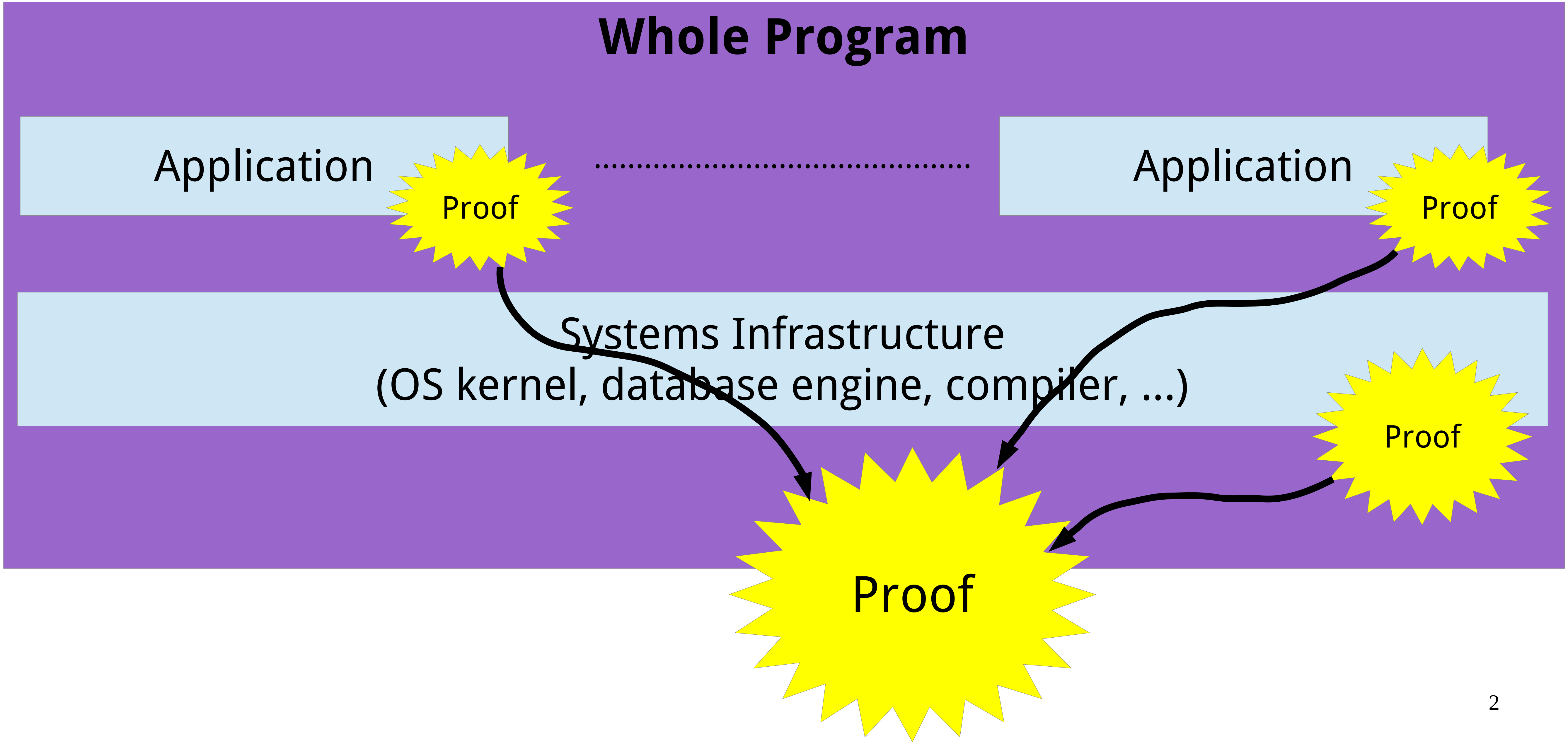
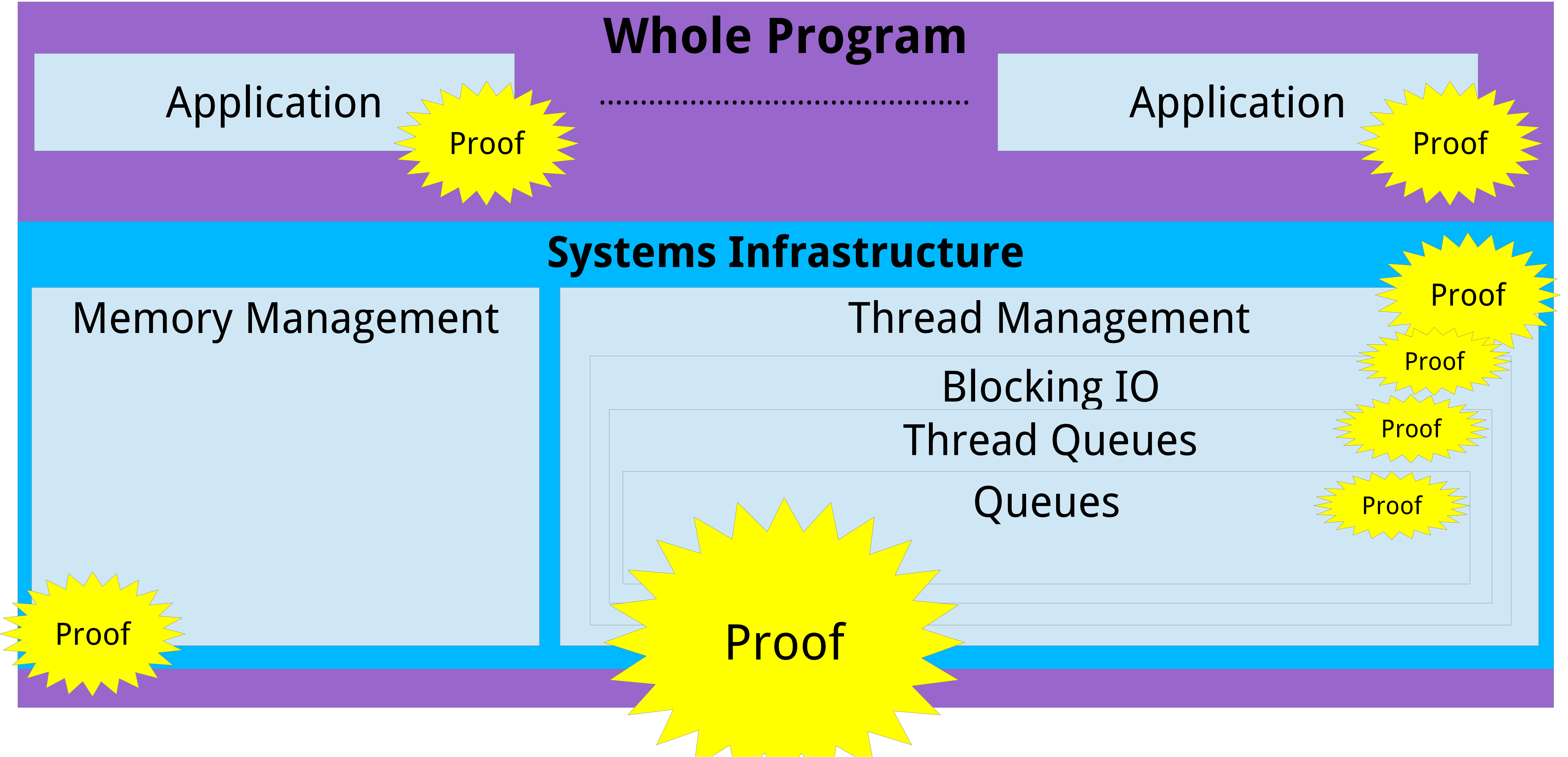
From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification

Adam Chlipala – MIT CSAIL
POPL 2015
January 17, 2015


Whole Program
Application
Proof
Application
Proof
Systems Infrastructure
(OS kernel, database engine, compiler, ...)
Proof
Proof

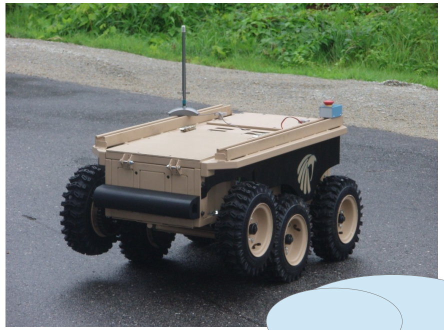

Whole Program
Application
Proof
Application
Proof
Systems Infrastructure
Proof
Memory Management
Thread Management
Proof
Blocking IO
Proof
Thread Queues
Proof
Queues
Proof
Proof

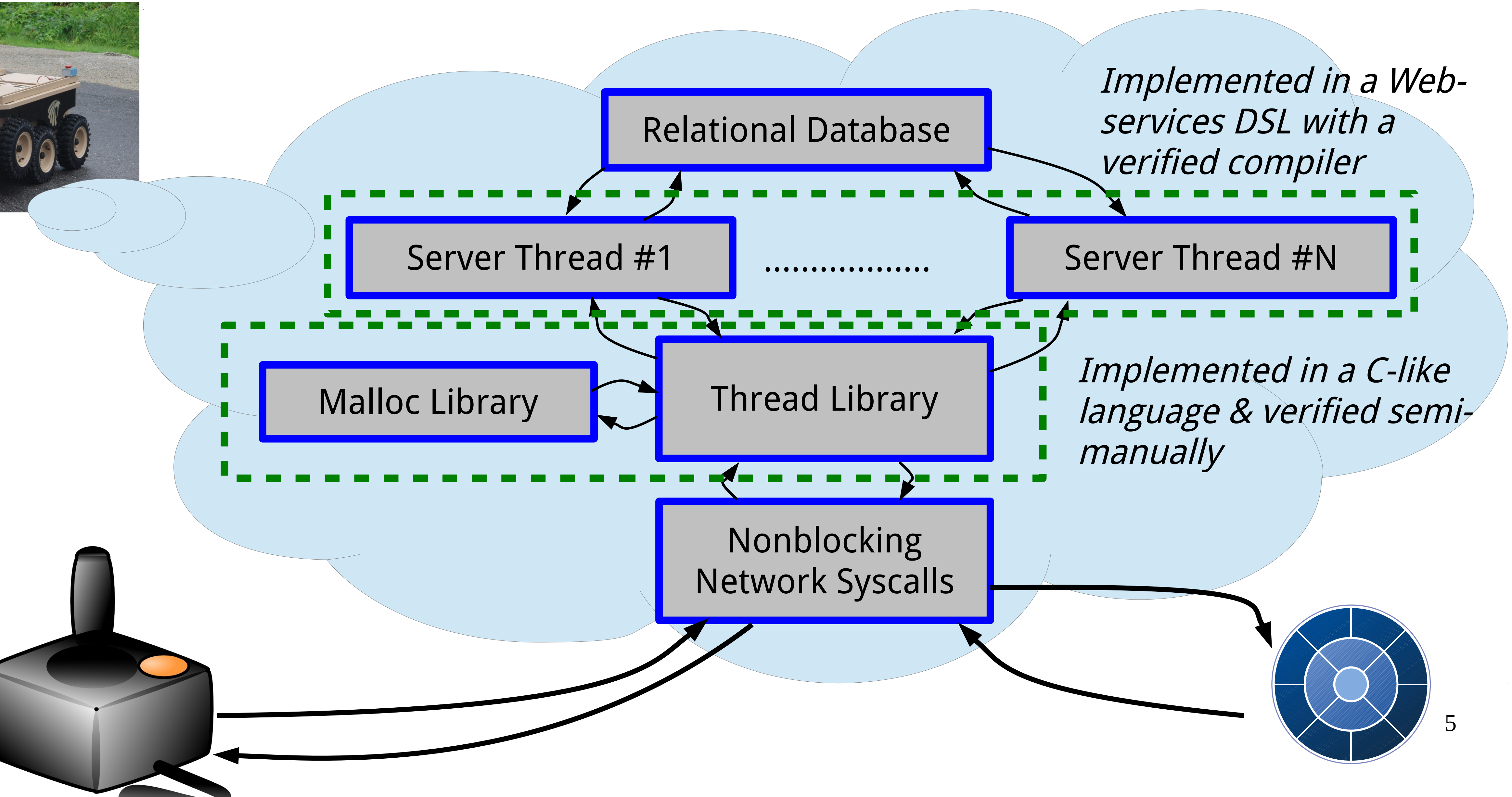
Surprisingly few systems-verification projects have used their results to connect to proofs of applications that someone is actually running.

