Automatisierte Logik und Programmierung

Einheit 16

Anwendungsbeispiele

1. Mathematik:
   Automatisierung von Kategorientheorie
2. Programmierung:
   Analyse und Optimierung verteilter Systeme
3. Aktuelle Fragestellungen:
   Language-based Security

Automating Proofs in Category Theory

- **Category Theory analyzes structure**

  What properties of mathematical domains depend only on structure?

  · Focus on mathematical objects and morphisms on these objects

  · Develop a generic framework for expressing abstract properties

  – Results have a wide impact on mathematics and computer science

- **Category Theory is extremely precise**

  – Even basic proofs can be very tedious

  – Diagrams illustrate the essential insights but are not considered proofs

- **Can category theoretical proofs be automated ?**

  – Detailed proofs often follow standard patterns of reasoning

  · It should be possible to formalize these as proof rules

  – Key insights are often considered "the only obvious choice"

  · It should be possible to write tactics that construct proofs

Axiomatization of Elementary Category Theory

Make standard reasoning patterns precise

- **Formulate as first-order reasoning system**
  - Rules about products, functors, natural transformations, ...
  - Analysis and synthesis of structures
  - Equational reasoning is essential in most proofs
- **Example: Analysis rules for functors**
  - Rules essentially explain what functors are and how to use them

$$\frac{\Gamma \vdash F : \mathsf{Fun}[\mathsf{C},\mathsf{D}]\,, \quad \Gamma \vdash \mathsf{A} : \mathsf{C}}{\Gamma \vdash F^1 A : \mathsf{D}}$$

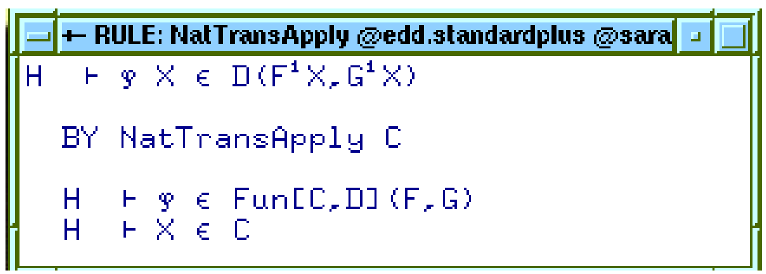
$$\frac{\Gamma \vdash F : \mathsf{Fun}[\mathsf{C},\mathsf{D}]\,, \quad \Gamma \vdash \mathsf{A},\mathsf{B} : \mathsf{C}, \quad \Gamma \vdash \mathsf{f} : \mathsf{C}(\mathsf{A},\mathsf{B})}{\Gamma \vdash F^2 f : \mathsf{D}(F^1 A, F^1 B)}$$

$$\frac{\Gamma \vdash F : \mathsf{Fun}[\mathsf{C},\mathsf{D}]\,, \quad \Gamma \vdash \mathsf{A},\mathsf{B},\mathsf{C} : \mathsf{C}, \quad \Gamma \vdash \mathsf{f} : \mathsf{C}(\mathsf{A},\mathsf{B}), \quad \Gamma \vdash \mathsf{g} : \mathsf{C}(\mathsf{B},\mathsf{C})}{\Gamma \vdash F^2(g \circ f) = F^2 g \circ F^2 f}$$

$$\frac{\Gamma \vdash F : \mathsf{Fun}[\mathsf{C},\mathsf{D}]\,, \quad \Gamma \vdash \mathsf{A} : \mathsf{C}}{\Gamma \vdash F^2 1_{\mathsf{A}} = 1_{\mathsf{F^1A}}}$$

Non-conservative extension with CTT support

- **Encode the language of Category Theory**
  - Abstractions explain category theoretical concepts in CTT
  - Display forms intruduce MacLane's textbook notation
- **Implement first-order inference rules**
  - Introduce (top-down) rule objects for each inference rule



  - Justify inference rules by proving them correct in CTT
  - Convert primitive inferences into simple tactics

Proof Automation

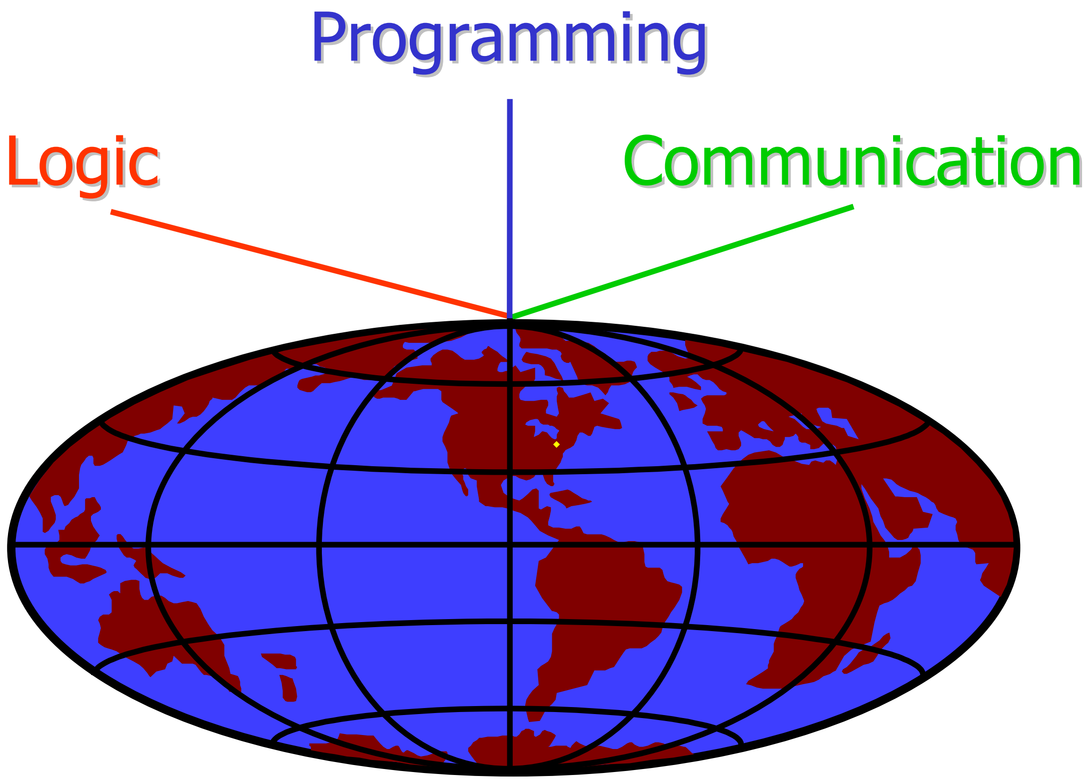
- **Most steps are straightforward decompositions**
  - Apply analysis and synthesis rules where possible
  - Block application of analysis rules that create subgoals previously decomposed by a synthesis rule
  - Assert goals that will re-occur several times in subproofs
- **Equality reasoning needs guidance**
  - Convert equality rules into directed rewrite rules
  - Use Knuth-Bendix completion to make the rewrite system confluent
- **Specify functors component-wise**
  - E.g. use $\vartheta^1 \mathtt{G}^1 \mathtt{A}^1 \mathtt{X} \equiv \mathtt{G}^1\mathtt{<A,X>}$ and $\vartheta^1 \mathtt{G}^1 \mathtt{A}^2 \mathtt{h} \equiv \mathtt{G}^2\mathtt{<1}_A\mathtt{,h>}$ instead of $\vartheta \equiv \lambda\mathtt{G,A...}$, which is no category theory expression
  - Method keeps reasoning methods first-order
- **Guess witnesses for existential quantifiers**
  - Introduce meta-variables for existentially quantified variables
  - Decompose as long as possible and focus of typing subgoals
  - Introduce the simplest term that satisfies the typing requirements

