```hs
let init _ (ls,vs) = {...........
let hdlrs _ (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

let l args vs = Layer.hdr init hdlrs args vs
Layer.install name (Layer.init l)

```
● **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
2. Compose into optimizations of protocol stacks
3. Integrate message header compression
4. Generate code from optimization theorems and reconfigure system

A priori: **ENSEMBLE + NuPRL experts**

Automatic: application designer

Automatic: :
• 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 $\text{Send}(p,m)$ and $\text{Deliver}(p,m)$ events
  
  \[ \text{e.g. } \text{Reliable}(tr) \equiv \forall p,q:\text{PID}. \forall m:\text{Msg. } \text{Send}(p,m) \in tr \Rightarrow \text{Deliver}(q,m) \in tr \]

- Characterize switchable properties by meta-properties
  - Predicates on communication properties
  - Expressed by relation $R$ between traces $tr_u, tr_l$ above/below a protocol
    
    \[ R \text{ preserves } P \equiv \forall tr_u, tr_l:\text{Trace. } (P(tr_l) \land tr_u R tr_l) \Rightarrow P(tr_u) \]

Examples of meta-properties:

\[
\begin{align*}
tr_u & \ R_{\text{safety}} & tr_l & \equiv tr_u \subseteq tr_l \\
tr_u & \ R_{\text{asynchrony}} & tr_l & \equiv tr_u \text{ swap-adjacent}_{\text{loc}(e) \neq \text{loc}(e')} tr_l \\
tr_u & \ R_{\text{delayable}} & tr_l & \equiv tr_u \text{ swap-adjacent}_{\text{msg}(e) \neq \text{msg}(e')} \land \text{is-} \text{send}(e) \neq \text{is-} \text{send}(e')} tr_l \\
tr_u & \ R_{\text{send-enabled}} & tr_l & \equiv \exists e:\text{Events. } \text{is-} \text{send}(e) \land tr_u = tr_l \circlearrowleft e \\
tr_u & \ R_{\text{memoryless}} & tr_l & \equiv \exists e:\text{Events. } tr_u = [e_i \in tr_l | \text{msg}(e) \neq \text{msg}(e_i)] \\
\text{R_composable}(tr_u, tr_1, tr_2) & \equiv tr_u = tr_1 \circlearrowleft tr_2 \land \forall e_1 \in tr_1, \forall e_2 \in tr_2. \text{msg}(e_1) \neq \text{msg}(e_2)
\end{align*}
\]
Verifying the Correctness of Switching

- Characterize **switch invariant** between \( tr_u \) and \( tr_1, \ldots, tr_n \)
  - \( tr_u \) 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

\[
\forall P: \text{TraceProperty}. \forall tr_u, tr_1, \ldots, tr_n: \text{Trace}.
\]
\[
\text{switchable}(P) \land \text{switch_invariant}(tr_u; tr_1, \ldots, tr_n)
\]
\[
\Rightarrow ( \forall i \leq n. P(tr_i) \Rightarrow P(tr_u) )
\]

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, Xuming 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