From C to Interaction Trees

Specifying, Verifying, and Testing a Networked Server

Nicolas Koh[†] Yao Li[†] Yishuai Li[†] Li-yao Xia[†]
Lennart Beringer[§] Wolf Honore[*] William Mansky[¶] Benjamin C. Pierce[†] Steve Zdancewic[†]
University of Pennsylvania[†] Princeton University[§] Yale University[*] University of Illinois at Chicago[¶]

Abstract

We present the first formal verification of a networked server implemented in C. *Interaction trees*, a general structure for representing reactive computations, are used to tie together disparate verification and testing tools (Coq, VST, and QuickChick) and to axiomatize the behavior of the operating system on which the server runs (CertiKOS). The main theorem connects a specification of acceptable server behaviors, written in a straightforward "one client at a time" style, with the CompCert semantics of the C program. The variability introduced by low-level buffering of messages and interleaving of multiple TCP connections is captured using *network refinement*, a variant of observational refinement.

Keywords formal verification, testing, TCP, interaction trees, network refinement, VST, QuickChick

1 Introduction

The Science of Deep Specification [Appel et al. 2017] is an ambitious experiment in specification, rigorous testing, and formal verification of real-world systems such as web servers "from internet RFCs all the way to transistors." The principal challenges lie in integrating disparate specification styles, legacy specifications, and testing and verification tools to build and reason about complex, multi-layered systems.

We report here on a first step toward realizing this vision: an in-depth case study demonstrating how to specify, test, and verify a simple networked server with the same fundamental interaction model as more sophisticated ones—it communicates with multiple clients via ordered, reliable TCP connections. Our server is implemented in C and verified, using the Verified Software Toolchain [Appel 2014], against a formal "implementation model" written in Coq [2018]; this is further verified (in Coq) against a linear "one client at a time" specification of allowed behaviors. The main property we prove is that any trace that can be observed by a collection of concurrent clients interacting with the server over the network can be rearranged into a trace that is allowed by the linear specification. We also show how property-based random testing using Coq's QuickChick plug-in [Lampropoulos and Pierce 2018] can be deployed in this setting. We compile the server code with the CompCert verified compiler [Leroy 2009] and run it on CertiKOS [Gu et al. 2016], a verified operating system with support for TCP socket operations.

Our verified server provides a simple "swap" interface that allows clients to send a new bytestring to the server and receive the currently stored one in exchange. It is simpler in many respects than a full-blown web server; in particular, it follows a much simpler protocol (no authentication, encryption, header parsing, *etc.*), which means that it can be implemented with much less code.

Moreover, the degree of vertical integration falls short of our ultimate ambitions for the DeepSpec project, since we stop at the CertiKOS interface (which we axiomatize) instead of going all the way down to transistors. On the other hand, the C implementation of our server is realistic enough that it offers a challenging test of how to integrate disparate Coq-based methodologies and tools for verifying and testing systems software. In particular, it uses a single-process, event-driven architecture [Pai et al. 1999], hides latency by buffering interleaved TCP communications from multiple clients, and is built on the POSIX socket API.

Contributions We describe our experiences integrating Coq, CompCert, VST, CertiKOS, and QuickChick to build a verified swap server. This is the first VST verification of a program that interacts with the external environment. It is also, to the best of our knowledge, the first verification of functional correctness of a networked server implemented in C. Our technical contributions are as follows:

First, we identify *interaction trees* (ITrees)—a Coq adaptation of structures known variously as "freer" [Kiselyov and Ishii 2015], "general" [McBride 2015], or "program" [Letan et al. 2018] monads—as a suitable unifying structure for expressing and relating specifications at different levels of abstraction (Section 3).

Second, we adapt standard notions of *linearizability* and *observational refinement* from the literature on concurrent data structures to give a simple specification methodology for networked servers that is suitable both for rigorous property-based testing and for formal verification. We call this variant *network refinement* (Section 4).

