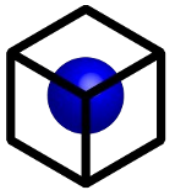


VST & DeepSpec: program verification in a verified system stack

Lennart Beringer

Princeton University

Septmber 23th – 25th, 2019

VST: a foundational verification tool for C in Coq

Floyd: forward-symbolic analysis, partial solution of side conditions using Ltac or verified decision procedures.

Concurrency (Dijkstra-Hoare + fine-grained), impredicative quantification, ...

Partial correctness + safety + limited information flow.

Expressive, modular, foundational, semi-automatic program logic for C.

Higher-order separation logic

Soundness proof for step-indexed model formalized w.r.t. operational semantics.

Clight, as formalized in CompCert

CompCert: compilation to x86-32/64, ARM, PowerPC, RiscV preserves externally visible behavior

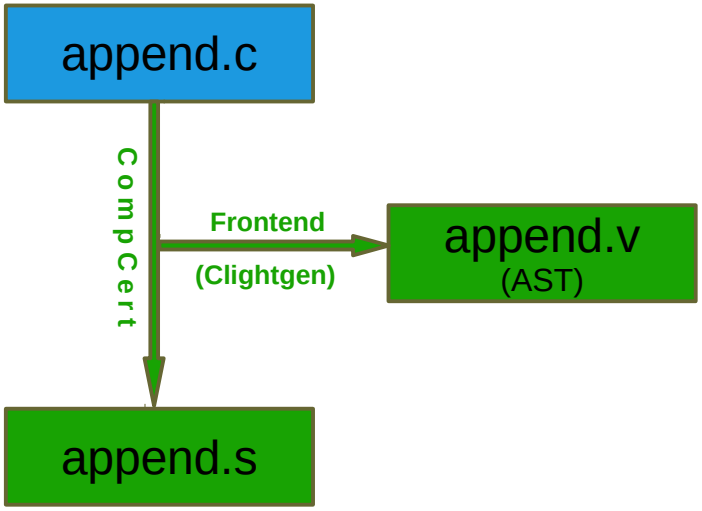
Typical workflow

Statically provided | Dynamically generated | User supplied

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert



Typical workflow

Statically provided | Dynamically generated | User supplied

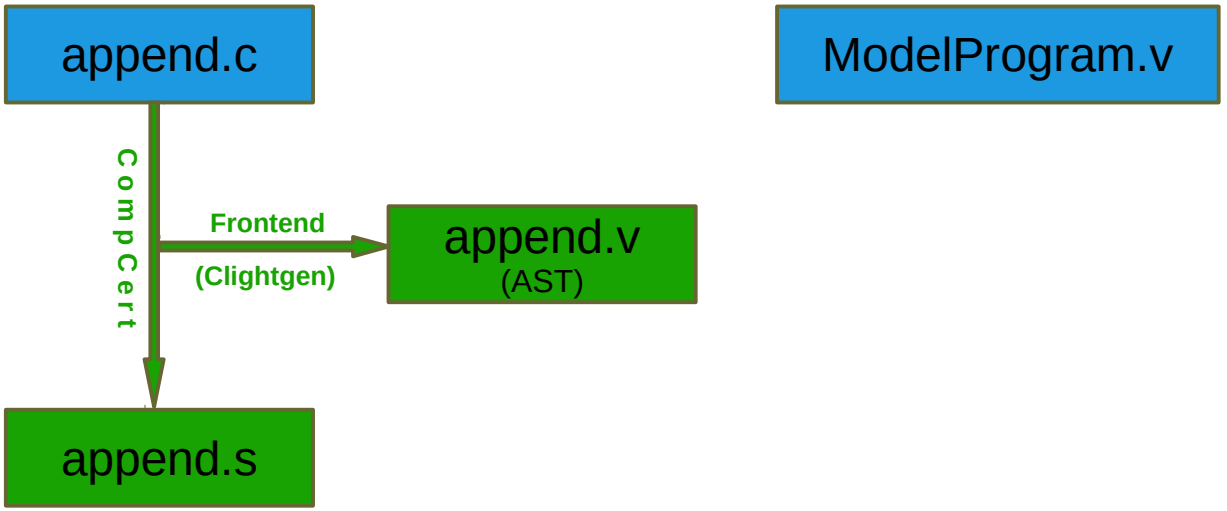
1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```