The modular style also doesn't seem to have been used previously to verify infrastructure serious enough to connect to real applications.

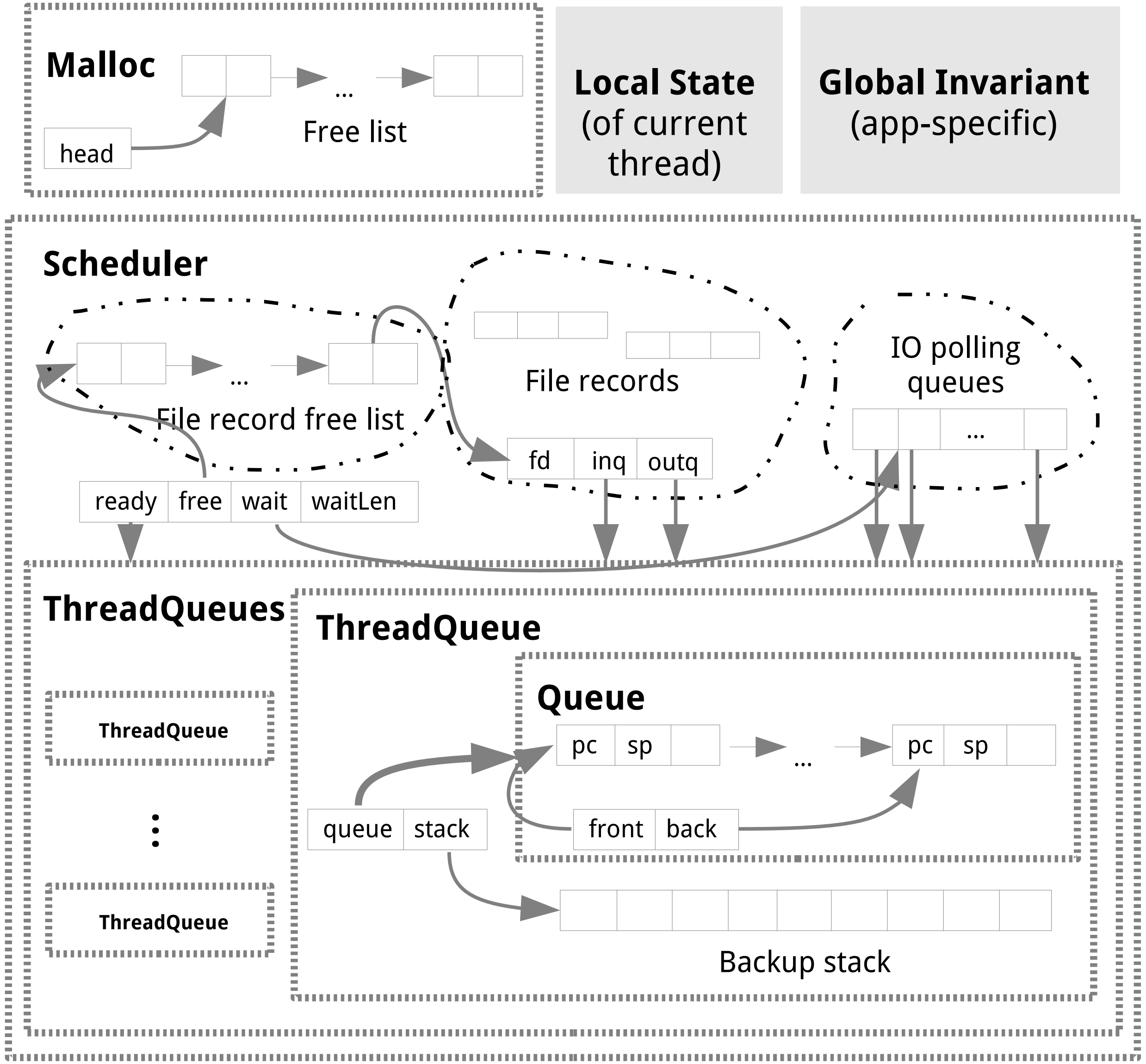
This talk: a case study doing all of the above inside **Coq**,
using the **Bedrock** framework

Deployed on Autonomous Vehicles



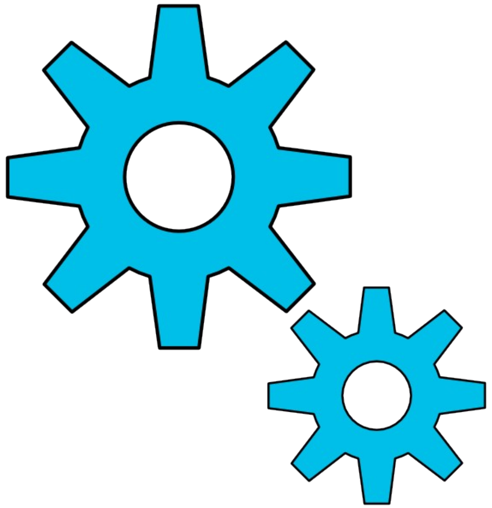


Example Module Decomposition: Nested Threading Abstractions

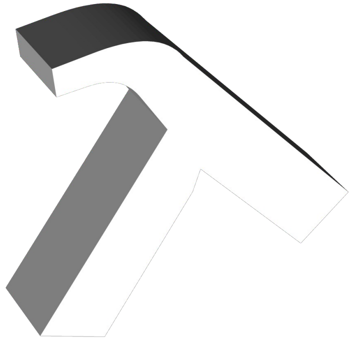


Plan for Rest of Talk

- Basic stage-setting about this style of verification
- Fundamentals of the Bedrock framework
- Adapting to interface with unverified code in a principled way
- Three “design patterns” of general interest
  - Recursive definitions of higher-order, stateful predicates
  - Good formal interfaces for threading components
  - Modular verification of DSL compilers
- Code & performance



<u>Highly Automated:</u>
Tools should fill in most of the boring details of proofs.