Third, we demonstrate practical techniques for both *verifying* (Section 5) and *testing* (Section 6) network refinement between a low-level implementation model and a simple linear specification. We also demonstrate testing against the compiled C implementation across a network interface.

CPP, January 2019, Lisbon, Portugal
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

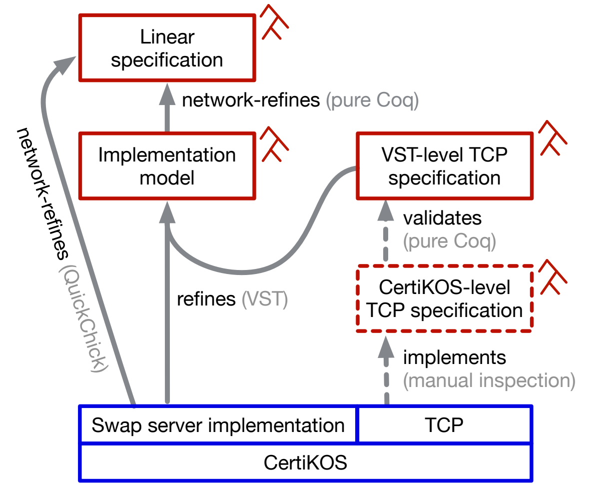

Figure 1. Overview. The blue parts of the figure represent components written in C, the red parts specifications in Coq. The swap server implementation runs on top of CertiKOS; it is proved to refine the implementation model with respect to a VST axiomatization of the TCP system calls; this, in turn, is validated by a lower-level axiomatization in the style of CertiKOS, which is manually compared to the (unverified) TCP implementation. The implementation model, in turn, "network refines" the linear specification. The fact that the C implementation network refines the linear specification is independently validated by property-based random testing. In all the Coq models and specifications, interaction trees model the observable behaviors of computations. The dotted parts of the figure are either informal or incomplete.

Lastly, the ITrees embedded into both VST's separation logic and CertiKOS's socket model allow us to make progress on connecting the two developments. Though completing the formal proofs remains for future work, we identify the challenges and describe preliminary results in Section 7.

Section 2 summarizes the whole development. Sections 8 and 9 discuss related and future work. A tarball containing all our Coq and C code has been provided to the PC chairs.

2 Overview

Figure 1 shows the high-level architecture of the entire case study. This section surveys the major components, starting with the high-level, user-facing specification (the linear specification shown at the top of the figure) and working down to OS-level details.

Specifying the Swap Server Informally, the intended behavior of the swap server is straightforward. Any number of clients can connect and send "swap requests," each containing a fixed-size message. The server acts as a one-element concurrent buffer: it retains the most recent message that it has received and, upon getting a swap request, updates its state with the new message and replies to the sender with the old one. The left-hand side of Figure 2 shows a simple example of correct behavior of a swap server.

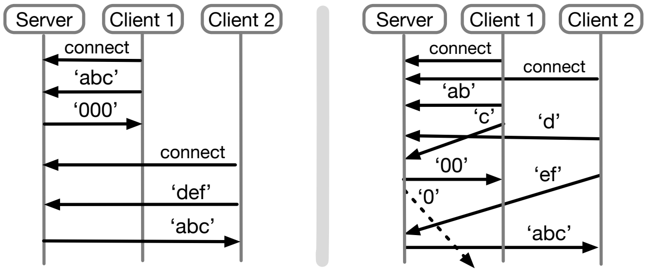

Figure 2. Swap server examples. On the left is a simple run that directly illustrates the linear specification. Each client in turn establishes a connection, sends a three-byte message, and receives the message currently stored on the server as a response. ('000' is the server's initial state.) On the right is another run illustrating internal buffering by the swap server and reordering by the network. Messages may be sent in multiple chunks, messages from different clients may be received out of order, and messages may be delayed indefinitely (dotted arrow). The "explanation" of the two runs in terms of the linear specification is the same.

```
CoFixpoint linear_spec' (conns : list connection_id)
           (last_msg : bytes) : itree specE unit :=
  or ( (* Accept a new connection. *)
       c ← obs_connect;;
       linear_spec' (c :: conns) last_msg )
     ( (* Exchange a pair of messages on a connection. *)
       c ← choose conns;;
       msg ← obs_msg_to_server c;;
       obs_msg_from_server c last_msg;;
       linear_spec' conns msg ).

Definition linear_spec := linear_spec' [] zeros.
```