Typical workflow

Statically provided | Dynamically generated | User supplied

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```

4. Write a VST specification

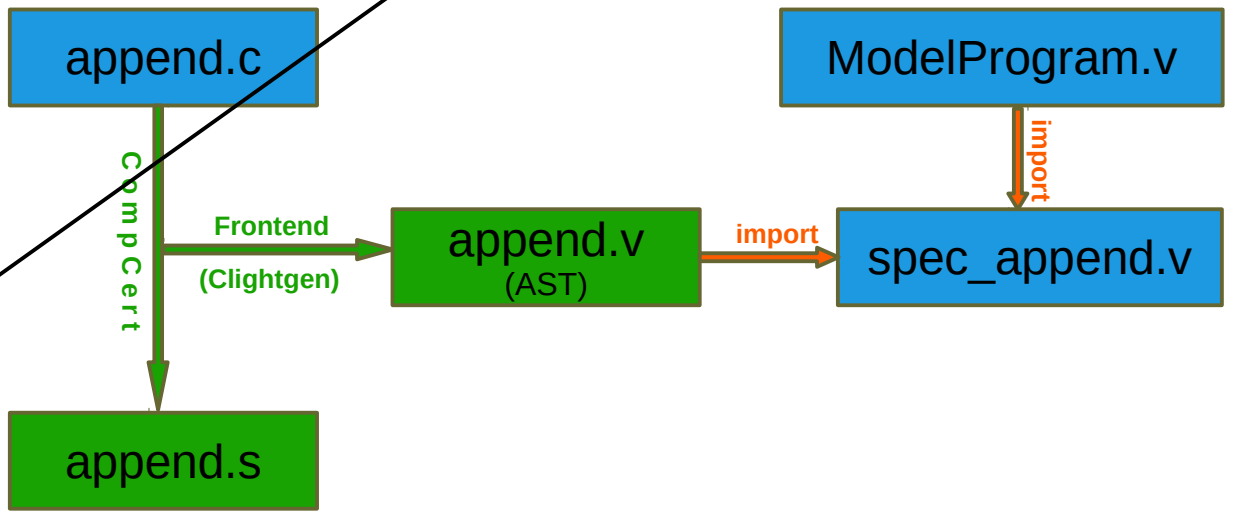
Aux. Variables (arb. Coq type)

```
Definition append_spec :=
 DECLARE _append
  WITH sh : share, x: val, y: val, s1: list val, s2: list val
  PRE [ _x OF (tptr t_struct_list) , _y OF (tptr t_struct_list)]
     PROP(writable_share sh)
     LOCAL (temp _x x; temp _y y)
     SEP (lseg_LS sh s1 x nullval; lseg_LS sh s2 y nullval)
  POST [ tptr t_struct_list ]
    EX r: val,
     PROP()
     LOCAL(temp ret_temp r)
     SEP (lseg_LS sh (s1++s2) r nullval).
```

Precondition

User-defined repr. predicate

Postcondition



Typical workflow

Statically provided | Dynamically generated | User supplied

1. Write a C program

```
#include <stddef.h>

struct list {int head; struct list *tail;};

struct list *append (struct list *x, struct list *y) {
  struct list *t, *u;
  if (x==NULL)
    return y;
  else {
    t = x;
    u = t->tail;
    while (u!=NULL) {
      t = u;
      u = t->tail;
    }
    t->tail = y;
    return x;
  }
}
```

2. Parse and compile using Clightgen/Compcert

3. Write a model program in Gallina

```
Fixpoint app (al bl: list Z) : list Z :=
 match al with
 | nil => bl
 | a::al' => a :: app al' bl
 end.
```