AN EXAMPLE PROOF

*- PRF : Currying Tactic Test @edd.standardplus @sarah

```
* top
∀C,D,E:Categories.  Fun[C×D,E] ≙ Fun[C,Fun[D,E]]

BY Unfold `CatIso` 0 THEN prover·

* 1

1. C : Categories
2. D : Categories
3. E : Categories
⊢ ∃ϑ:Fun[Fun[C×D,E],Fun[C,Fun[D,E]]].
 ∃η:Fun[Fun[C,Fun[D,E]],Fun[C×D,E]]. ϑ and η are inverse

BY GUESS ⌜ϑ⌝ THEN GUESS ⌜η⌝·

* 1 1

4. theta : Top
5. ∀phi,k,X1,X,f,G:Top.
      (ϑ^1 G^2 f X ≡ G^2 <f, 1X>
      ∧ ϑ^1 G^1 X^1 X1 ≡ G^1 <X, X1>
      ∧ ϑ^1 G^1 X^2 k ≡ G^2 <1X, k>
      ∧ ϑ^2 φ X X1 ≡ φ <X, X1>)
6. eta : Top
7. ∀X1,phi,g,f,X,A,G:Top.
      (η^1 G^1 <A, X> ≡ G^1 A^1 X
      ∧ η^1 G^2 <f, g> ≡ ((G^2 f cod(g))∘G^1 dom(f)^2 g)
      ∧ η^2 φ <A, X1> ≡ φ A X1)
⊢ ϑ and η are inverse

BY Unfold `FunInverse` 0 THEN AutoCAT2
```

Towards Reliable, High-Performance Networks



Secure software infrastructure

Apply Formal Reasoning to a real-world system

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

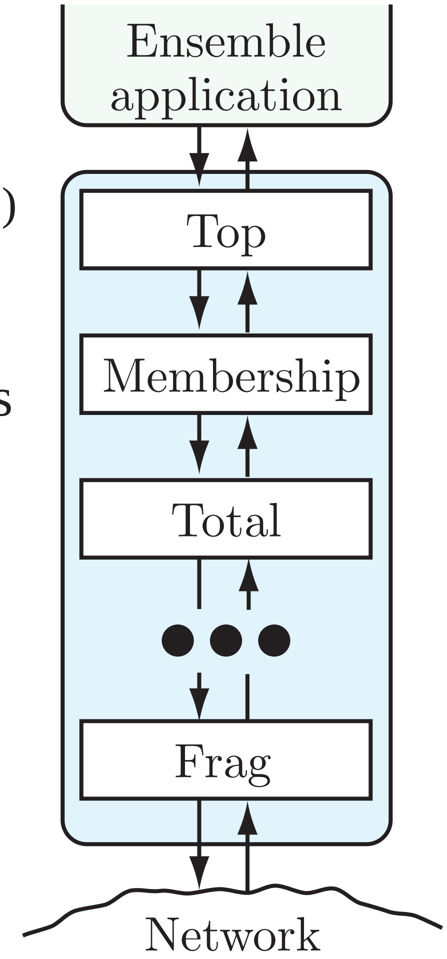
– Modeled as state/event machines

Implementation in Objective Caml (INRIA)

