Formal Methods for Software Evolution

(BAA98-10)

Interoperability Results

Robert L. Constable
Cornell University

Outline

- Executive Summary
- Background
- The Vision
- Results
- Applications and detailed results
- Next steps

Common Goals

Substantially improve the software production process

- to accomplish more with software
- to increase reliability of software systems
- to expedite timely production at lower cost
- to manage orderly evolution

Computer Science Approach

Computer science is concerned with automation of intellectual processes.

Formal Methods is the application of CS concepts and methods to the semantics-based automation of systems building.

Interoperability Program Goals

Inject formal methods throughout the software development process

- open the process to analytical scrutiny, to technically informed management and to automation
- support a culture conducive to high quality based on scientific knowledge
- provide more capabilities

  (concepts, tools, paradigms, vision)

Doing this will result in systems in which we have high confidence, and it will lower the cost of building them

Problems

- practitioners are unfamiliar with formal methods
  (e.g. foreign notation, “managers don’t believe in proofs”)
- practitioners are unaware of the value of FM
  (few “advertised adoptions”)
- no clear injection vector or entry points (even for a modicum of formal methods)
- FM community aimed at total solution or late phase applications
- lack of knowledge in machine useable form (formal knowledge)

Problems continued

- can formally design, develop, and verify small function-like code - can even synthesize from specifications
  (scheduling, Gröbner basis, protocols)
- but even for layers of simple functions we do not have these capabilities
  - old methods do not scale
  - need new ideas for reactive and distributed systems

Results from Interoperability Project

- demonstrating interoperability among alternative approaches to design, production and verification of systems
- applying substantial computation to automate code production ("spontaneous adoption")
- creating interoperability mechanisms to build a shared database of formal knowledge and cooperate in proving

Outline

- Summary
- Background
- The Vision
- Results
- Applications and detailed results
- Next steps

Background - economic issues

- reliable systems are “priceless”
- new capabilities open new markets
- we are steadily creating new capabilities

Background - strategic issues

- need a technical base for building large scale systems
- a technology of trust for high confidence systems

old state	:	hard to say anything about networks
current state	:	capable of proving some properties of running protocols
need	:	design and build reliable networks

- automation of software production is an inherently open-ended challenge worthy of sustained investment.

Outline

- The Vision
  - What are we trying to accomplish? (goals)
  - How will we change practice?
  - How is it done now?

The Programming Methodology Ideal

Service specifications
- natural
- analytical
- logical
- formal properties

Abstract annotated code
- guarantees
- invariants
- properties

Production code

Machine code

Environments/Frameworks

Key subsystems

languages (logical/functional/imperative/interactive)

editors/parsers/interfaces

type checkers/provers

interpreters/evaluators

translators/compilers/optimizers/extractors (synthesizers)