4. Write a VST specification

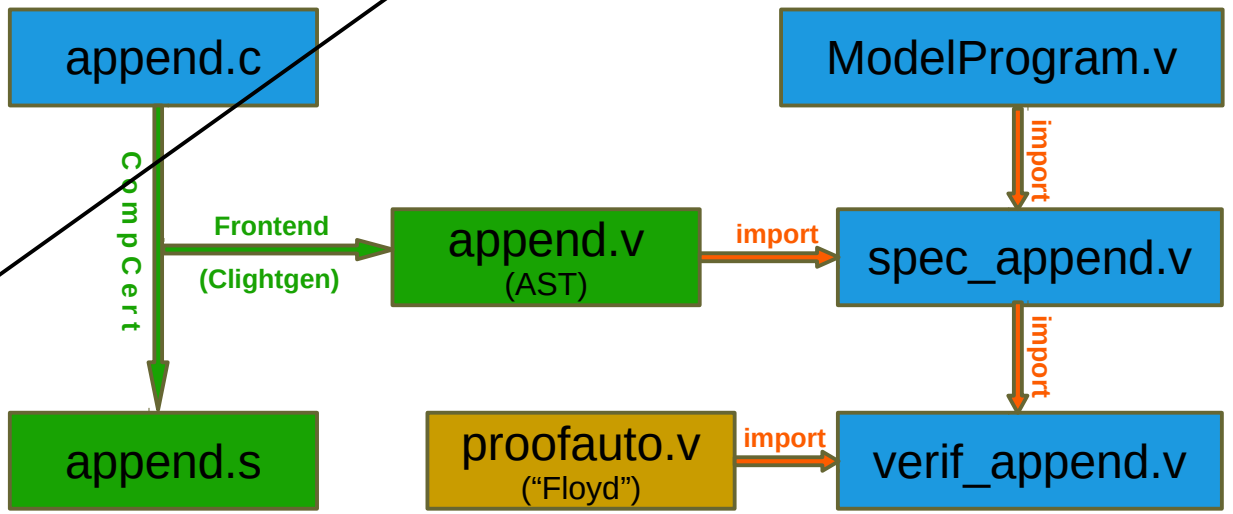
```
Definition append_spec :=
 DECLARE _append
  WITH sh : share, x: val, y: val, s1: list val, s2: list val
  PRE [ _x OF (tptr t_struct_list) , _y OF (tptr t_struct_list)]
     PROP(writable_share sh)
     LOCAL (temp _x x; temp _y y)
     SEP (lseg_LS sh s1 x nullval; lseg_LS sh s2 y nullval)
  POST [ tptr t_struct_list ]
    EX r: val,
     PROP()
     LOCAL(temp ret_temp r)
     SEP (lseg_LS sh (s1++s2) r nullval).
```

Aux. Variables (arb. Coq type)