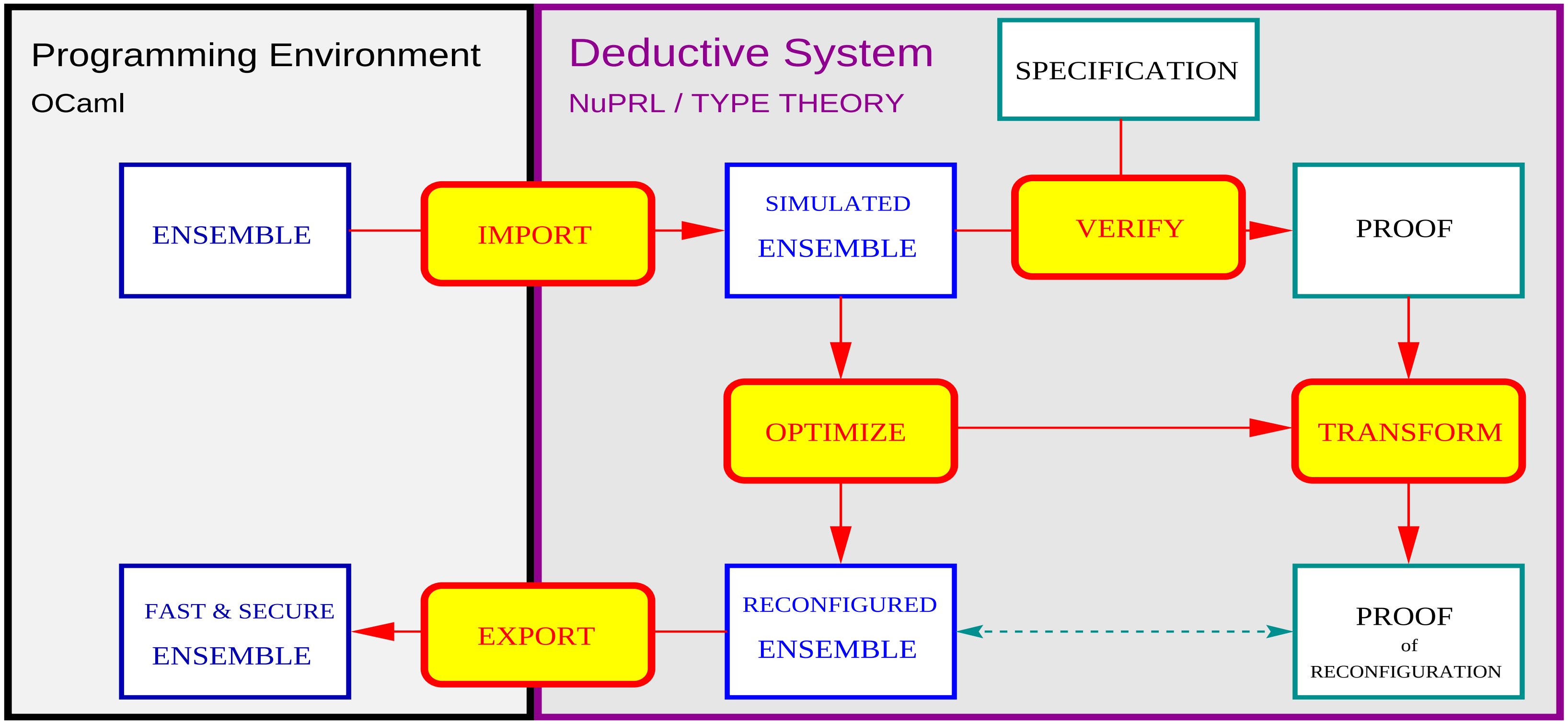
– Easy maintenance (small code, good data structures)

– Mathematical semantics, strict data type concepts

– Efficient compilers and type checkers



Formal reasoning about a real-world system



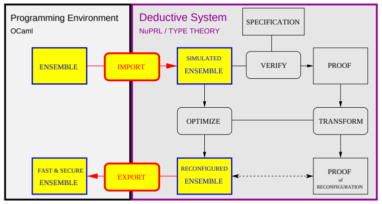
Link the ENSEMBLE and Nuprl systems

– Embed ENSEMBLE's code into Nuprl's language

– Verify protocol components and system configurations

– Optimize performance of configured systems

– Formally design and verify new protocols



Embedding Ensemble's code into Nuprl

- **Develop type-theoretical semantics of** OCaml
  - Functional core, pattern matching, exceptions, references, modules,…
- **Implement using** Nuprl's **definition mechanism**
  - Represent OCaml's semantics via abstraction objects
  - Represent OCaml's syntax using associated display objects
- **Develop programming logic for** OCaml
  - Implement as rules derived from the abstract representation
  - Raises the level of formal reasoning from Type Theory to OCaml
- **Develop tools for importing and exporting code**
  - Translators between OCaml program text and Nuprl terms

OCaml Semantics: The functional core

- **Basic OCaml expressions similar to CTT terms**
  - Numbers, tuples, lists etc. can be mapped directly onto CTT terms
- **Complex data structures have to be simulated**

  Records $\{f_1\texttt{=}e_1; \ldots; f_n\texttt{=}e_n\}$ are functions in $\texttt{f:FIELDS}\rightarrow T\texttt{[f]}$
  - Abstraction for representing the semantics of record expressions

    $\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;}\textit{next}\texttt{)} \equiv \lambda\texttt{f. if f=}\textit{field}\texttt{ then } e \texttt{ else } \textit{next}\texttt{(f)}$
  - Display form for representing the flexible syntax of record expressions

	$\{\textit{field}\texttt{=}e\texttt{;}\ \textit{next}\}$	$\equiv$	$\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;}\textit{next}\texttt{)}$
	$\{\textit{field}\texttt{=}e\}$	$\equiv$	$\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;}\lambda\texttt{f.())}$
HD::	$\{\textit{field}\texttt{=}e\texttt{;}\ \texttt{\#}$	$\equiv$	$\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;\#)}$
TL::	$\textit{field}\texttt{=}e\texttt{;}\ \texttt{\#}$	$\equiv$	$\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;\#)}$
TL::	$\textit{field}\texttt{=}e\}$	$\equiv$	$\texttt{RecordExpr(}\textit{field}\texttt{;}e\texttt{;}\lambda\texttt{f.())}$

- **Sufficient for representing micro protocols**
  - Simple state-event machines, encoded via updates to certain records
  - Transport module and protocol composition require imperative model

Extensions of the semantical model (1)

- **Type Theory is purely functional**
  - Terms are evaluated solely by reduction
  - OCaml has pattern matching, reference cells, exceptions, modules, …
- **Modelling Pattern Matching:** `let` pat=e `in` t

  "Variables of pat in t are bound to corresponding values of e"
  - Evaluation of OCaml-expressions uses an environment of bindings
  - Patterns are functions that modify the environment of expressions

x	$\equiv$	λ`val,t.`λ`env. t (env@`$\{x\mapsto$`val`$\}$`)`
p_1,p_2	$\equiv$	λ`val,t.`λ`env. let <`v_1,v_2`>=val in (`p_1 v_1 `(`p_2 v_2 `t)) env`
$\{f_1$=p_1; . . ; f_n=$p_n\}$	$\equiv$	λ`val,t.`λ`env.` p_1 `(val` f_1`) (..(`p_n `(val` f_n`) t)..) env`
⋮		⋮