<u>Higher-Order:</u>
Can use higher-order logic to state elegant & general specs.

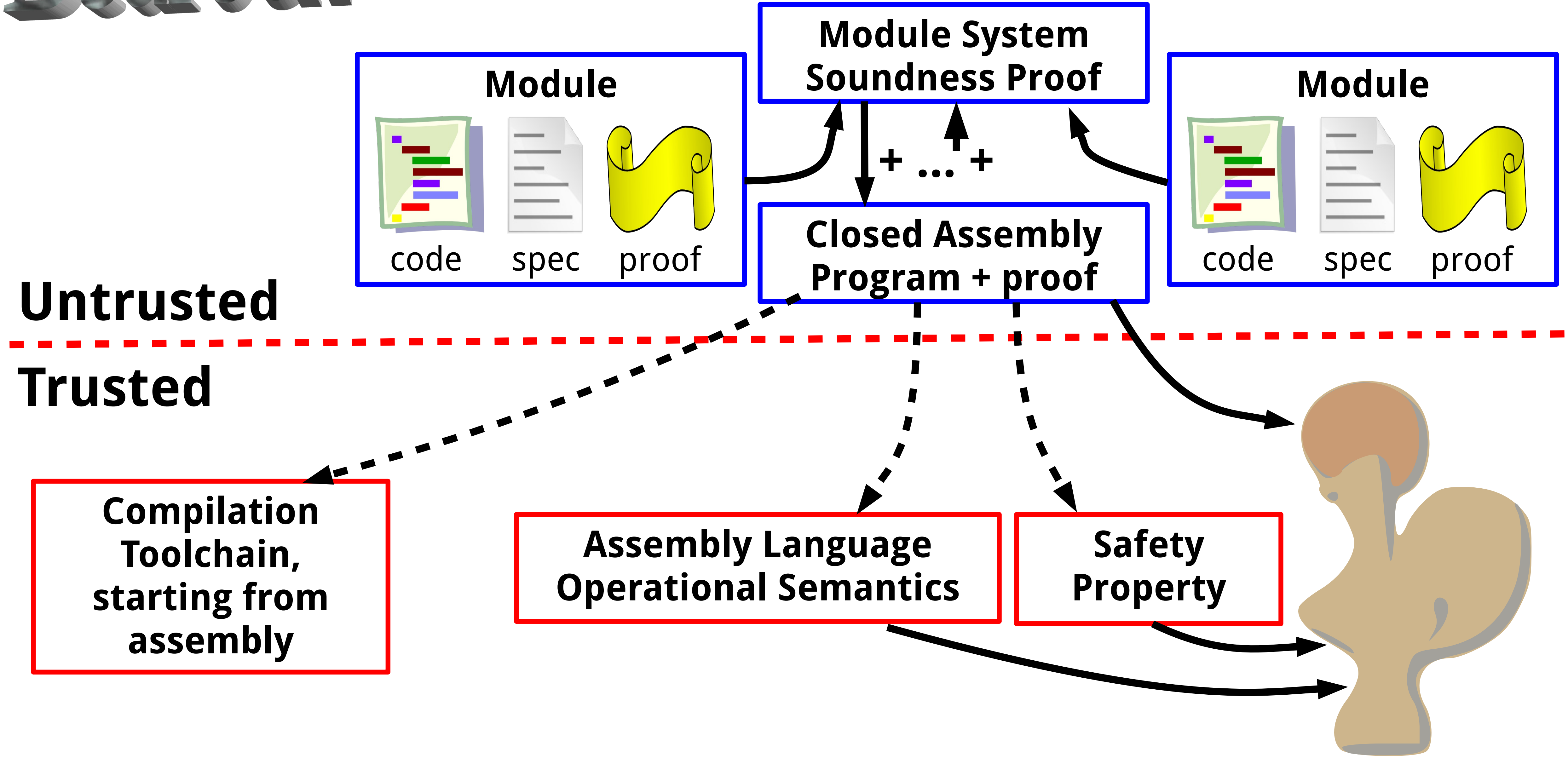
The approach in this case study starts from separation-logic tools of past work (PLDI 2011, ITP 2014) and adds a few new tricks, while also applying them on a much larger scale than before.

<u>Foundational:</u>
Verification leads to a proof checked by a general-purpose proof assistant (*Coq*, in this case).
Trusted code base includes just the *proof assistant*, operational semantics of *assembly language*, and a *specification* for the whole program.

Inventory of Corners Cut

- Proving functional correctness for systems code, but only data-structure shape invariants for application.
- Performance of verified server is OK, but it's not hard to do better. (Some parts simplified to make proofs easier.)
- Level of proof automation varies across components. Some proofs are fairly manual. (Overall proof-to-program ratio [~5:1] remains well below those reported in related projects [>= 20:1].)

What Should We Trust?



Bedrock version of linked list length

Specification

```
Definition lengthS : spec := SPEC("x") reserving 1
  Al ls,
  PRE[V] sll ls (V "x")
  POST[R] [| R = length ls |] * sll ls (V "x").
```

Implementation

```
bfunction "length"("x", "n") [lengthS]
  "n" <- 0;;
  [Al ls,
    PRE[V] sll ls (V "x")
    POST[R] [| R = V "n" ^+ length ls |] * sll ls (V "x")]
  While ("x" <> 0) {
    "n" <- "n" + 1;;
    "x" <-* "x" + 4
  };;
  Return "n"
end.
```

Loop invariant

This is all Coq code, taking advantage of Coq's extensible parser!

Proof

```
Theorem sllMOk : moduleOk sllM.
Proof.
  vcgen; abstract (sep hints; finish).
Qed.
```

Now Application Code Looks Like:

```
RosCommand "setParam"(!string $"caller_id",
                      !string $"key", !any $$"value")
Do
 Delete "params" Where ("key" = $"key");;
 Insert "params" ($"key", $"value");;

 From "paramSubscribers" Where ("key" = $"key") Do
  Callback "paramSubscribers"#"subscriber_api"
  Command "paramUpdate"(!string "/master", !string $"key", $"value");;

 Response Success
  Message "Parameter set."
  Body ignore
 end
end
```

Program defines **remote procedure call** entry points.