Precondition

User-defined repr. predicate

Postcondition

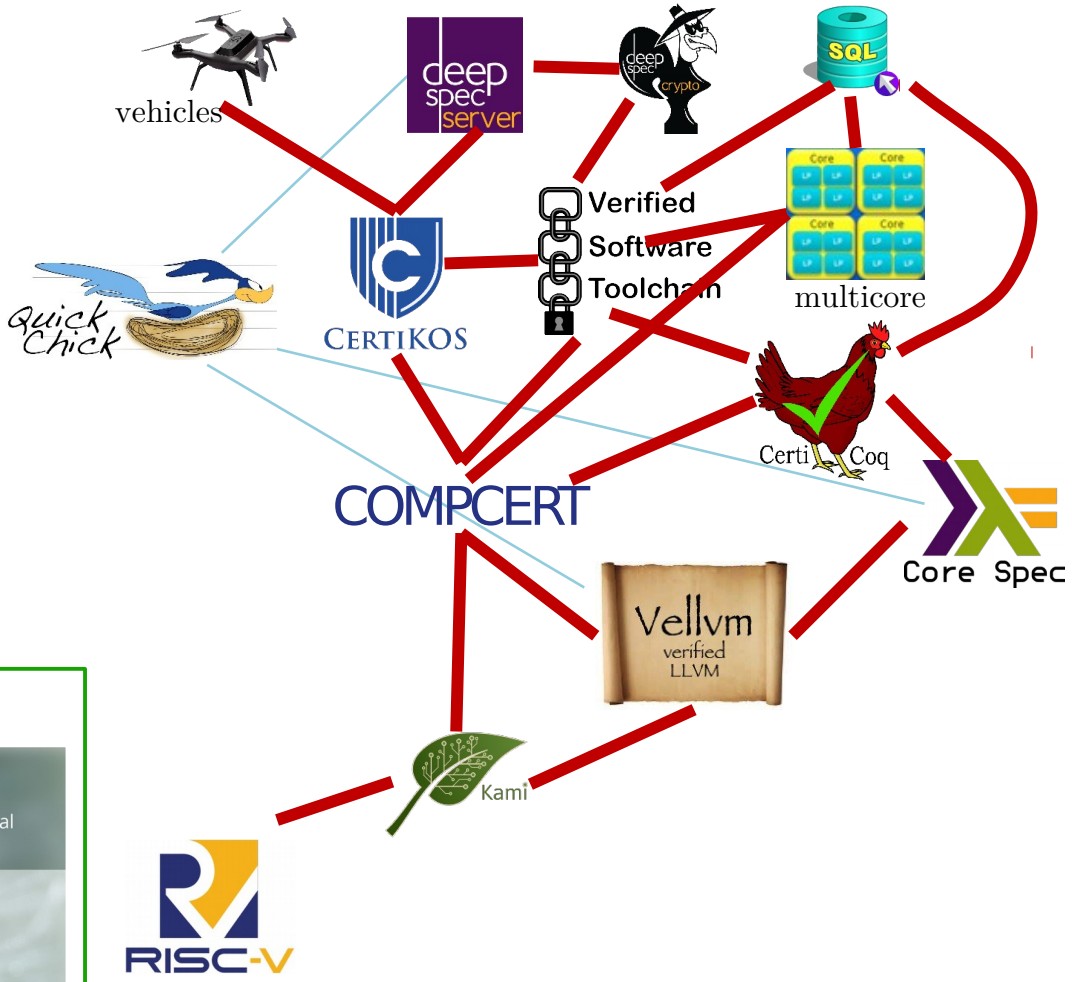


5. Prove the function body (define loop invariants on demand)

```
Lemma body_append: semax_body Vprog Gprog f_append append_spec.
Proof. start_function. ... ( proof script ) ... . Qed.
```

VST in context: deep spec (2016 – 2020), https://deepspec.org

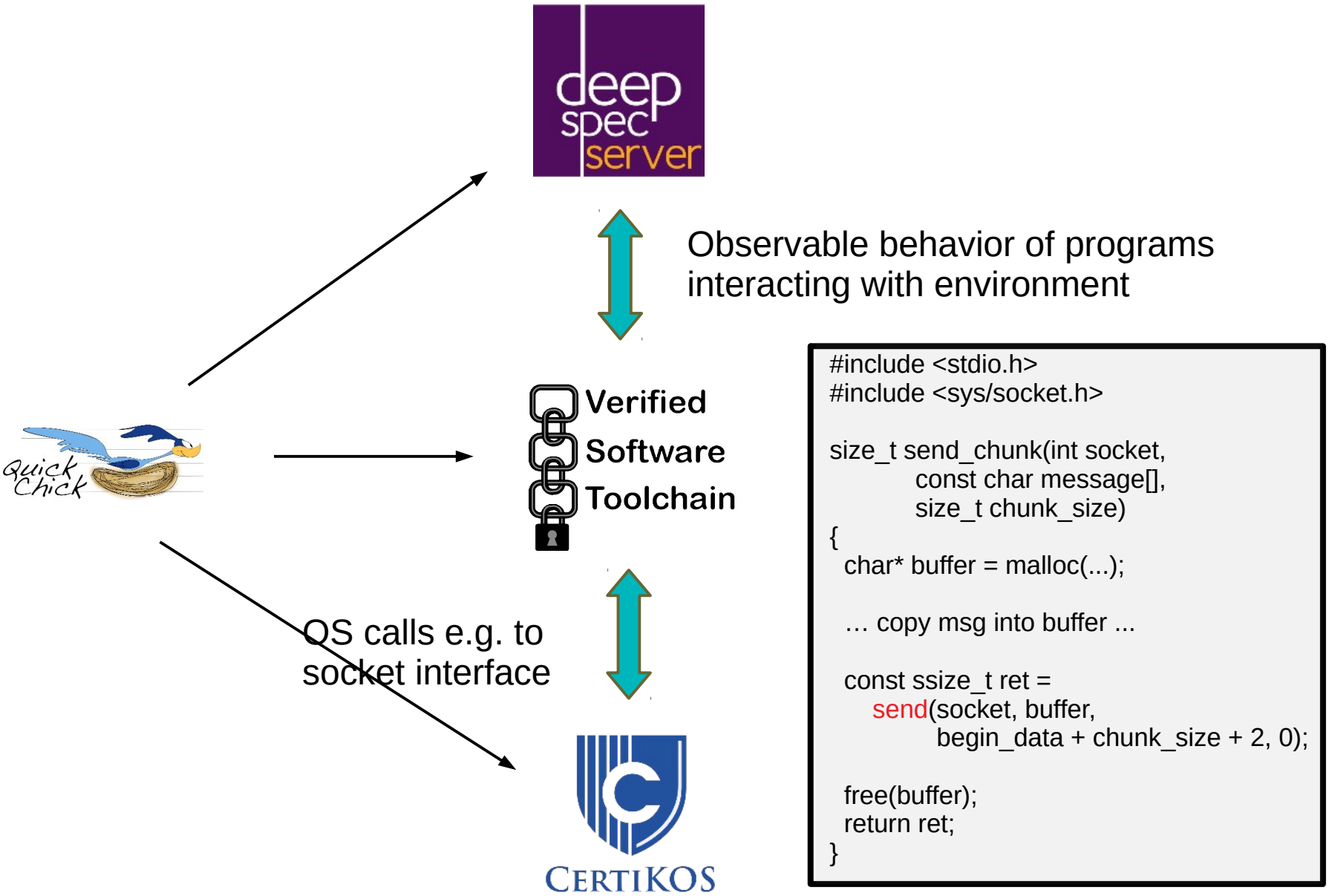
RICH	describe complex behaviors in detail
FORMAL	in notation with a clear semantics
2-SIDED	connected to clients & implementations
LIVE	machine-checked connection to implementations