Figure 3. Linear specification of the swap server. In the `linear_spec'` loop, the parameter `conns` maintains the list of open connections, while `last_msg` holds the message received from the last client (which will be sent back to the next client). The linear specification is initialized with an empty set of connections and a message filled with zeros.

Figure 3 shows the linear specification of the server's behavior. It says that the server can either accept a connection with a new client (`obs_connect`) or else receive a message from a client over some established connection (`obs_msg_to_server c`), send back the current stored message (`obs_msg_from_server c last_msg`), and then start over with the last-received message as the current state. The set of possible behaviors is represented as an interaction tree (of type `itree specE unit`).

Our main correctness theorem should relate the actual behavior of our server (the CompCert semantics of the C code) to this linear description of its desired behavior. Informally:

Theorem 1. *Any sequence of interactions with the swap server that can be observed by clients over the network could have been produced by the linear specification.*

```
Theorem swap_server_correct :
  ∃ impl_model, ext_behavior C_prog impl_model ∧
    network_refines linear_spec impl_model.
```

Figure 4. End-to-end swap server correctness theorem.

This theorem constrains the server to act as a swap server: it prevents the server from sending a message before it receives one, or while it has only received a partial message; it prevents it from sending an arbitrary value in response to a request, or replying multiple times with the same value that has only been received once; it prevents it from sending a response to a client from which it has not received a request. However, the “over the network” clause is a significant caveat: the server communicates with clients via TCP, and even a correct implementation might thus exhibit a number of undesirable behaviors from the clients' point of view. The network might drop all packets after a certain point, causing the server to appear to have stopped running, so the theorem allows the server to stop running at any point. Similarly, the network might delay messages and might reorder messages on different connections, so the theorem allows the server to respond to an earlier request after responding to a later request. The right-hand side of Figure 2 shows another run of the system illustrating these possibilities; it should also be accepted by the top-level theorem.

Figure 4 shows the formal specification linking the *linear specification* (`linear_spec`), which describes interactions with one client at a time, to the C program (`C_prog`). It is split in two parts articulated around an *implementation model* (`impl_model`). It is another interaction tree that describes the network-level behavior of the C program more closely than the linear specification. Like the C program, the implementation model interleaves requests from multiple clients and accounts for the effects of the network. A *refinement* between the C program and the implementation model is formalized by the VST property `ext_behavior`. Then the implementation model is connected to the specification by a different *network refinement* layer (`network_refines`).

Network refinement The linear specification is short and easy to understand, but an implementation that strictly followed it would be *obliged* to serve clients sequentially, which is not what real servers (including ours) want to do. Moreover, as shown on the right-hand side of Figure 2, the network may delay and reorder messages, so that, for example, the first two bytes of a message from client 1 might be received after the first byte of a message from client 2. The server should be able to account for this by buffering messages until they are complete. The second part of our server specification loosens the linear specification to account for the effects of communicating over a network; this also permits realistic implementations that serve multiple clients concurrently.

Network refinement states that every possible behavior of the implementation model is allowed by the linear specification, while accounting for message reordering and buffering that might be introduced by the network and/or server. Section 4 explains this process in more detail.

C Implementation Our C implementation is a simple but reasonably performant server in a classical single-process, event-driven style [Pai et al. 1999]. The implementation maintains a list of `connection` structures, each representing a state machine for one connection. Specifically, a connection structure contains (1) a state, which may be `RECVING`, `SENDING`, or `DELETED`; (2) a buffer for storing bytes that have been received on the connection; and (3) a buffer for storing bytes to send on the connection.