  - Local bindings are represented as applications of these functions

`let` p `=` e `in` t $\equiv$ λ`env. (`p `(`e `env)` t`) env`

Extensions of the semantical model (2)

- **Modelling Reference cells**
  - Evaluation of OCaml-expressions may lookup/modify a global store
  - The global store is represented as table with addresses and values

```
ref(e)   ≡ λs,env. let <v,s1>=e s env in
                    let addr=NEW(s1) in <addr, s1[addr←v]>
!e       ≡ λs,env. let <addr,s1>=e s env in <s1[addr], s1>
e1:=e2   ≡ λs,env. let <v,s1>=e2 s env in
                    let <addr,s2>=e1 s1 env in <(), s2[addr←v]>
```

- **Modelling Exceptions**
  - Expressions like `x/y` may raise exceptions, which can be caught
  - Exceptions must have the same type as the expression that raises them
  - An OCaml type T must be represented as `EXCEPTION` + T
- **Modules**
  - Modules are second class objects that structure the name space
  - Modules are represented by operations on a global environment

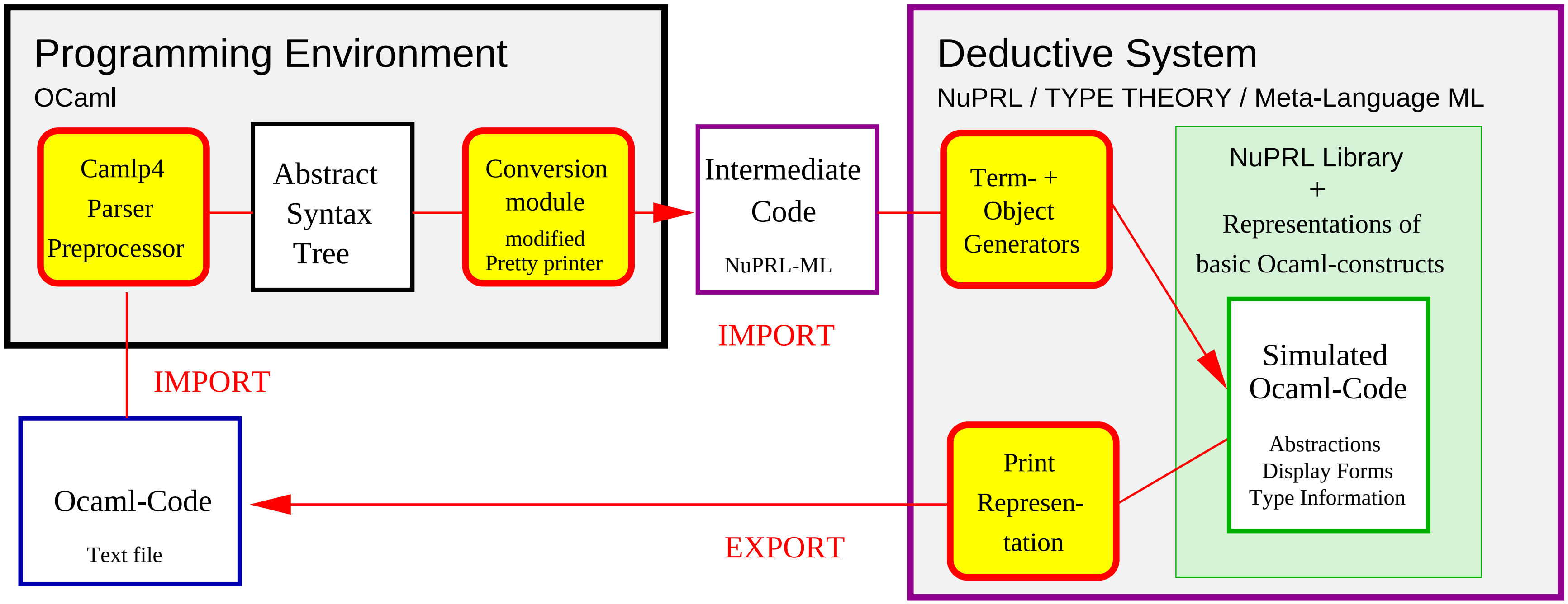
SUMMARY OF THE FORMAL MODEL

- **OCaml expressions are represented as functions**
  - Evaluation depends on environment and store
  - Evaluation results in value or exception and an updated store
  - Nuprl type is $\texttt{STORE} \rightarrow \texttt{ENV} \rightarrow (\texttt{EXCEPTION} + T) \times \texttt{STORE}$
- **Equivalent to Wright/Felleisen model**
  - The standard model for building ML compilers
  - Model combines several mechanisms for evaluating ML programs
  - Nuprl representation simulates these models functionally

$$\Downarrow$$

Genuine OCaml code may occur in formal theorems

Importing and Exporting System Code



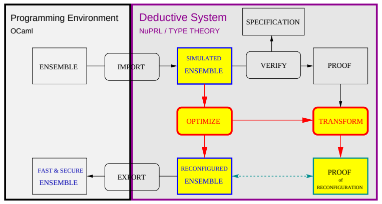
Import: – Parse with Camlp4 parser-preprocessor