model checkers/decision procedures

libraries/databases/version control

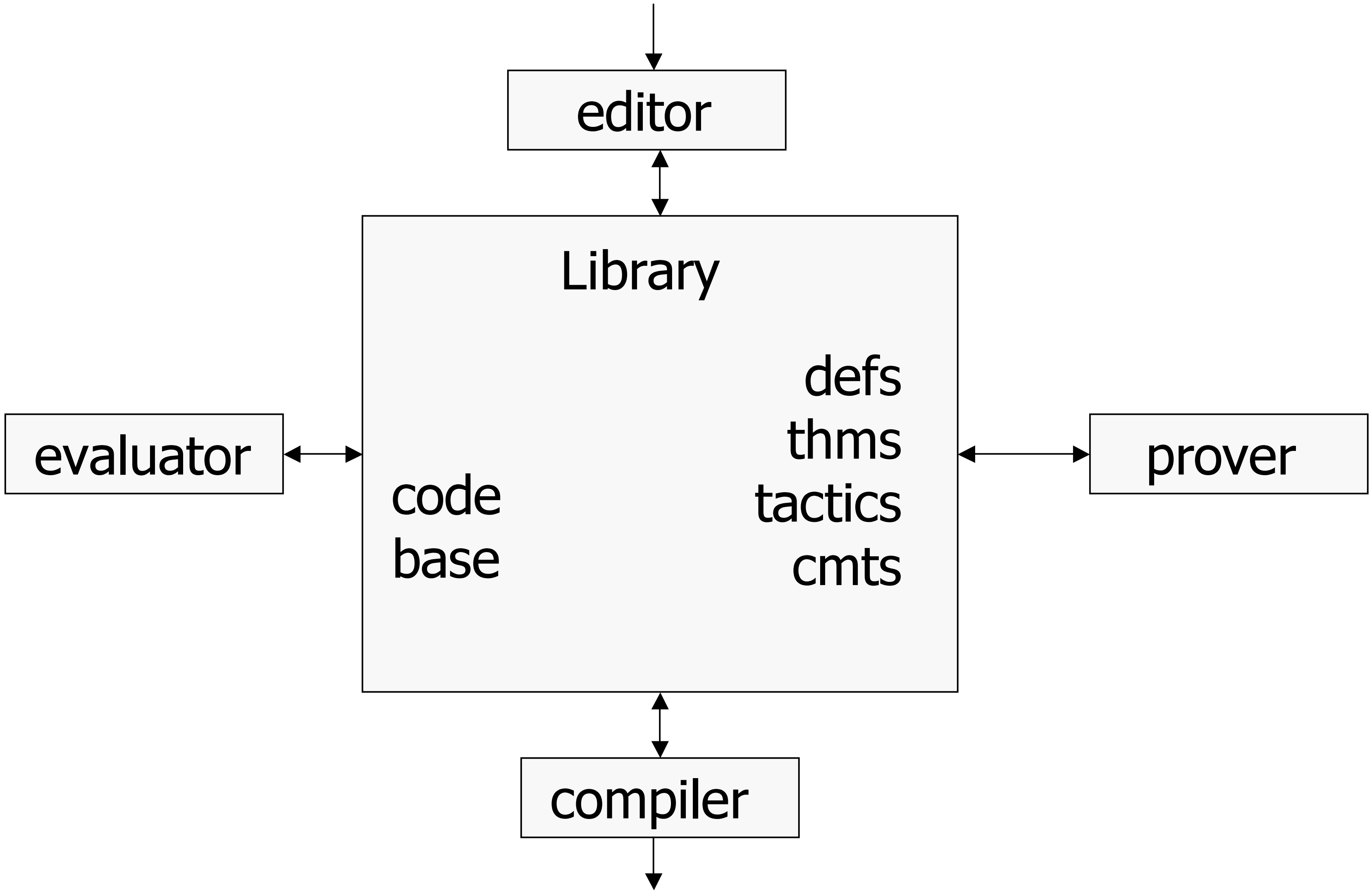
metacontrol/extenders

Environments/Frameworks

An organizing architecture

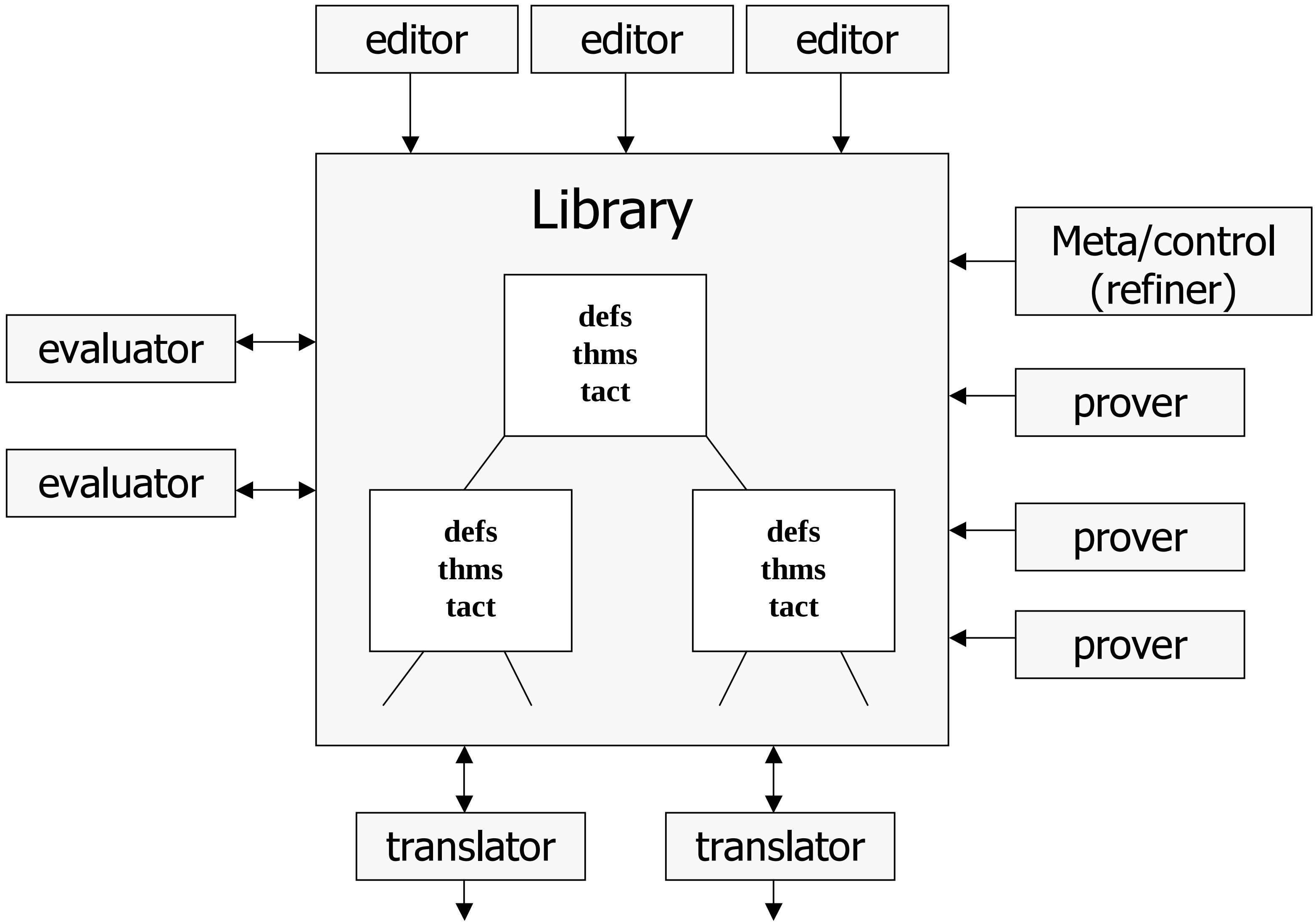
- closed systems vs open systems
- integrating mechanisms

Closed FM Environment Architecture



Typical Environment/Framework/Development System

Consider an open system to integrate all key subsystems.


editor
editor
editor
Library
Meta/control
(refiner)
evaluator
evaluator
defs
thms
tact
defs
thms
tact
defs
thms
tact
prover
prover
prover
translator
translator

Software Support for Programming

This is the backdrop for Interoperability results

Logical Programming Environments (LPE)	- Cornell
Designware/Specware	- Kestrel
Reflective Frameworks	- SRI/Stanford
Verinet	- U Penn
Haskell, Meta-ML, and extended type systems	- OGI

Outline

- Results
  - What have we done to advance our goals?
  - What is new in our approach?
  - How have we overcome old limitations?

Results Summary - Richer Languages

Specware

- already very expressive
- Slang adds dependent types
- MetaSlang is ML-like prog language
  also uses reflection

PVS - subset dependent types

Nuprl/MetaPRL

- intersection types, very dependent types
- class theory (large types or categories), formal modules

Maude - reflection

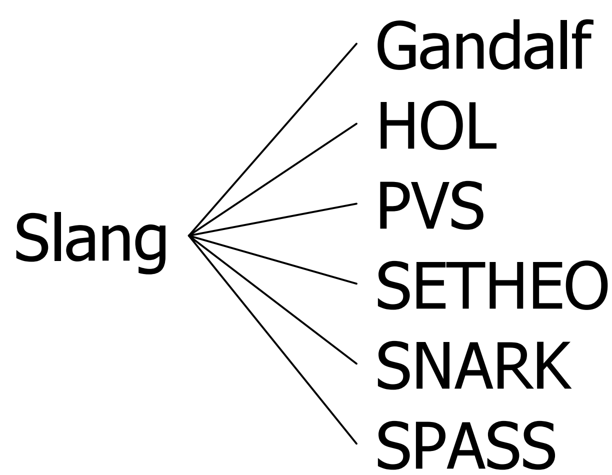
Why do richer languages matter?

- basis of understanding and integration
- support higher level abstractions
- connection to natural specifications is easier
- greater leverage of most effective techniques
    - rewriting – extremely general
    - tactics – extremely general