Manipulates **relational database** (simple updates and queries).

Callbacks trigger calls to similar functions on other nodes.

Notations are hiding underlying DSL features for **XML pattern-matching and generation**.
(Network communication is all via XML over HTTP.)

```
Theorem Wf : wf ts pr buf_size outbuf_size.
Proof.
  wf.
Qed.
```

Automatic well-formedness proof establishes assumption of *verified compiler*.

Connecting with Untrusted Support Code

We assume that the following *nonblocking system calls* exist, abstracting a TCP/IP network interface:

```
// Standard TCP socket operations
fd_t listen(int port);
fd_t accept(fd_t sock);
int read(fd_t sock, void *buf, int n_bytes);
int write(fd_t sock, void *buf, int n_bytes);
void close(fd_t sock);

// epoll-style IO event notification
res_t declare(fd_t sock, bool isWrite);
res_t wait(bool isBlocking);
```

Add System Calls to Operational Semantics

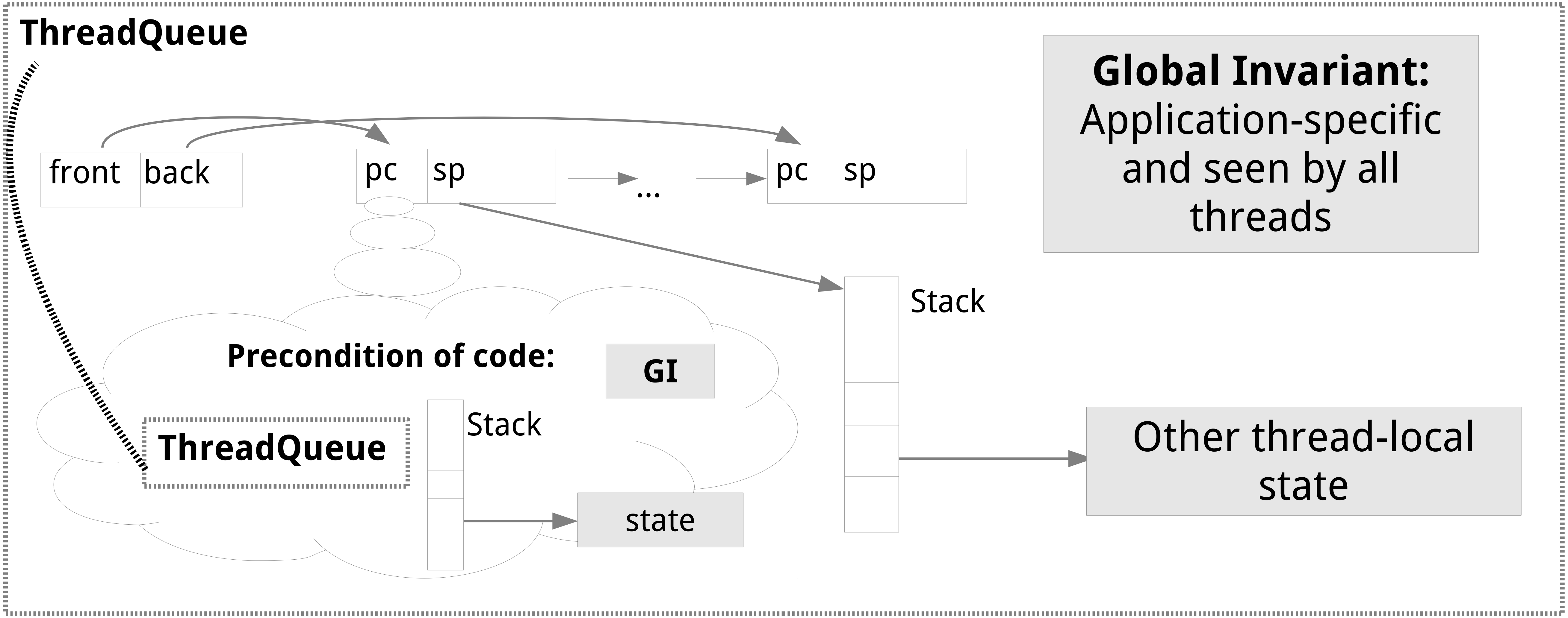
$$\frac{\begin{array}{c}[r.Sp,\ r.Sp + 16) \in ValidMem \\ m[r.Sp + 8] = buf \\ m[r.Sp + 12] = len \\ [buf,\ buf + len) \in ValidMem \\ r'.Sp = r.Sp \\ \forall a.\ a \notin [buf,\ buf + len) \rightarrow m'[a] = m[a]\end{array}}{(m,\ read,\ r) \rightarrow (m',\ r.Rp,\ r')}$$

State: (memory, program counter, registers)

Each op. sem. rule has a corresponding **separation logic** rule, proved sound w.r.t. the original. E.g.:

$$\{buf \overset{?}{\rightarrow} len\}$$
$$read(sock,\ buf,\ len)$$
$$\{buf \overset{?}{\rightarrow} len\}$$

Recursive, Higher-Order, Stateful Predicates



A Simpler Case

Watch out for recursive predicate definitions that are **inconsistent** when interpreted naively! E.g.:

$P(p) \stackrel{def}{=} \exists q.\ p \rightarrow 42,\ q \wedge \{P(p)\}\ q\ \{P(p)\}$

“p po
po
and
e

Allow code modules to come packaged with **named predicate definitions**, which can be looked up within specs. Now the funky reasoning only applies when we reason about the higher-order parts of a definition.

$P(p) \stackrel{def}{=} \mu\rho.\ \exists q.\ p \rightarrow 42,$

Bravely going ahead with *general-recursive predicates*! Requires restrictive **side conditions** throughout proofs, to avoid inconsistency.

$P(p) \stackrel{def}{=} \exists q.\ p \rightarrow 42,$ … ecOf(“P”)(p)}

See paper for: a cute trick to encode named predicates as named functions in the code to verify.