– Convert abstract syntax tree into term- & object generators

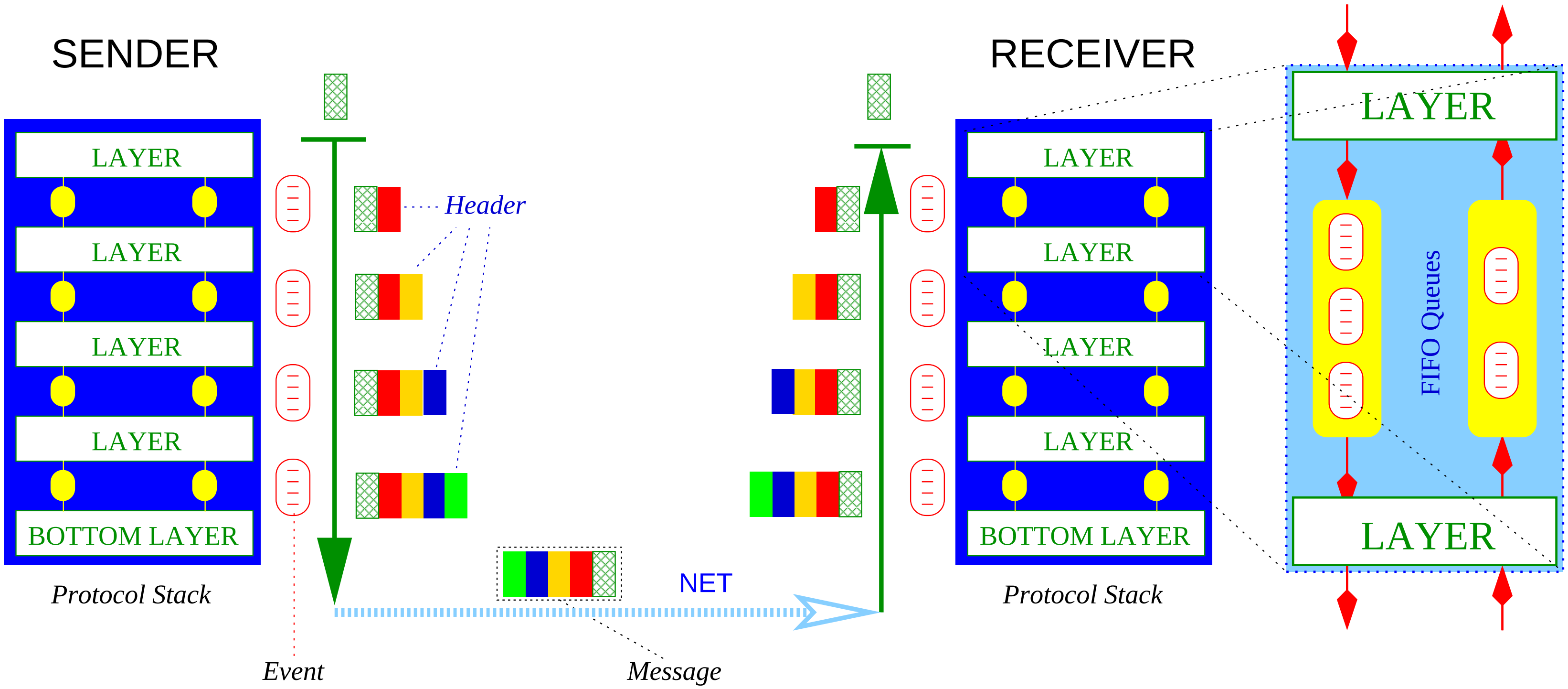
– Generators perform second pass and create Nuprl library objects

Export: – Print-representation is genuine OCaml-code

Actual ENSEMBLE code available for formal reasoning



Optimization of Protocol Stacks



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)

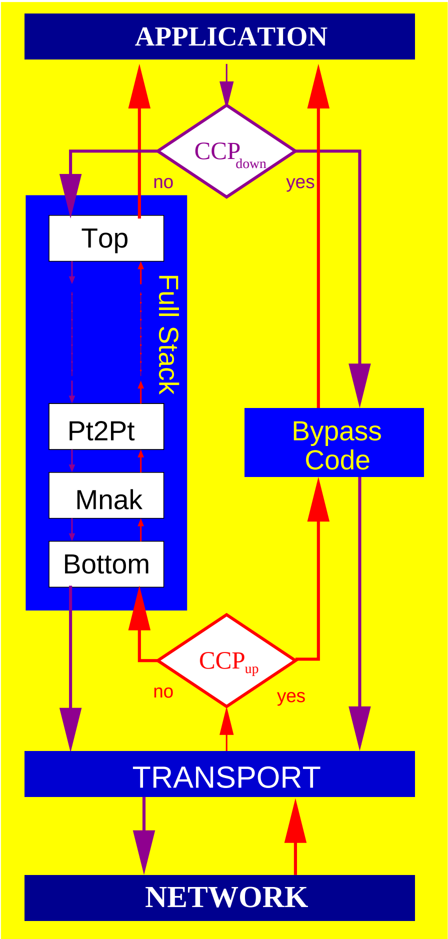
```
let init ack_rate (ls,vs) = {.........}
let hdlrs s (ls,vs) { ......... }
 = ...
 let ...
 and dn_hdlr ev abv     =
   match getType ev with
   | ECast ->
      let iov = getIov ev in
      let buf = Arraye.get s.buf ls.rank in
      let seqno = Iq.hi buf in
      assert (Iq.opt_insert_check buf 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
  ⋮
```