Results Summary - From monolithic to integrated systems

Kestrel

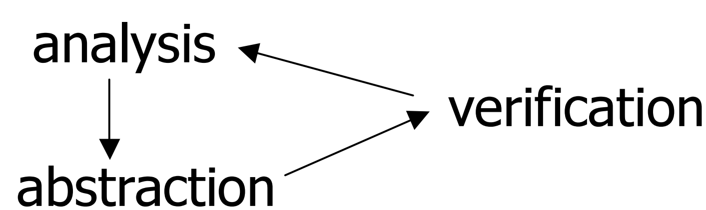
MetaSlang and cooperative proving



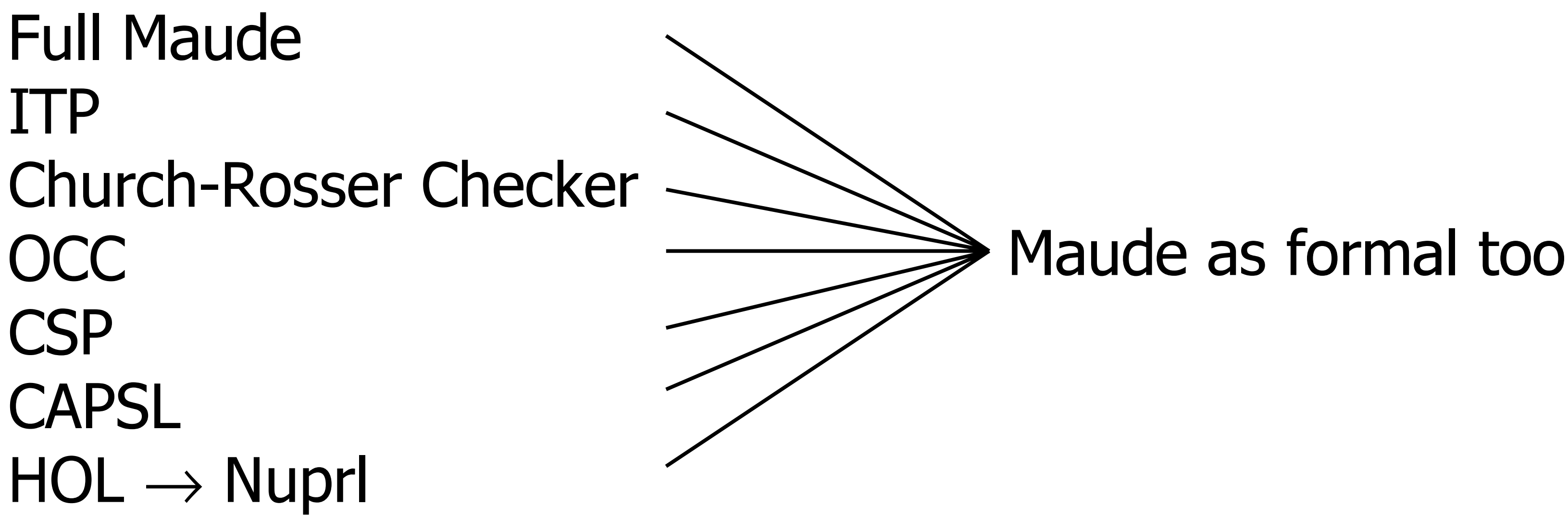
Using MetaSlang

Stanford/SRI

SAL



Results Summary



Where: OCC stands for Open Calculus of Constructions
ITP stands for Inductive Theorem Prover

Results Summary – more integration

SRI/Cornell

HOL $\rightarrow$ Nuprl proof translation

Cornell/Bell Labs

HOL/Nuprl/PVS link
"supernova" proof method

UPenn

HOL/SPIN link

Results Summary

advances in base technology

Maude rewrite engine – speed

MetaPRL distributed tactic prover – speed

Outline

- Applications
  - If successful what difference will it make?
  - What are current successes?

Applications

Kestrel Specware

Boeing : formal spec into CAD design
with OGI : specs for processors

Cornell

Ensemble/Nuprl
- reconfiguration/compression
  - Nortel Networks
- verification
  - BBN
  - NASA

SRI/Stanford

with Penn, active networks protocols
CAPSL for security protocols

Upenn

Verinet

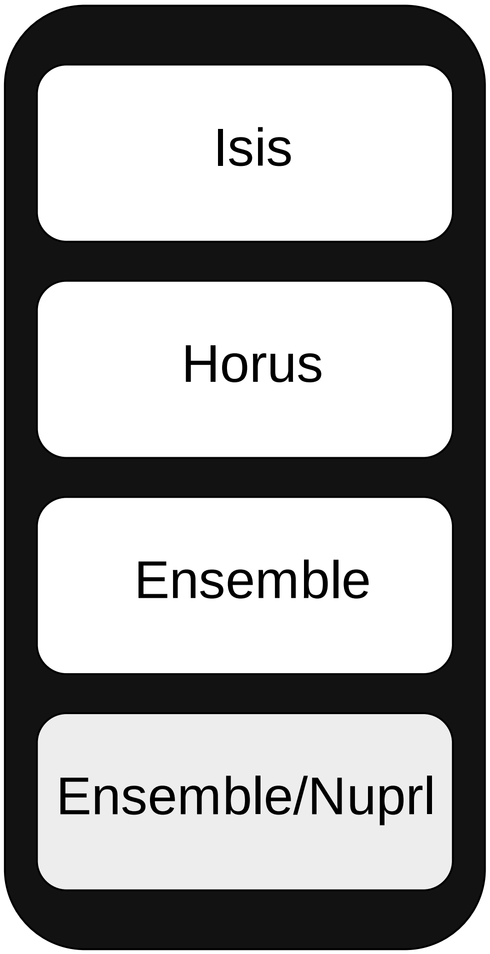
- RIP and AODV

Outline

- Ensemble / Nuprl fastpath
    - Ensemble Architecture
    - Compressing stacks
    - Verification of stacks

Group Communication Systems

Reliable and secure networking in safety-critical applications



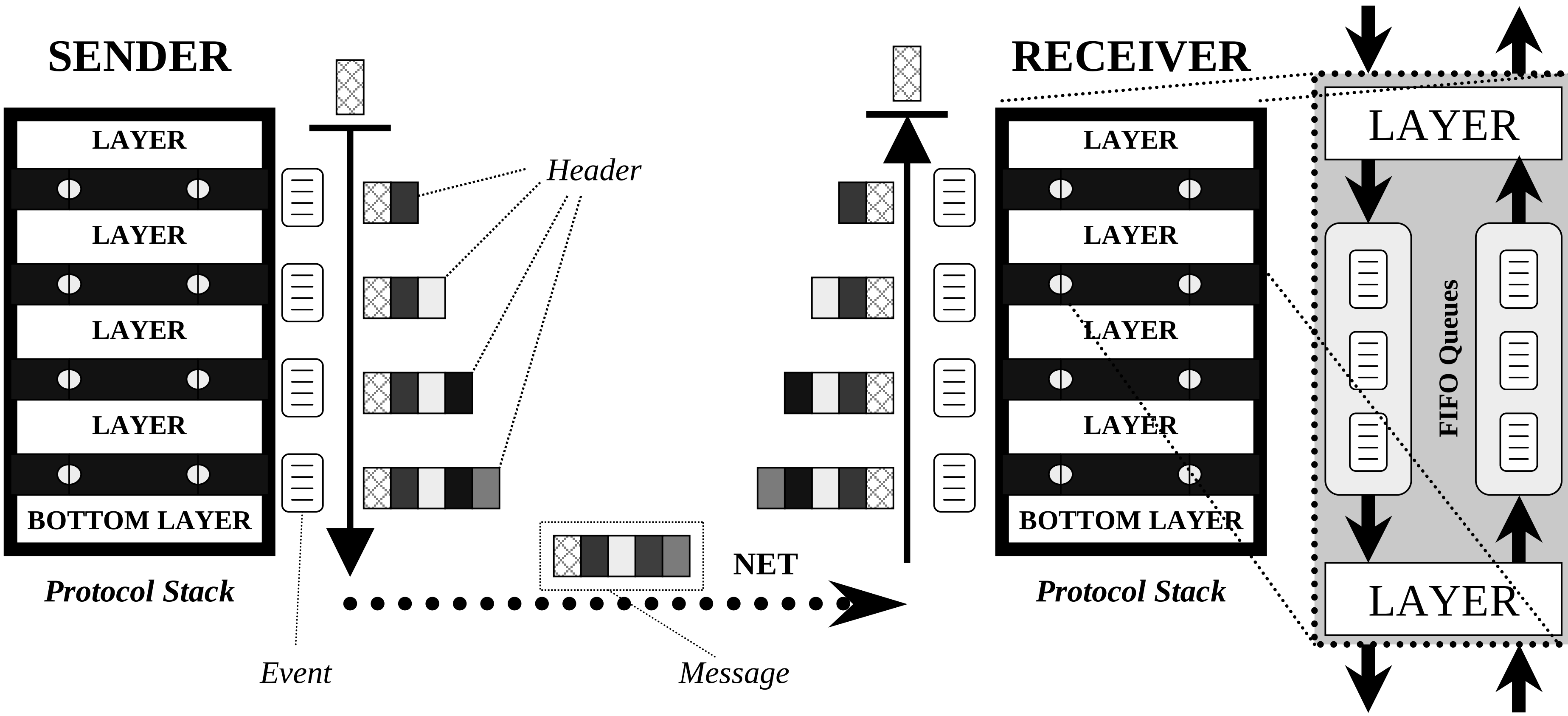
Technology for securing networked applications
- widely used: NY Stock Exchange, French Air Traffic Control …