Community building:
- summer schools ‘17, ‘18, ‘20
- workshops at PLDI etc.

Curriculum development:

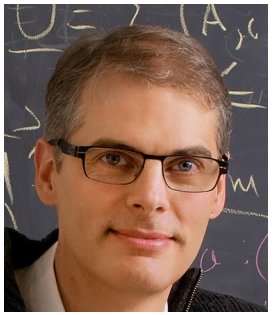
Today's topics




```
#include <stdio.h>
#include <sys/socket.h>

size_t send_chunk(int socket,
          const char message[],
          size_t chunk_size)
{
  char* buffer = malloc(...);

  … copy msg into buffer ...

  const ssize_t ret =
     send(socket, buffer,
            begin_data + chunk_size + 2, 0);

  free(buffer);
  return ret;
}
```

Leonidas Lampropoulos

QuickChick



Benjamin C. Pierce

Extends random-based testing (QuickCheck) to **property-based** testing of **executable** Coq formalizations

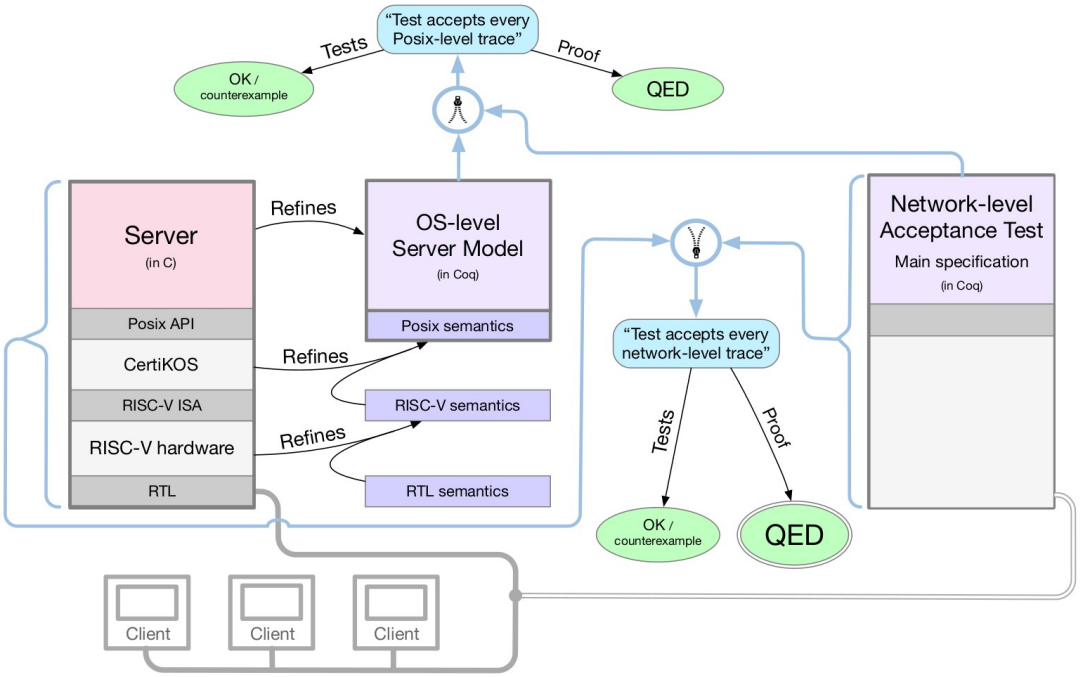
Can precede formal proof, to help debugging specs or components