Specifying a Thread Stack

ThreadQueue module
<u>Parameters</u>: a set of **worlds**, a **global invariant** ginv in terms of it, and an **evolution relation** $\preccurlyeq$

$$\forall w.\ \{tq(w, q) \ast ginv(w, q)\}$$
$$yield(q)$$
$$\{\exists w'.\ w \preccurlyeq w' \land tq(w', q) \ast ginv(w', q)\}$$

ThreadQueues module
<u>Parameters</u>: like above, except argument to ginv is *set of queues*, not just one queue

$$\forall w, Q.\ \{tqs(w, Q) \ast ginv(w, Q) \land inq \in Q \land outq \in Q\}$$
$$yield(inq, outq)$$
$$\{\exists w', Q'.\ w \preccurlyeq w' \land Q \subseteq Q' \land tqs(w', Q') \ast ginv(w', Q')\}$$

Use **ThreadQueue** as a submodule by *deriving its parameters from these*!

Scheduler module
<u>Parameters</u>: like above, except argument to ginv is hardcoded as *set of open files*

$$\forall F.\ \{sched(F) \ast ginv(F)\}\ yield()\ \{\exists F'.\ F \subseteq F' \land sched(F') \ast ginv(F')\}$$

Example verification of a client application (echo server)

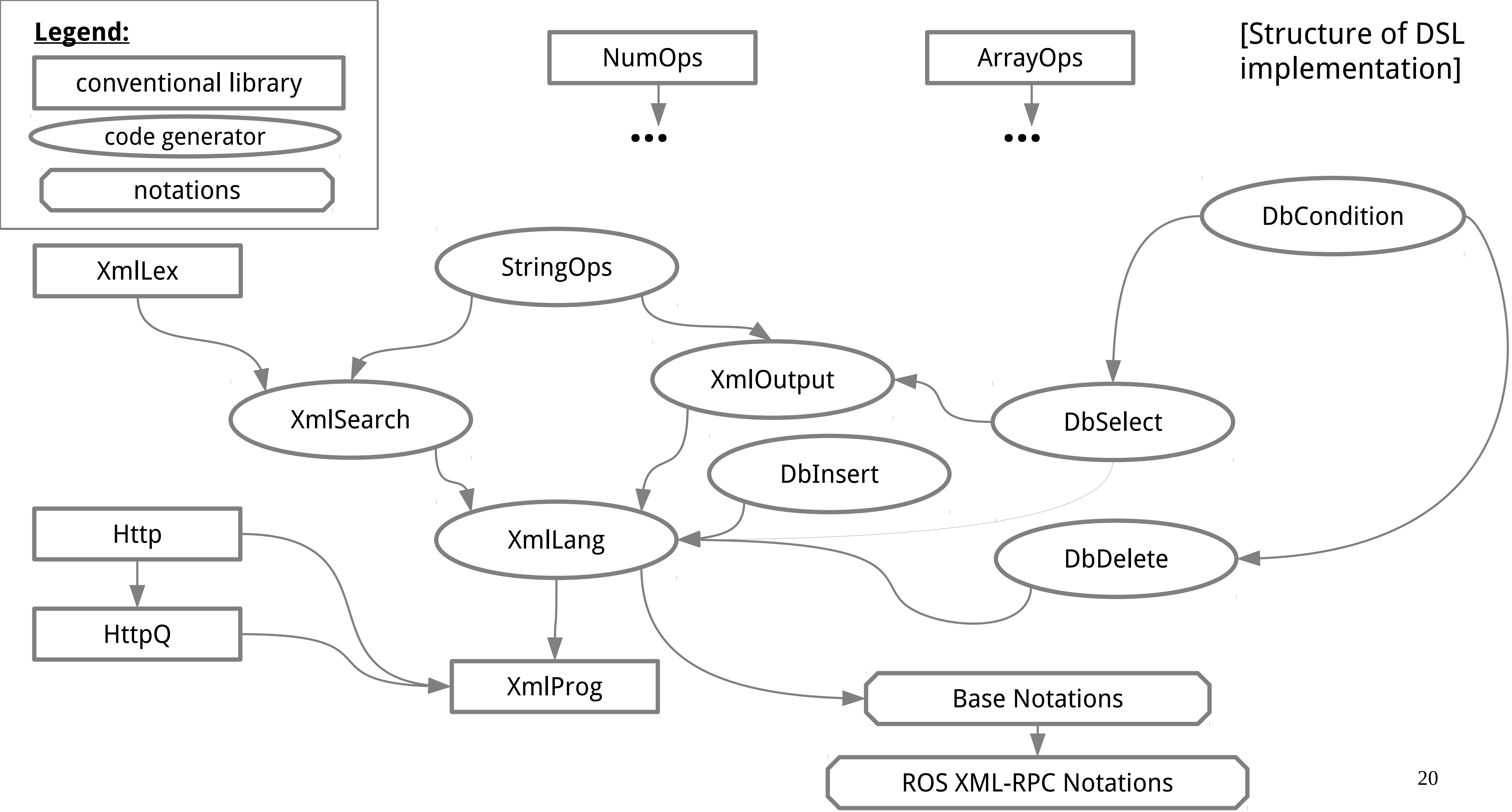
```
bfunctionNoRet "handler"("buf", "listener", "accepted", "n", "Sn")
  [handlerS]
  "listener" <-- Call "scheduler"!"listen"(port)
  [Al fs, PREmain[_, R] [| R %in fs |] * sched fs * mallocHeap 0];;
  "buf" <-- Call "buffers"!"bmalloc"(inbuf_size)
  [Al fs, PREmain[V, R] R =?>8 bsize * [| R <> 0 |] * [| freeable R inbuf_size |] * [| V "listener" %in fs|] * sched fs *
mallocHeap 0];;
  "accepted" <-- Call "scheduler"!"accept"("listener")
  [Al fs, PREmain[V, R] [| R %in fs |] * V "buf" =?>8 bsize * [| V "buf" <> 0 |] * [| freeable (V "buf") inbuf_size |] * [| V
"listener" %in fs|] * sched fs * mallocHeap 0];;
  "n" <-- Call "scheduler"!"read"("accepted", "buf", bsize)
  [Al fs, PREmain[V] [| V "accepted" %in fs |] * V "buf" =?>8 bsize * [| V "buf" <> 0 |] * [| freeable (V "buf") inbuf_size |] *
[| V "listener" %in fs|] * sched fs * mallocHeap 0];;
  "Sn" <- "n" + 1;;
  Call "scheduler"!"close"("accepted")
  [Al fs, PREmain[V] V "buf" =?>8 bsize * [| V "buf" <> 0 |] * [| freeable (V "buf") inbuf_size |] * [| V "listener" %in fs|] *
sched fs * mallocHeap 0 * [| V "Sn" = V "n" ^+ $1 |] ];;
  Call "scheduler"!"close"("listener")
  [Al fs, PREmain[V] V "buf" =?>8 bsize * [| V "buf" <> 0 |] * [| freeable (V "buf") inbuf_size |] * sched fs * mallocHeap 0 * [|
V "Sn" = V "n" ^+ $1 |] ];;
  Call "buffers"!"bfree"("buf", inbuf_size)
  [Al fs, PREmain[V] sched fs * mallocHeap 0 * [| V "Sn" = V "n" ^+ $1 |] ];;
  Call "sys"!"printInt"("Sn")
  [Al fs, PREmain[V] sched fs * mallocHeap 0 * [| V "Sn" = V "n" ^+ $1 |] ];;
  Exit 100
end

Ltac t := try solve [ sep unf hints; auto ];
  unf; unfold localsInvariantMain; post; evaluate hints; descend;
    try match_locals; sep unf hints; auto.

Theorem ok : moduleOk m.
Proof.
  vcgen; abstract t.
Qed.
```