Added flexibility through protocol stacking
- reconfigurable to specific needs of applications

Reference implementation in Ocaml
- small protocol layers, easy to check and modify
- portable to a variety of platforms
- highest performance due to fast-track reconfiguration

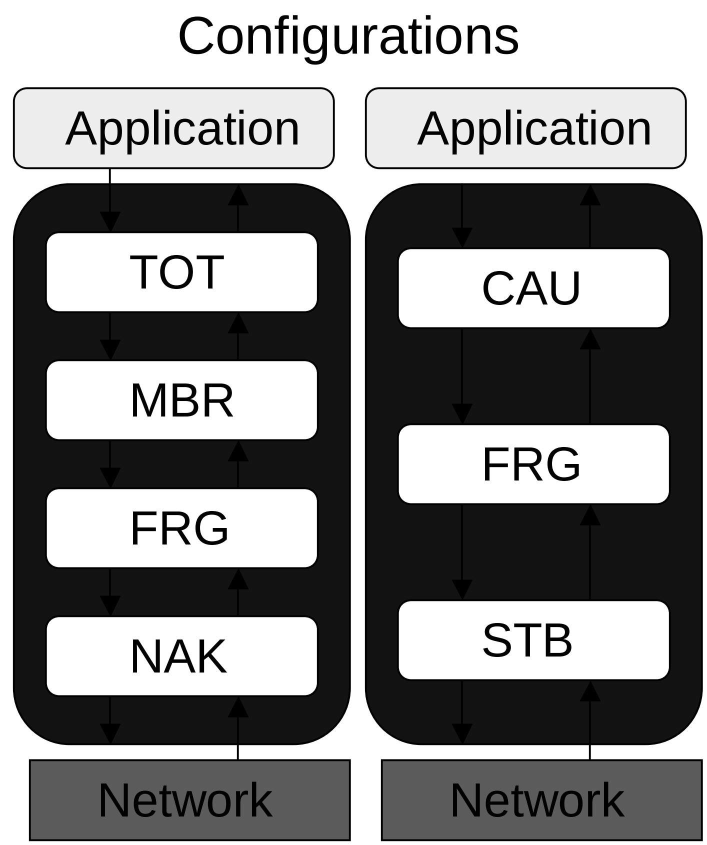
Logical development tools for network security
- verification of critical properties (beyond type checking)
- formal documentation / logical debugging
- automated and verified fast-track reconfiguration

Architecture of Ensemble

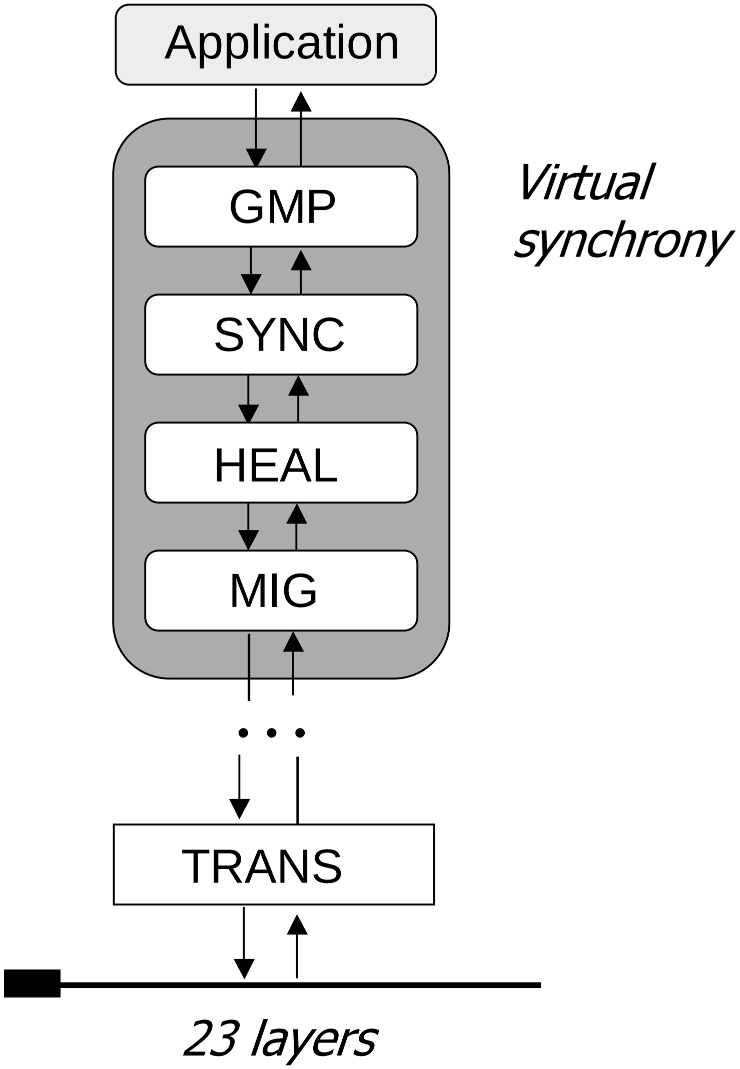


Cost of modularity

- Poor performance
  - redundant code
  - abstraction enforcement
- Difficult to verify complete systems
  - combinatorial number of configurations