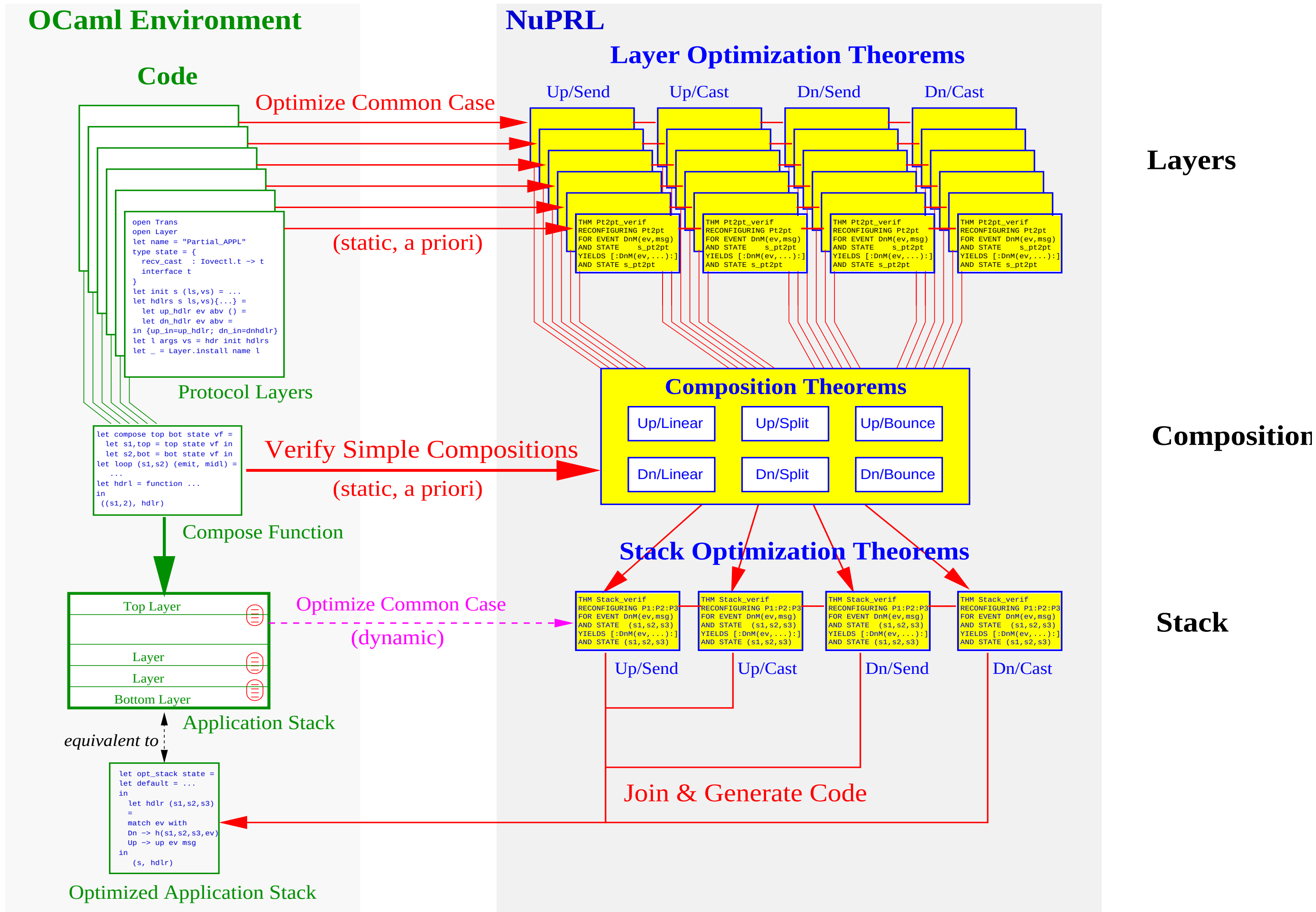
Fast-path Optimization with Nuprl

- **Identify Common Case**
  - Events and protocol states of regular communication
  - Formalize as **C**ommon **C**ase **P**redicate
- **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: ⋮

Static optimization of micro protocols

- **A-priori analysis of common execution sequences**
  - Generate local CCP from conditionals in a layer's code
- **Assuming the CCP, apply code transformations**
  - Controlled function inlining and symbolic evaluation (rewrite tactics)
  - Directed equality substitutions (lemma application)
  - Context-dependent simplifications (substitute part of CCP and rewrite)
- **Store result in library as optimization theorem**

```
 OPTIMIZING LAYER Pt2pt
        FOR EVENT DnM (ev, msg)
        AND STATE s_pt2pt
         ASSUMING (getType ev) = ESend ∧ not (getPeer ev = ls.rank)
   YIELDS HANDLERS dn ev (Full (Data (Iq.hi
                       (Arraye.get s_pt2pt.sends (getPeer ev))), msg))
      AND UPDATES Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))
                       (getIov ev) msg
```

  - Theorem proves correctness of the local optimization
  - Optimizations of micro protocols part of ENSEMBLE's distribution

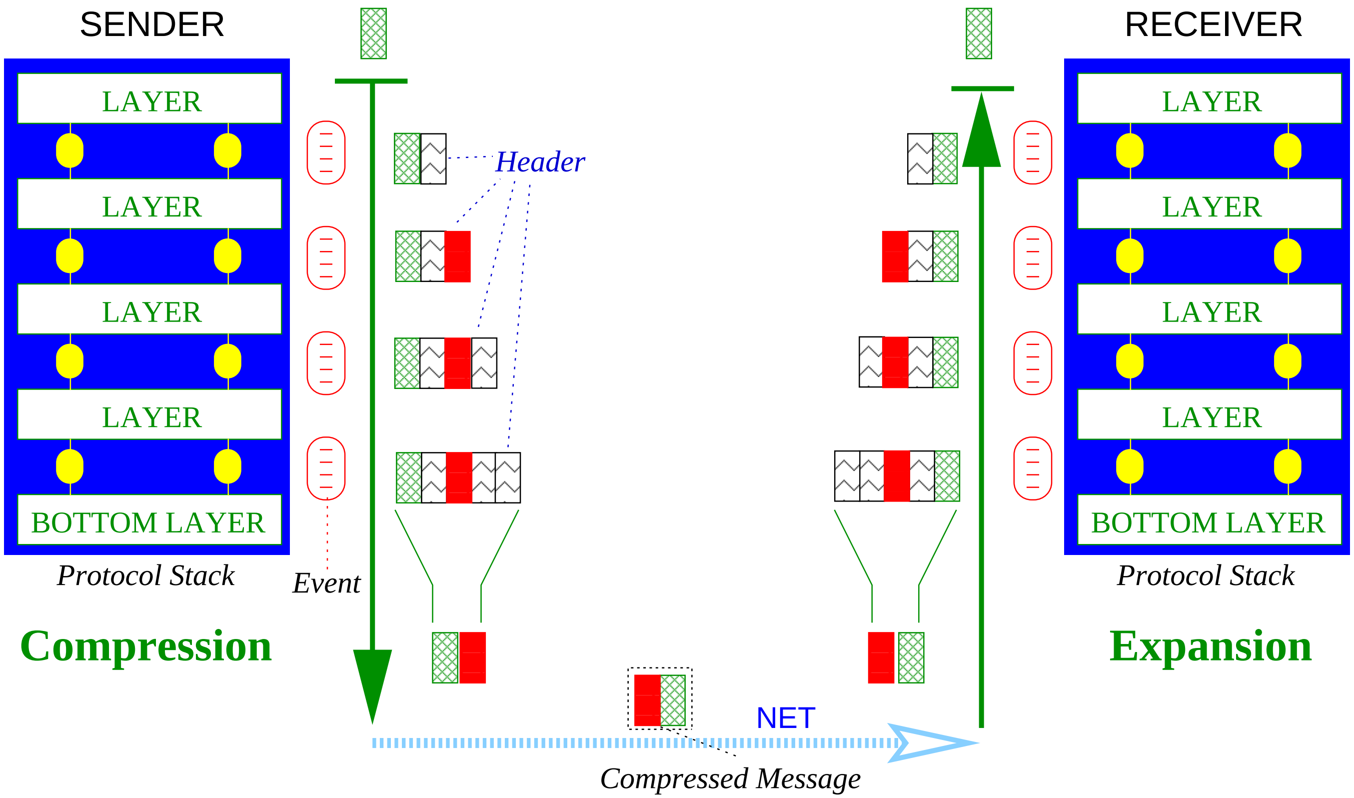
Dynamic Optimization of Application Stacks