Steer test case synthesis towards “areas of interest”

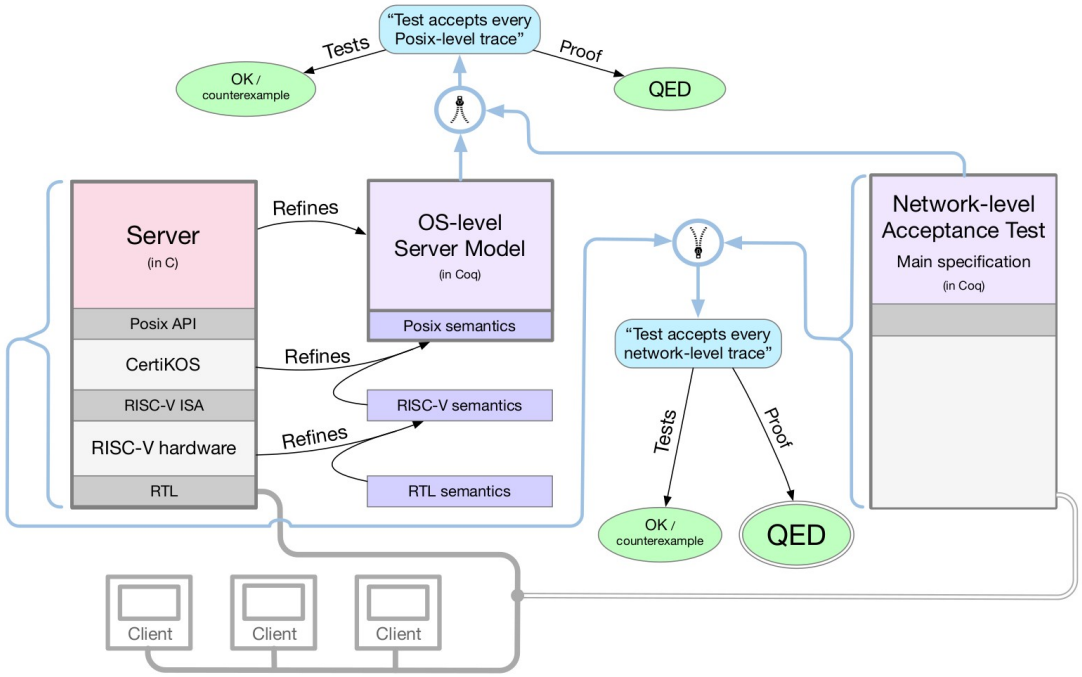
Example: testing soundness of a type system

Applied extensively to web servers (real-world, and DS simplifications) but also C programs (AES), Vellum, Haskell2Coq, equivalence of functions/models