Ensemble Implementation

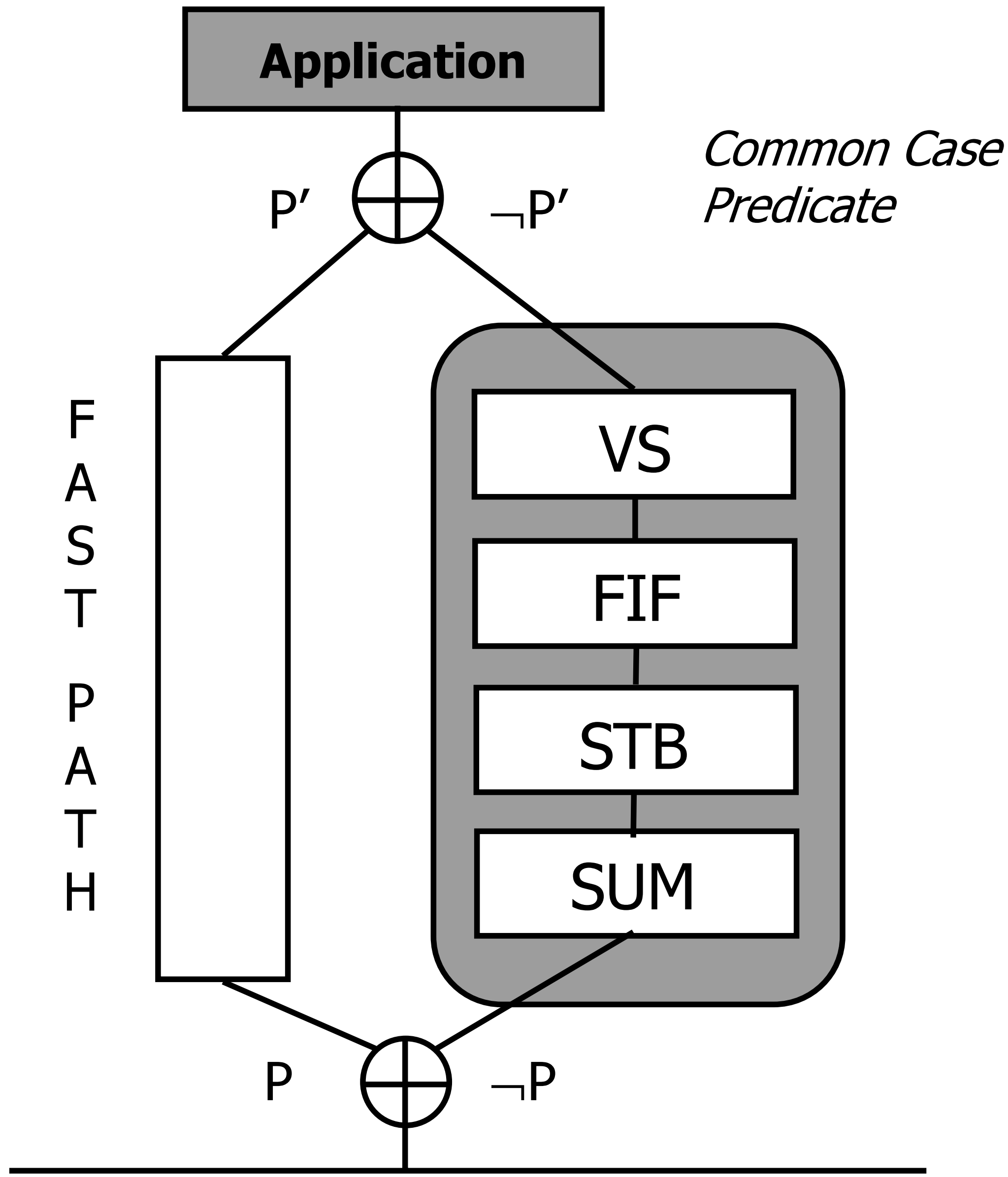


- Layered protocol stacks
- Each layer implements a property
- Protocols are
  - small (-300 lines ML)
  - roughly orthogonal
- Configuration is application-specific
- About 50 layers; thousands of protocols