- **Compose Optimization Theorems**
  - Consult optimization theorems for individual layers
  - Apply composition theorems to generate stack optimization theorems (Linear, simple split, bouncing – send/receive)

```
    OPTIMIZING LAYER Upper
           FOR EVENT DnM(ev, hdr) AND STATE s_up
     YIELDS HANDLERS dn ev msg1 AND UPDATES stmt1
  ∧ OPTIMIZING LAYER Lower
           FOR EVENT DnM(ev, hdr1) AND STATE s_low
     YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2

 ⇒  OPTIMIZING LAYER Upper || Lower
           FOR EVENT DnM(ev, hdr) AND STATE (s_up, s_low)
     YIELDS HANDLERS dn ev msg2 AND UPDATES stmt2; stmt1
```

  - Formal proof complex because of complex code for composition
- **Optimization of Protocol Stacks in Linear Time**
  - Use of optimization theorems reduces proof burden for optimizer
  - Pushbutton Technology: requires only configuration of stack

Header compression for fast-path code



Integrate compression into optimization process

– Generate code for compression and expansion from fast-path headers

– Combine optimization theorem for stack with compression theorems

– Optimized stack uses compressed headers directly

Example Optimization of `Bottom::Mnak::Pt2pt`

- **Generated optimization theorem for application stack**

```
OPTIMIZING 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 HANDLERS dn ev (Full(NoHdr, Full(NoHdr,
                          Full(Data(Iq.hi s_pt2pt.sends.(getPeer ev)),msg))))
    AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg
```

- **Generated code for header compression**

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

- **Optimization theorem including header compression**

```
OPTIMIZING 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 HANDLERS dn ev (OptSend(Iq.hi s_pt2pt.sends.(getPeer ev), msg))
    AND UPDATES  Iq.add (Arraye.get s_pt2pt.sends (getPeer ev))(getIov ev) msg
```

Code generation

1. Convert Theorems into Code Pieces

– handlers + updates $\mapsto$ command sequence

– CCP $\mapsto$ conditional / case-expression

– Call original event handler if CCP fails

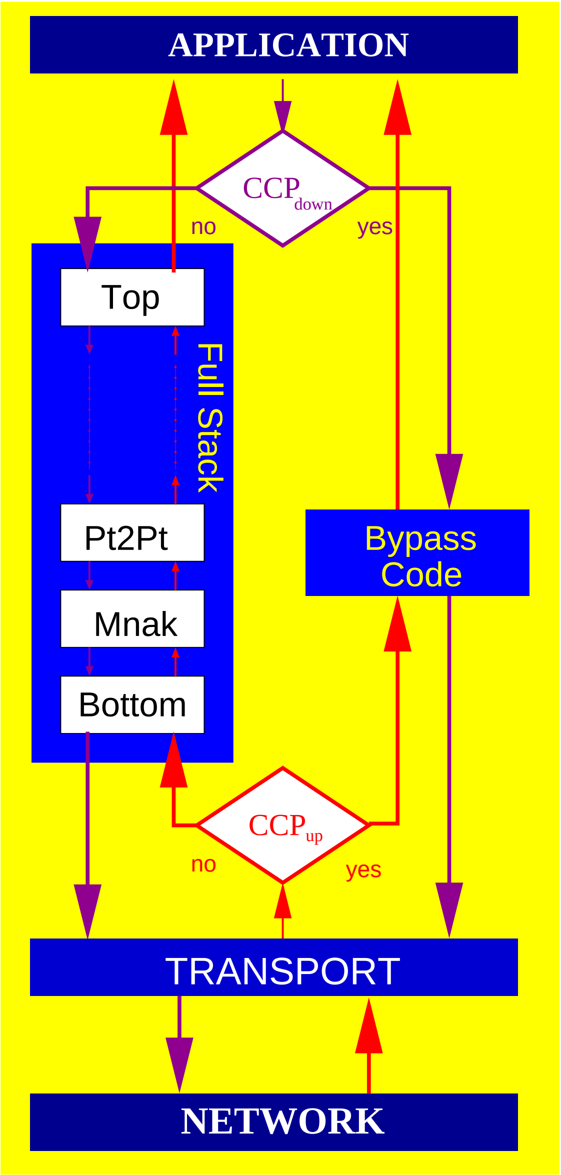
2. Assemble Code Pieces

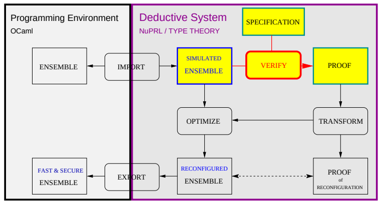
– Case expression for 4 common cases
(send/receive, broadcast/point-to-point)

– Call original event handler in final case

3. Export into OCaml environment

Fully automated,
generated code 3–10 times faster





SPECIFICATIONS AND CORRECTNESS

- **System properties**

  "Messages are received in the same order in which they were sent"

  – Represented in formal mathematics

- **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 can be represented in Nuprl's type theory