Webserver challenges

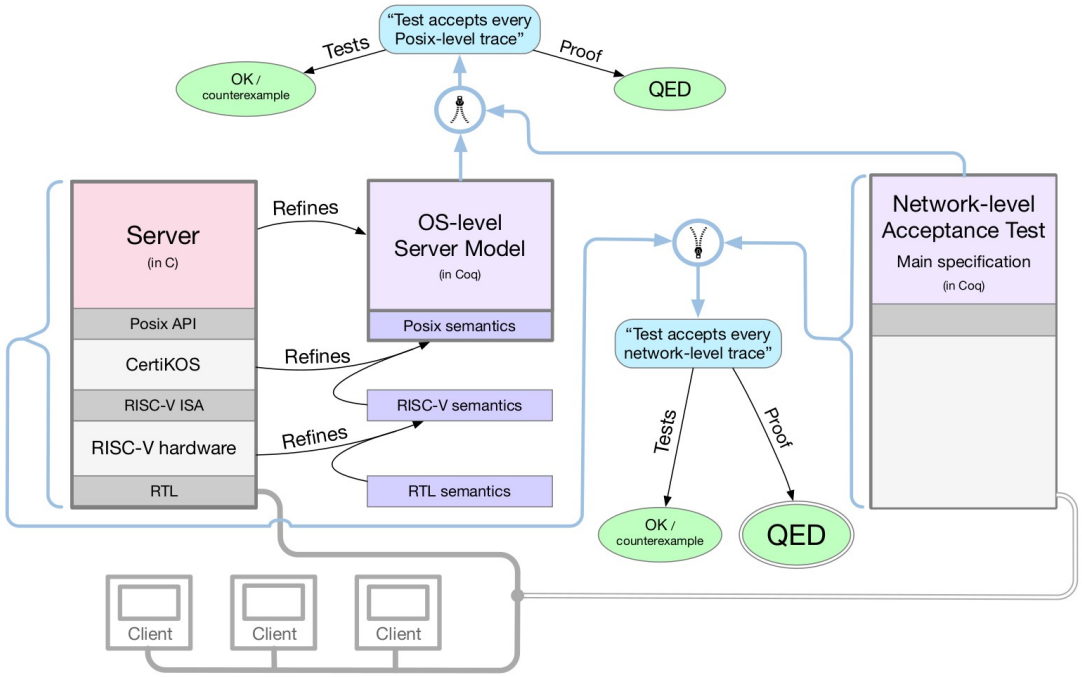


Webserver challenges



- Spec of a (web)server (cf REMS)?
  - RFC 2616 (http),
  - Testing
- level of abstraction: messages, packets, bytes ?
- how can we relate these to each other and to the C code?
- reorderings in OS, on the network..

Webserver challenges



- Spec of a (web)server (cf REMS)?
  - RFC 2616 (http),
  - Testing
- level of abstraction: messages, packets, bytes ?
- how can we relate these to each other and to the C code?
- reorderings in OS, on the network..

- Implementation-level / interaction with OS:
  - spec of socket interface (RFC)
  - compatibility of specification formalisms of VST and CertiKOS
- Specification / verification / engineering tradeoffs?
  - robustness of specification approaches for different implementation styles (event loops, …), concurrency...

Interaction Trees

An `M E X` is the denotation of a program as a possibly infinite ("coinductive") tree, parameterized over a collection `Event` of *observable events* where:

- the leaves correspond to *final results* labeled with `X`,
- internal nodes node are either *internal events* (labeled `Tau`),
- or *observable events* (labeled `Vis`, with a child for every `y` of the event's result type `Y`).

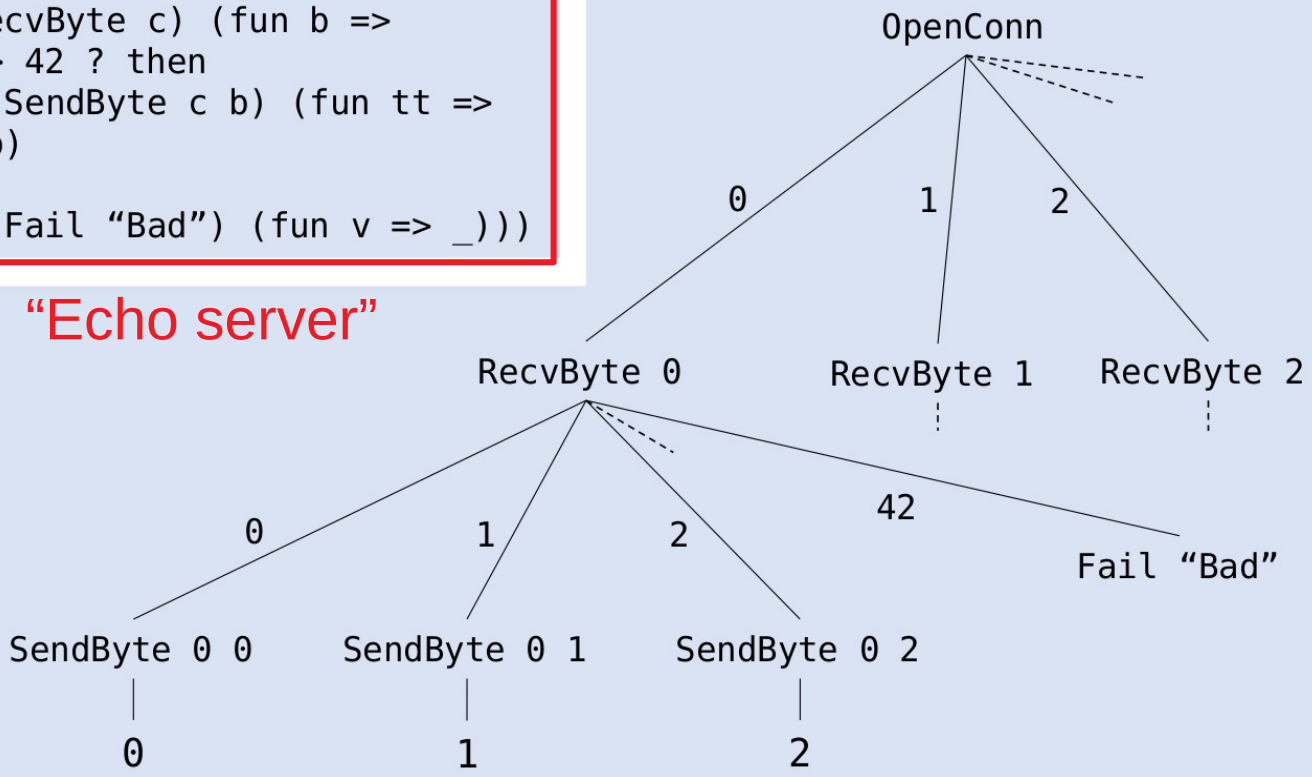
```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Tau (k: M Event X).
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
```