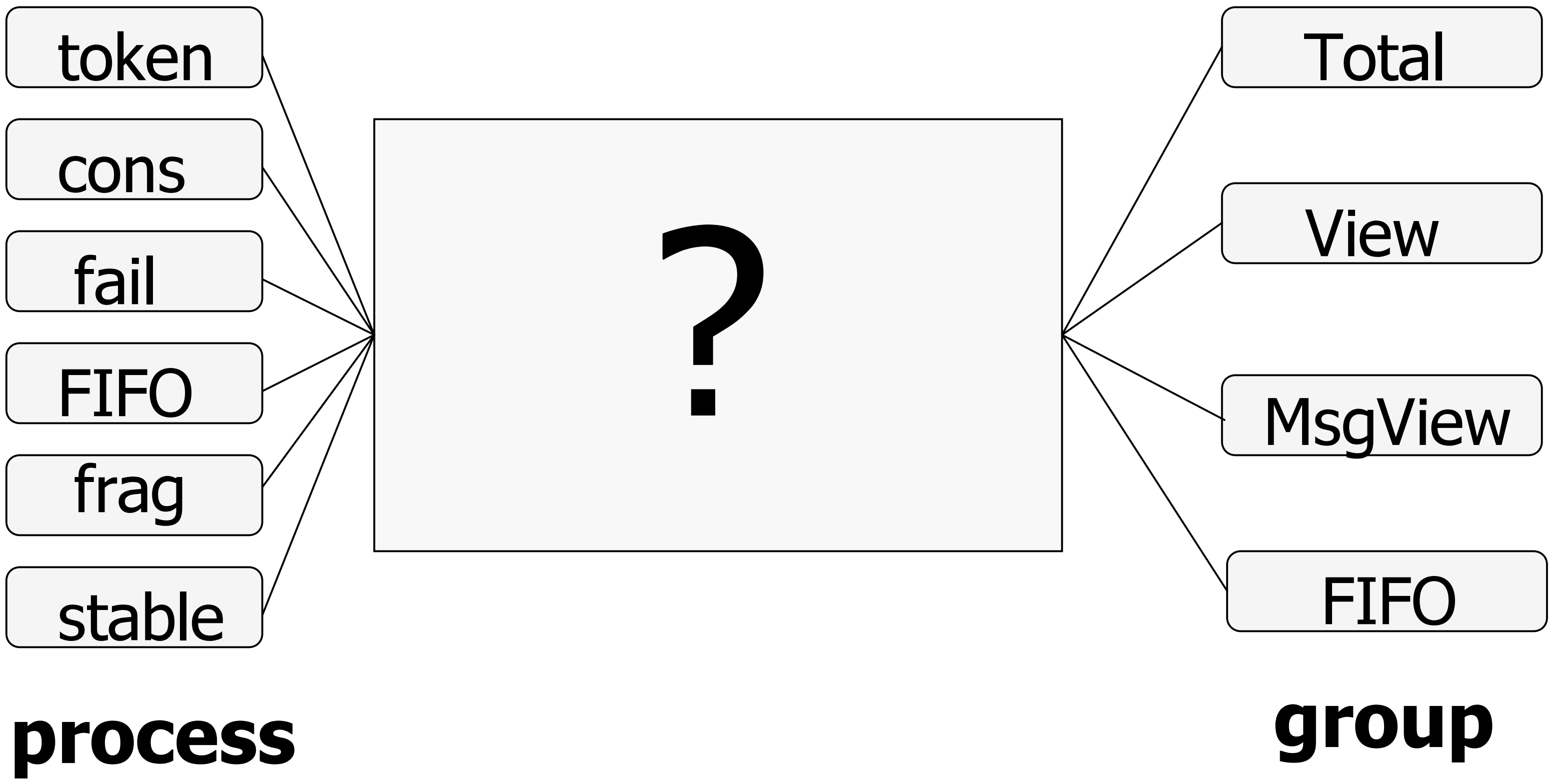
Fast path

Extract common path from the protocol

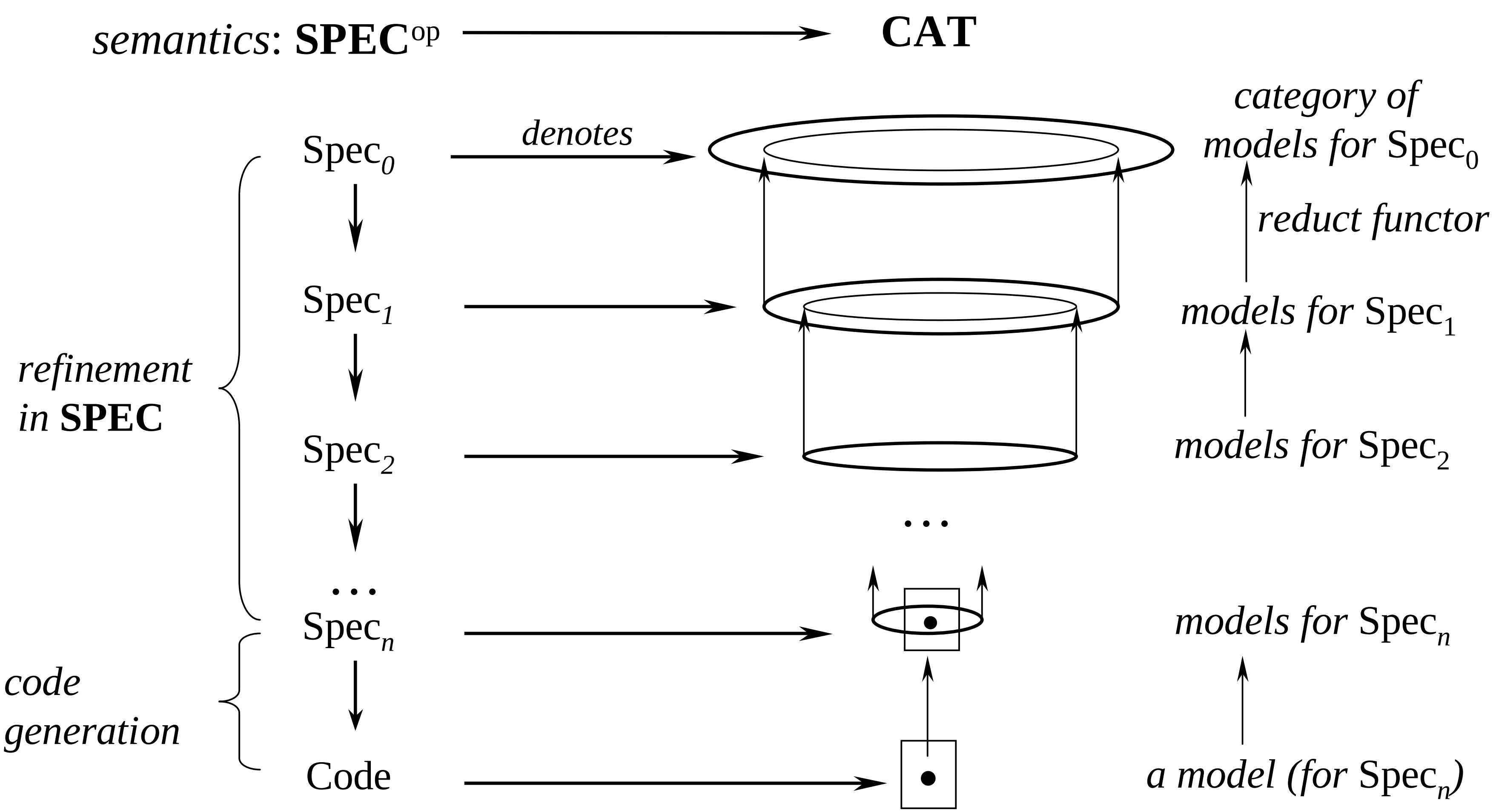
Speedups of x2-x50



Verification - How do we bridge the gap?

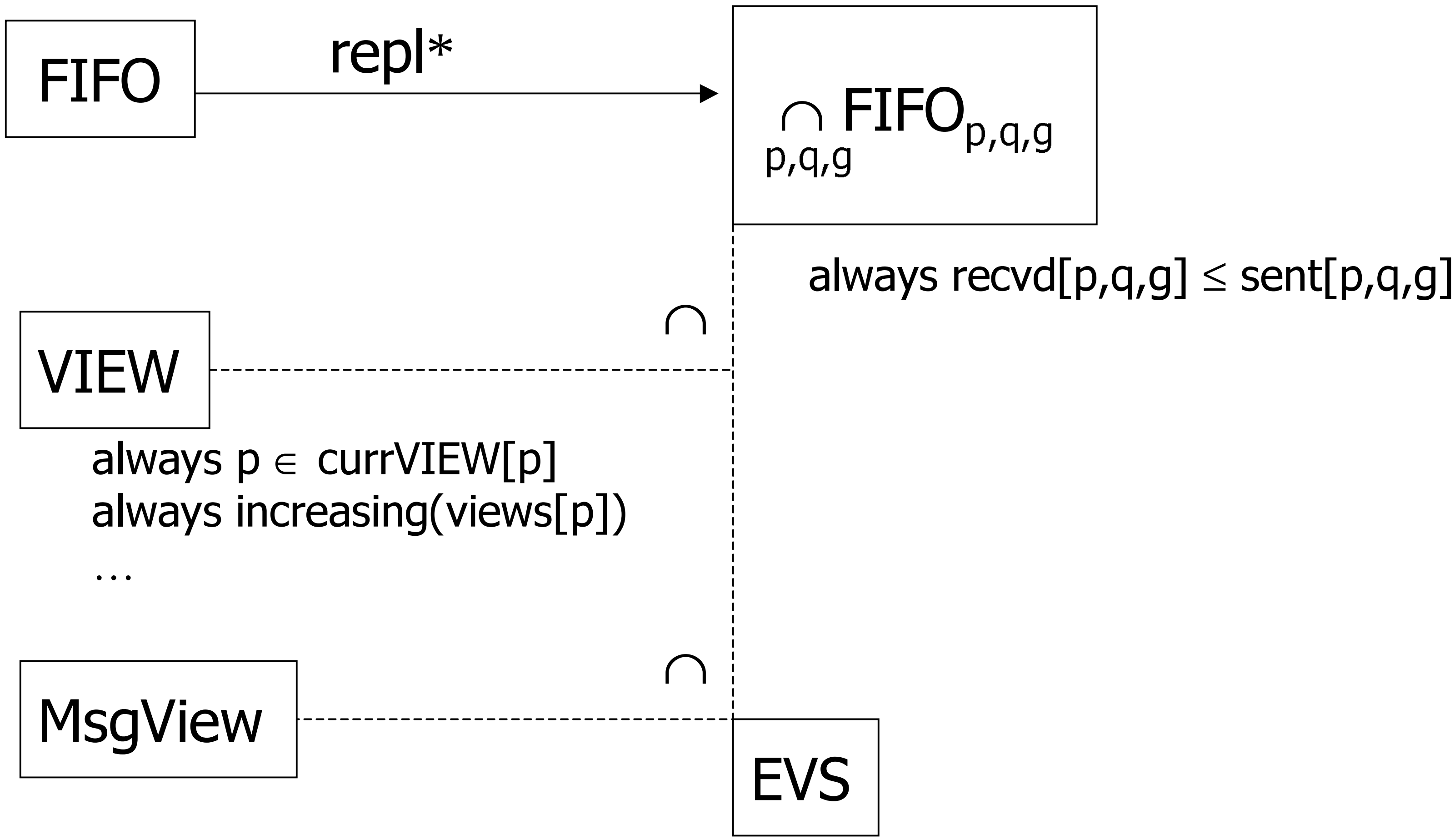


Software Development by Refinement



Code generation via a logic morphism from **SPEC** *to the logic of a programming language*