Example specifications of a FIFO network

FIFO property

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

Abstract behavioral specification as formal I/O-automaton

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

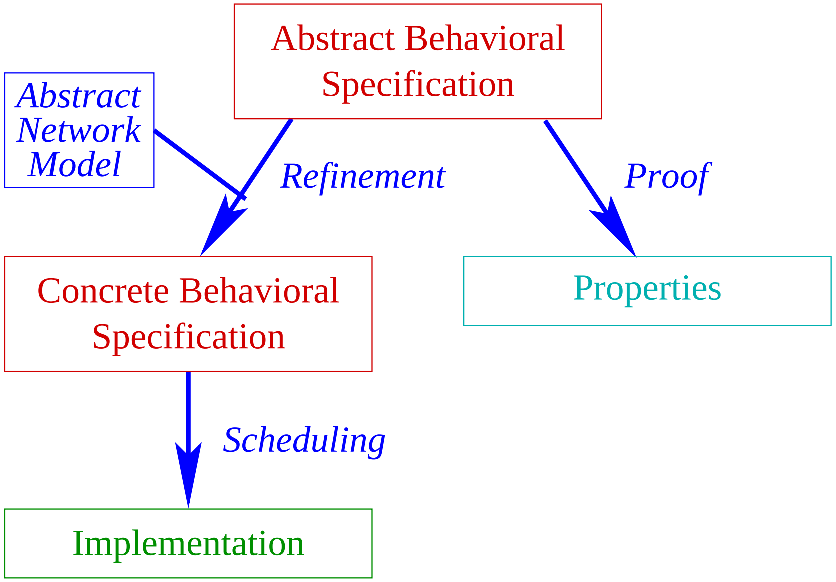
$Deliver(dst : Address; msg : Message)$
condition: in-transit.head() = $\langle dst, msg\rangle$ {in-transit.dequeue()}

Concrete behavioral specification as formal I/O-automaton

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

Verification Methodology

- **Verify IOA-specifications of micro-protocols**
  - Concrete specification $\leftrightarrow$ abstract specification $\rightarrow$ system properties
  - Easy for benign networks $\leadsto$ **subtle bug discovered**
- **Verify protocol stacks by IOA-composition**
  - IOA-Composition preserves all safety properties
- **Weave Aspects** (subtle, still open)
  - Transformations add tolerance against network failures or security attacks
- **Verify code**
  - Micro-protocols $\leftrightarrow$ IOA-specifications
  - Layer composition $\leftrightarrow$ IOA-composition



Lessons learned

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

Future Challenges

The Ensemble case study is just a 'proof of concept'

- **Build better reasoning tools**
  - Build interactive library of formal algorithmic knowledge
  - Increase performance and application range of proof tools
  - Connect more external systems
  - **Improve cooperation between research groups**
- **Learn more from applications**
  - Support reasoning about real-time & embedded systems
    - reason about probabilistic protocols
    - reason about end-to-end quality of service
  - Support programming languages with less clean semantics
  - Invert reasoning direction from verification to synthesis

New research issues: Language-based Security

When can we trust downloaded code?

- **Application scenarios**
  - Programmable mobile devices (cell phones, smart cards, ...)
  - Plug-ins for internet browsers
  - Downloaded code has to be checked for secure behavior
  - Acceptable code does not leak private data to other processes
- **This has nothing to do with**
  - Encryption algorithms or cryptographic protocols
  - Malicious behavior beyond revealing secrets
- **It is about information flow**
  - Simple setting: distinguish high and low confidence levels
  - Secure code will be able to access high-confidence information (h)
  - Modifications of low-confidence data (l) must not depend on h-data

Can we check secure behavior statically by looking at the code?

When can code be considered secure?

- **The decision is not always easy**
  - `l:=h` *obviously insecure, data are copied directly*
  - `if h then l:=0 else l:=1` *boolean information is revealed*
  - `l:=h;...; l:=0` *secure if attacker only sees final result*
  - `if h then (skip; l:=1) else l:=1`
    not secure if attacker can measure execution time
- **It depends on the capabilities of the attacker**
  - Can the attacker see intermediate results?
  - Can the attacker analyze hardware performance (heat, time, ...)?
- **It also depends on the computation environment**
  - How are multi-threaded processes handled by the scheduler?
    What if program 4 runs concurrently with `skip; l:=0` ?
    - Code is secure in sequential setting
    - Not secure when schedule involves shared memory, round robin:
      Final result of `l` is 1, if `h` and otherwise 0

Security checkers need precise security models

SECURITY FORMALIZED

- **Use an abstract programming language**
  - MWL: Multi-threaded while language
  - Allows clearer formulation of security conditions
  - Mechanism can be adapted to, e.g., Java byte code, "automatically"
- **Formalize security relations**
  - Define program states that are undistinguishable for attackers
    - $s_1 =_L s_2 \equiv \forall v{:}\texttt{Lowvar}.\ s_1(v) = s_2(v)$
  - Programs are undistinguishable if they are always "low-equal"
    - $\sim_L \equiv \bigcup\{\sim \mid P{\sim}P' \wedge s{=_L}s' \wedge [P_i, s]{\mapsto}[X, t]$ $\Rightarrow \exists X'{:}\texttt{Com}.\exists t'{:}\texttt{St}.\ [P_i', s']{\mapsto}[X', t'] \wedge t{=_L}t' \wedge X{\sim}X'\}$
- **Introduce type-checking rules for security**
  - 15 inference rules for proving equivalence of programs
  - **Prove that the rules are correct and complete**

Checking Security of Code

Code is acceptable if it can be made secure

- **Make information flow secure by transformation**
  - Insert and instantiate meta-variables into the code
    - Essentially add appropriate skip-operations
  - 11 transformation rules for modifying programs
  - **Prove that the transformed program is equivalent and has secure information flow**
- **Automate application of transformation rules**
  - Automate injection and unification of meta-variables
  - Involves inference rules for lifting and AC1-unification
  - **Prove that automated mechanism is complete**
- **Research challenge (e.g. thesis work)**
  - Validate correctness and completeness with tactical theorem prover
  - Develop proof tactics that can validate similar security concepts