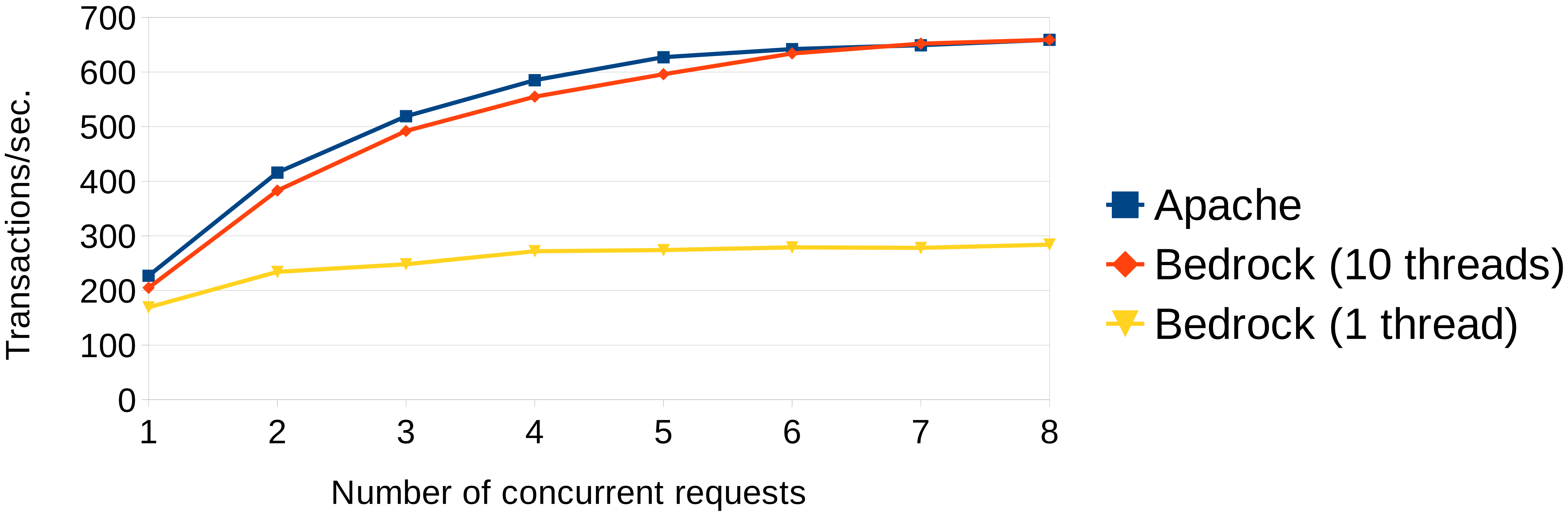
Modular Verification of a DSL Compiler

Idea: Give a **feature-modular** proof of the DSL compiler.
Define different language features as standalone **macros** that should be usable independently or within other DSLs.