FIFO-EVS-View in Nuprl



*

Repl (I,f,ls,A) == $\cap$i$\in$I. Rename (subscript(ls,Fi),A)

Subscript (ls,q) == λl. If ls $\leq$ l then l[q] else l

Operations on records $\{x_1{:}T_1; \ldots x_n{:}T_n\}$

- Intersection $\cap$
  - Union of fields: $x_1 = x_2 \Rightarrow \{x_1{:}T_1\} \cap \{x_2{:}T_2\} = \{x_1{:}T_1; x_2{:}T_2\}$
  - Intersection of types in joint fields: $\{x_1{:}T_1\} \cap \{x_1{:}T_2\} = \{x_1{:}T_1 \cap T_2\}$

- Relabeling ρ
  - Renaming function ρ : Label $\rightarrow$ label
  - $\overline{\rho}(\{ \ldots x_i{:}T_i \ldots\}) = \{\ldots x_j : \cap\{T_i \mid \rho(i)=j\} \ldots \}$ $\quad$ $\{\cap\emptyset = \mathrm{Top})$

- Refinement is Subtyping $\sqsubseteq$
  - $r_1 = \{x_1{:}T_1; \ldots x_n{:}T_n\} \sqsubseteq r_2 = \{y_1{:}S_1; \ldots y_m{:}S_m\}$
    if r1 has more fields and finer types on joint fields

What is a record?

Nuprl answer:

Example: $\{x:\mathbf{N};\ y:Q\ list;\ f: \mathbf{N} \rightarrow \mathbf{B}\}$

Generally $\{x_1:A_1; \ldots ; x_m : A_m\}$

Define a signature $A: \{x_1, \ldots , x_m\} \rightarrow Type$

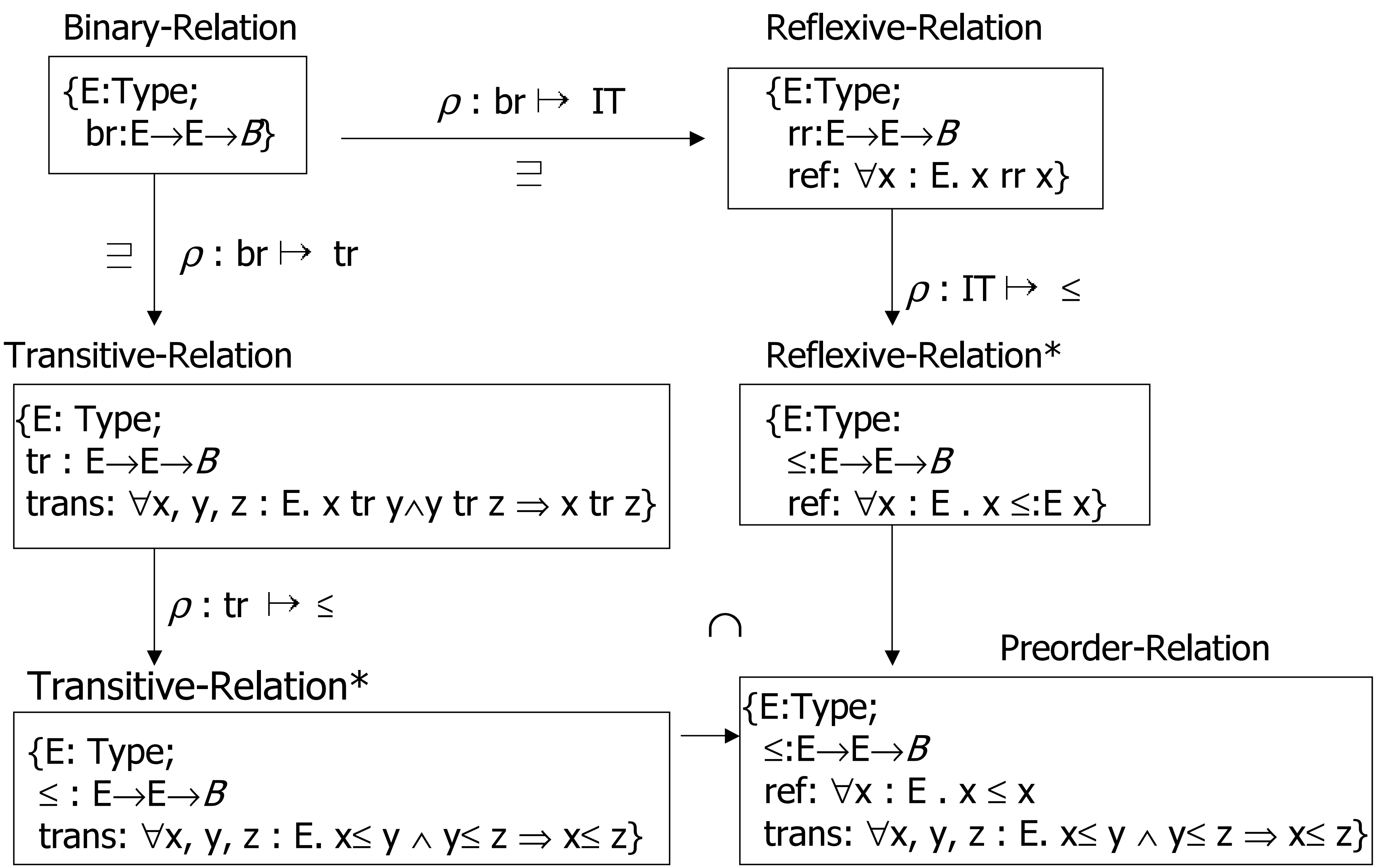
e.g., $A(x) = \mathbf{N}$, $A(y) = Q\ list$, $A(f) = \mathbf{N} \rightarrow \mathbf{B}$

Define a record as a dependent function, an element of

$i: \{x_1, \ldots , x_m\} \rightarrow A(i)$ such as

$r(x) = 0$ $r(y) = nil$ $r(f) = \lambda(x.true)$.

Calculating the Colimit in Nuprl



Main point

constructive type theory provides good object-oriented methods,

especially classes, subtyping, inheritance.

these are very useful in verifications

especially **proof reuse**

modular decomposition

there is a great deal to say

- Jason Hickey and Mark Bickford Verification work
- applications to algebra
- design of MetaPRL system

See Nuprl Web page

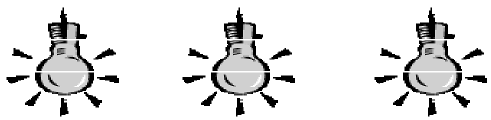
- Hickey

Outline

- Next steps
  - How can we move closer?
  - What milestones can we achieve?