The main body of the server is a non-terminating loop (Figure 5); in each iteration, it uses the `select` system call to check for pending connections to accept and for existing connections ready for receiving/sending bytes from/to, and processes them. A new connection is handled by initializing a new connection structure and adding it into the list, and an existing connection is processed by updating the read/write buffers and advancing the connection's state appropriately. This buffering strategy lets the server interleave processing of multiple connections without having to wait for one client to send or receive a complete message.

Verifying the C code To prove that the C implementation refines the implementation model (that is, that every possible network behavior of the C program is allowed by the implementation model), we use VST, a tool for proving correctness of C programs using separation logic. The VST predicate `ext_behavior C_prog impl_model` in Figure 4 relates the operational semantics of the C program `C_prog` to the interaction tree description given by `impl_model`. Section 3 describes the implementation model in more detail.

VST's model of program execution includes both conventional program state (memory, local variables, *etc.*) and *external state*, an abstract representation of the state of the environment in which the program is running. We connect the C program semantics to the implementation model by adding a predicate `ITree`(t) to VST's separation logic, asserting that the environment expects the C program's network behavior to match the interaction tree t. Section 5 describes this process.

Assumptions and modeling gaps We have a complete proof (using VST) that the C implementation compiled with CompCert network-refines the linear specification—that is, a complete proof of the claim in Figure 4. This proof is grounded in axiomatic specifications of the OS-level system calls, and some library functions. We rely on the soundness of the Coq proof assistant, plus the standard axioms of functional and propositional extensionality and proof irrelevance.

For this case study, our verification bottoms out at the interface between the application program and the operating system; we rely on the correctness of the OS's socket library and of the OS itself. Since we are running on CertiKOS,

```
while(1 == 1) {
  ...
  int num_ready =
    select(maxsock + 1, &rs, &ws, &es, &timeout);
  if (num_ready <= 0) { continue; }
  int socket_ready = fd_isset_macro(server_socket, &rs);
  if (socket_ready) {
    /* Accept a new connection on the socket, create a
       connection structure for it, and link it into the
       head of the linked list of connections. */
    accept_connection(server_socket, &head);
  }
  /* For each connection in the list pointed to by head,
     read from or write to its buffer of data. */
  process_connections(head, &rs, &ws, last_msg_store);
}
```

Figure 5. Main loop of swap server (in C)

the OS has actually been proved correct, but its correctness proofs and ours are not formally connected. That is, our specification of its socket API is axiomatized, but the axioms are partially validated by connection to the corresponding CertiKOS specifications (specifically, a VST specification of `recv` has been partly connected to the CertiKOS-level one; the other socket primitives remain to be connected). There are several remaining challenges with connecting VST to CertiKOS, ranging from the semantic—one critical technicality is connecting VST's step-indexed view of memory with the flat memory model used by CertiKOS—to the technical—they use different versions of Coq. See Section 7 for a fuller description of what we have done to bridge these two formalizations. Also, because CertiKOS currently does not provide a verified TCP implementation, the best it can do is mediate between the VST axioms and some, possibly lower-level, axiomatization of the untrusted TCP stack. Filling these gaps is left to future work.

Testing network refinement For our long-term goal of building verified systems software like web servers, rigorous testing will be crucial, for two reasons. First, even small web servers are fairly complex programs, and they take significant effort to verify; streamlining this effort by catching as many bugs as possible before spending much time on verification makes good economic sense, especially if the code can be automatically tested against the very same specification that will later be used in the verification effort. Second, programs like web servers must often fit into an existing ecosystem—a verified web server that interpreted the HTTP RFCs (*e.g.*, Belshe et al. [2015]) differently from Apache and Nginx would not be used. Testing can be used to validate the formal specification against existing implementations.