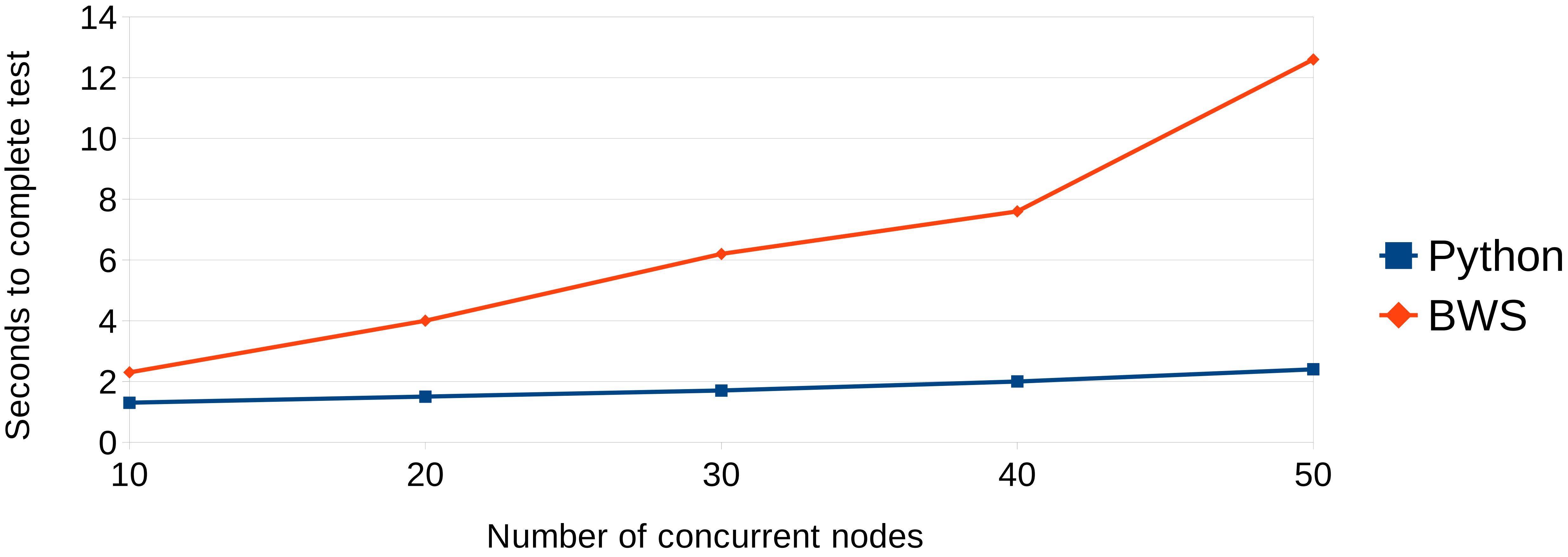


Legend:
conventional library
code generator
notations
NumOps
ArrayOps
[Structure of DSL implementation]
DbCondition
XmlLex
StringOps
XmlOutput
DbSelect
XmlSearch
DbInsert
Http
XmlLang
DbDelete
HttpQ
XmlProg
Base Notations
ROS XML-RPC Notations

Performance Test #1: Static Web Server



Performance Test #2: Robot Directory Server



Thanks for listening!

Summary: It is feasible today to verify a usable system
including both infrastructure and application code,
with a modular reasoning style,
mostly automated proofs,
and a final theorem checked in Coq with minimal trust dependencies.

Bedrock is on the Web at:

`http://plv.csail.mit.edu/bedrock/`

Backup Slides

The Bedrock Intermediate Language

W ::= (* width-32 bitvectors *)
L ::= (* program code block labels *)

Reg ::= Sp | Rp | Rv
Loc ::= Reg | W | Reg + W
Lvalue ::= Reg | $[\text{Loc}]_{32}$ | $[\text{Loc}]_{8}$
Rvalue ::= Lvalue | W | L
Binop ::= + | - | *
Test ::= = | != | < | <=

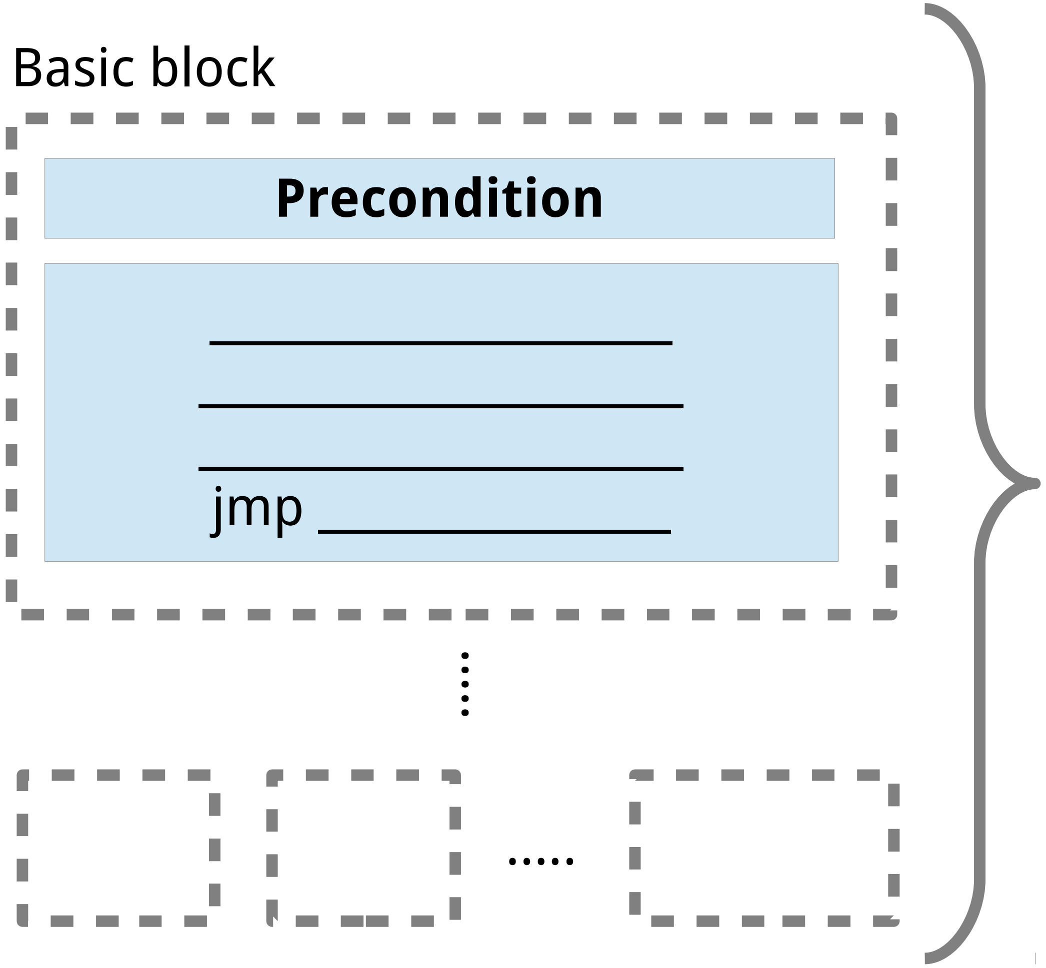
Instr ::= Lvalue := Rvalue | Lvalue := Rvalue Binop Rvalue

Jump ::= goto Rvalue | if Rvalue Test Rvalue then goto L else goto L

Block ::= Instr*; Jump
Module ::= (L: Block)*

Verification Foundation: XCAP [Ni & Shao, POPL 2006]

1. Whole programs



2. Modules

Code: .....

Assumptions: {spec1}label1, {spec2}label2, ...

Proof: Assumptions imply no precondition violations within these blocks.

Correct program:
Each precondition is true each time we reach it.

3. Linking