Progress

Basic Discoveries (CS, Math, Logic)

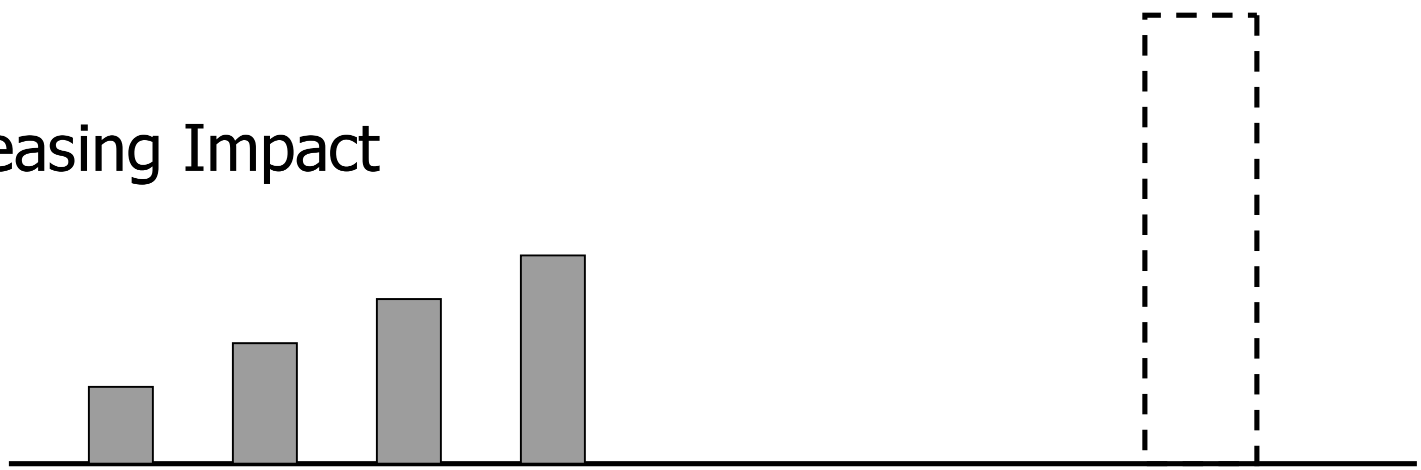
 …

I_1 I_2 I_3 …

Increasing Capabilities (15-25 years down stream)

$C_1, C_2, C_3, \ldots, C_n, \ldots$

Increasing Impact



Next steps

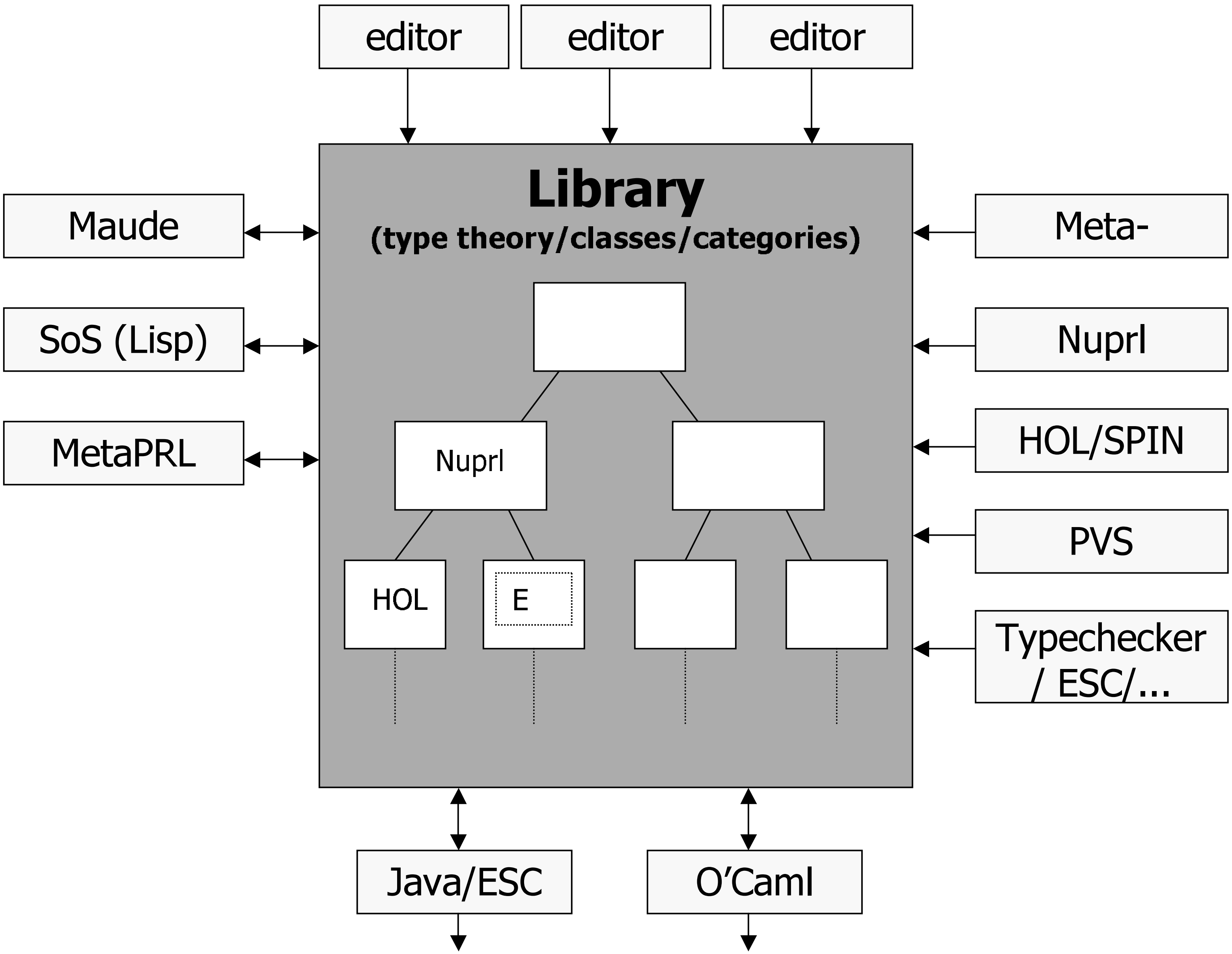
Applications on the horizon

reliable protocols in use
supporting massive user base

from compression to reconfiguration

reactive systems

beyond extended type checking



editor
editor
editor
Library
(type theory/classes/categories)
Maude
SoS (Lisp)
MetaPRL
Meta-
Nuprl
HOL/SPIN
PVS
Typechecker / ESC/...
Nuprl
HOL
E
Java/ESC
O'Caml

Advances toward a technology of trust

delivering new architectures

new capabilities depend on:

- feedback to the system architectures
- sharing ideas and technology
- pushing the envelope

Capitalizing on Speed

Moore’s law has dramatic effect,

must keep pushing the envelope

can automate more

Capitalizing on Knowledge

Integrating formal and informal knowledge

Storing knowledge in a shared library

Internal Milestones

We can do more for each other beyond publishing ideas

Shared verification of a subsystem or procedure is possible

- constraint solver
- arithmetic decision procedure
- extended type checker

Shared components are possible, e.g. a Library.

Sociology

critical mass of talent will come to the area with a proper research environment

capabilities are fusing into a technology for building high confidence systems

Summary

We can see a clear path to providing new semantics-based automation capabilities in the system development process.

There is more theory to apply but we must also push the theory further to reap the benefits of vastly increased computing power.

Conclusion

A realistic and robust vision for formal methods is emerging. It will focus US efforts and accelerate the emergence of a new enabling technology of trust.