For the present case study, we use QuickChick [Lampropoulos and Pierce 2018], a Coq plug-in for property-based testing based on the popular QuickCheck tool [Claessen and Hughes 2000]. We test both the compiled C code (by sending it messages over a network interface) and the implementation model (by exploring its behaviors within Coq) against the linear specification.

```
CoInductive itree (E : Type → Type) (R : Type) :=
| Ret (r : R)
| Vis {X : Type} (e : E X) (k : X → itree E R)
| Tau (t : itree E R).

Inductive event (E : Type → Type) : Type :=
| Event : ∀ X, E X → X → event E.

Definition trace E := list (event E)

Inductive is_trace E R
  : itree E R → trace E → option R → Prop := ...
  (* straightforward definition omitted *)
```

Figure 6. Interaction trees and their traces of events.

Supporting property-based testing requires *executable* specifications of the properties involved. Happily, interaction trees, which play a crucial role throughout our development, also work well with Coq-style program extraction, and hence with testing. Testing must also be performed "modulo network refinement" in the same way as verification. Section 6 describes this in more detail.

3 Interaction Trees

Components that interact with their environment appear at many levels in our development (see Figure 1). We use *interaction trees* (ITrees) as a general-purpose structure for specifying such components. ITrees are a Coq adaptation of similar concepts known variously as "freer," "general," or "program" monads [Kiselyov and Ishii 2015; Letan et al. 2018; McBride 2015]. We defer a deeper comparison until Section 8.

Constructing ITrees Figure 6 defines the type `itree E R`. The definition is *coinductive*, so that it can represent potentially infinite sequences of interactions, as well as divergent behaviors. The parameter `E` is a type of *external interactions*—it defines the interface by which a computation interacts with its context. `R` is the *result* of the computation: if the computation halts, it returns a value of type `R`.

There are three ways to construct an ITree. The `Ret r` constructor corresponds to the trivial computation that halts and yields the value `r`. The `Tau t` constructor corresponds to a silent step of computation, which does something internal that does not produce any visible effect and then continues as `t`. Representing silent steps explicitly with `Tau` allows us, for example, to represent diverging computation without violating Coq's guardedness condition:

```
CoFixpoint spin {E R} : itree E R := Tau spin.
```

The final, and most interesting, way to construct an ITree is with the `Vis X e k` constructor. Here, `e : E X` is a "visible" external effect (including any outputs provided by the computation to its context) and `X` is the type of data that the context provides in response to the event. The constructor also specifies a continuation, `k`, which produces the rest of the computation given the response from the context. `Vis` introduces branches into the interaction tree, because `k` can behave differently for distinct values of type `X`.

Here is a small example that defines a type `IO` of output or input interactions, each of which works with natural numbers. It is then straightforward to define an ITree computation that loops forever, echoing each input received to the output:

```
Variant IO : Type → Type :=
| Input  : IO nat
| Output : nat → IO ().

CoInductive echo : itree IO () :=
  Vis Input (fun x ⇒ Vis (Output x) (fun _ ⇒ echo)).
```

Working with ITrees Several properties of ITrees make them appealing as a structure for representing interactive computations. First, they are *generic* in the sense that, by varying the `E` parameter, they can be instantiated to work with different external interfaces. Moreover, such interfaces can be built compositionally: for example, we can combine a computation with external effects in `E1` with a different computation with effects in `E2`, yielding a computation with effects in `E1 + E2`, the disjoint union of `E1` and `E2`; there is a natural inclusion of ITrees with interface `E1` into ITrees with interface `E1 + E2`. This approach is reminiscent of *algebraic effects* [Plotkin and Power 2003]. Our development exploits this flexibility to easily combine generic functionality, such as a nondeterministic choice effect (which provides the `or` operator used by the linear specification of Figure 3) with domain-specific interactions such as the network send and receive events. As with algebraic effects, we can write a *handler* or *interpreter* for some or all of the external interactions in an interface, for example to narrow the effects `E1 + E2` down to just those in `E1`. Typically, such a handler will process the events of `E2` and "internalize" them by replacing them with `Tau` steps.