(The Vis/Ret/Tau constructors are going to be hidden behind monadic notations.)

Interaction Trees (graphically)

```
Vis  OpenConn    (fun c =>
Vis (RecvByte c) (fun b =>
if b <> 42 ? then
  Vis (SendByte c b) (fun tt =>
  Ret b)
else
  Vis (Fail “Bad”) (fun v => _)))
```

“Echo server”



Itrees are executable in Coq, so specs are amenable to QuickChick

Network events:

```
Inductive networkE : Type -> Type :=
| OpenConn : networkE connection
| CloseConn : connection -> networkE unit
| ReadByte : connection -> networkE (option byte)
| WriteByte : connection -> byte -> networkE unit.

Definition read_byte conn : M networkE (option byte) :=
  Vis (ReadByte conn) Ret.
```

Interaction Trees in VST

Feature of SL (since 2000): resource-accounting

- define **abstract** SL assertion **Itree (t)** that embeds an I-Tree **t** in a spec
- predicate **Itree (t)** can't be duplicated or created out-of-thin-air
- regular C instructions cannot touch **Itree (t)**
- calls to socket-API functions "advance the Itree", via specs like

{ SockAPI st *
Itree (r ← recv fd len; k r) *
data_at_ len buf }

r = **recv** (buf,len)

{ r>0 /\ SockAPI st' * Itree (k r)
* data_at len msg buf
\/ r=0 /\ … (error case) }

Feature of SL (since ca 2015): "separating ghost state"

- can semantially ensure non-modifiability of **Itree (t)** by regular code
- **Itree (t)** 's footprint is not in real memory but in the outside world
- other use of ghost state with separation structure: concurrency

Ongoing work

Itree modeling network communication

"network refinement"

- **abstract from control flow**
- **relate byte stream to packets and messages**
- **Non-determinism / reordering**

Itree modeling C ↔ **OS interaction**

Ongoing work

Itree modeling network communication

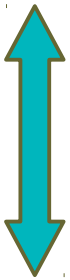
"network refinement"

- **abstract from control flow**
- **relate byte stream to packets and messages**
- **Non-determinism / reordering**

Itree modeling C ↔ **OS interaction**

VST interpretation of Itree-enriched socket-specs

- interpreted in (concurrent) ghost state SL
- with "step-indexed" model of assertions



- **common basis: CompCert (memory model)**
- **can "hide" SL information before socket call and "recover" it upon return**

CertiKOS interpretation of socket-specs, justified w.r.t. OS implementation

- no step-indexing
- non-SL notions of resource isolation and abstraction

Discussion

Verified operating systems (CertiKOS, seL4):

- excellent basis for trustworthy system stacks
- complementary (foundational) interfaces

Progress on VST

- modularity: subsumption, component framework
- concurrency

Formal verification and testing can go hand-in-hand

- Unifying framework Coq/Isabelle-HOL

Complementary VST/OS interface:
OS is master, initiates execution of C program (slave)

Interaction trees:

- general formalism for external interactions
- rich refinement and observational equivalence theory, implemented in stand-alone Coq library
- coinductive / monadic code representation: potential bridge to seL4?