Second, the type `itree E` is a monad [Wadler 1992], which makes it convenient to structure effectful computations using the conventions and notations of functional programming. We package up the `Ret` constructor as a `ret` (return) operation and use the sequencing notation `x ← e ;; k` for the monad's bind. With a bit of wrapping and a loop combinator `forever`, we can rewrite the echo example with less syntactic clutter:

```
Definition echo : itree IO () :=
  forever (x ← input ;; output x)
```

Third, the ITree definition works well with Coq's extraction mechanism, allowing us to represent computations as ITrees and run them for testing purposes. Here again, the ability to provide a separate interpretation of events is useful, since its meaning can be defined outside of Coq. In the echo example, `Output` events could be linked to a console output or to an OS's network-send system call. ITrees thus provide *executable* specifications.

```
r ← or e1 e2 ;; k  ⊑  r ← ei ;; k        i ∈ {1,2}
r ← choose l ;; k  ⊑  k x                    x ∈ l
   r ← ret e ;; k  ≡  k e
            Tau k  ≡  k
b ← (a ← e ;; f a);; g b ≡ a ← e ;; b ← f a ;; g b
```

Figure 7. Trace refinement and equivalence for ITrees.

Equivalence and Refinement Intuitively, ITrees that encode the same computation should be considered equivalent. In particular, we want to equate ITrees that agree on their terminal behavior (they return the same value) and on `Vis` events; they may differ by inserting or removing any finite number of `Tau` constructors. This "equivalence up to `Tau`" is a form of weak bisimulation. We write `t ≡ u` when `t` and `u` are equivalent up to `Tau`. The monad laws for ITrees also hold modulo this notion of equivalence. (Some of the laws used in our development are shown in Figure 7.)

ITrees that contain nondeterministic effects or that receive inputs from the environment denote a set of possible *traces*—(finite prefixes of) execution sequences that record each visible event together with the environment's response. The definitions of `trace` and the predicate `is_trace`, which asserts that a trace belongs to an ITree, are shown in Figure 6. Subset inclusion of behaviors gives rise to a natural notion of ITree *refinement*, written `t ⊑ u`, which says that the traces of `t` are a subset of those allowed by `u`. We use this refinement relation to allow an implementation to exhibit fewer behaviors than those permitted by its specification. Note that `t ≡ u` implies `t ⊑ u`.

ITrees as specifications: the linear specification Interaction trees provide a convenient yet rigorous way of formalizing specifications. We have already seen them in the linear specification of the swap server in Figure 3. The `itree specE` type there is an instance of `itree` whose visible events include nondeterministic choice as well as observations of swap request and response messages, which are events that include message content and connection ID information. The specification itself looks like a standard functional program that uses an effects monad to capture network interactions.

ITrees as specifications: the implementation model We use the same `itree` datatype, this time instantiated with a socket API interface included in `implE`, to define the implementation model, which is a lower-level (but still purely functional) specification of the swap server that more closely resembles the C code. Figure 8 shows the body of the main loop from the implementation model.

In contrast to the linear specification, the implementation model maintains a list of connection structures instead of bare connection identifiers. Each structure records the state for some connection. The state indicates whether the server should be `SENDING` or `RECVING` on the connection (or whether the connection is closed). The state also records the contents of receive and send buffers. In each iteration of the loop,

```
Definition select_loop_body
  (server_addr : endpoint_id)
  (buffer_size : Z)
  (server_st : list connection * string)
  : itree implE (bool * (list connection * string)) :=
  let '(conns, last_full_msg) := server_st in
  or
    (r ← accept_connection server_addr ;;
     match r with
     | Some c ⇒ ret (true, (c::conns, last_full_msg))
     | None   ⇒ ret (true, (conns, last_full_msg)) end)
    (let waiting_to_recv :=
         filter (has_conn_state RECVING) conns in
     let waiting_to_send :=
         filter (has_conn_state SENDING) conns in
     c ← choose (waiting_to_recv++waiting_to_send);;
     new_st ← process_conn buffer_size c last_full_msg;;
     let '(c', last_full_msg') := new_st in
     let conns' :=
         replace_when
           (fun x ⇒ if (has_conn_state RECVING x
                        || has_conn_state SENDING x)%bool
              then (conn_id x = conn_id c' ?)
              else false) c' conns in
     ret (true, (conns', last_full_msg'))).
```

Figure 8. Loop body of the implementation model

the server either accepts a new connection or services a connection that is in the `SENDING` or `RECVING` state. Servicing a connection in the `SENDING` state means sending some prefix of the bytes in the send buffer; servicing a connection in the `RECVING` state means receiving some bytes on the connection.

Note that the control flow of this model differs from both the linear specification and the C implementation. The linear specification bundles together request–response pairs and totally abstracts away from the details of buffering and interleaving communications among multiple clients. The relationship between the implementation model and the linear specification is given by *network refinement*, as we explain in the next section. For the C implementation, a single iteration of the main server loop in Figure 5 corresponds to multiple iterations of the select loop body of the model. Nevertheless, we can prove that the C behavior is a refinement of the implementation model, as we describe in Section 5.

4 Network Refinement

We show a "network refinement" relation between the implementation model and the linear specification. At a high level, this property is a form of *observational refinement* [He et al. 1986]: the behaviors of the implementation that can be observed from across the network are included in those of the specification. Intuitively, this property is also an analog, in the network setting, of *linearizability* for concurrent data structures; we compare them in detail in Section 8.

```
Inductive network_event : Type :=
| NewConnection (c : connection_id)
| ToServer      (c : connection_id) (b : byte)
| FromServer    (c : connection_id) (b : byte).

Definition network_trace : Type := list network_event.
```

Figure 9. Types for events and traces observed over the network. `network_event` maps to `event` values to form traces for both the specification and the implementation model.

```
Definition server_transition (ev : network_event)
    (ns ns' : network_state) : Prop :=
  match ev with
  | FromServer c b ⇒ let cs := Map.lookup ns c in
    match connection_status cs with
    | ACCEPTED  ⇒ let cs' := update_out
                      (connection_outbytes cs ++ [b]) cs
                  in ns' = Map.update c cs' ns
    | PENDING | CLOSED ⇒ False end.
  | ... (* Other two cases *) end.

Definition client_transition : network_event →
  network_state → network_state → Prop := ...
```

Figure 10. Network transitions labeled by `network_event`, showing only the case where the server sends a byte.

The network We model a simple subset of the TCP socket interface, where connections carry bytestreams (the bytes sent on an individual connection are ordered); they are bidirectional (both ends can send bytes) and reliable (what is received is a prefix of what was sent). This network model is represented by a nondeterministic state machine where each connection carries a pair of buffers of "in flight" bytes, with labeled transitions for a client to open a connection, a server to accept it, and either party to send and receive bytes (Figures 9 and 10). For example, there is a transition from network state `ns` to state `ns'`, labeled `FromServer c b`, if the connection `c` was previously accepted by the server (its status in `ns` is `ACCEPTED`) and the state `ns'` is obtained from `ns` by adding byte `b` to the outgoing bytes on connection `c`.

We define a relation `network_reordered_ ns ts tc : Prop` between server- and client-side traces of network events `ts` and `tc`, which holds if they can be produced by an execution of the network starting from state `ns`. For the initial state with all connections closed, we define `network_reordered ts tc = network_reordered_ initial_ns ts tc`. The trace `tc` is a "disordering" of `ts`—*i.e.,* `tc` is one possible trace a client may observe if the server generated the trace `ts`. Conversely, `ts` is a "reordering" of `tc`.

Network behavior of ITrees As mentioned in Section 3, ITrees such as the implementation model (of type `itree implE`) and the linear specification (`itree specE`) define sets of event traces. From across the network, those events can appear *disordered* to the client, so the *network behavior* of an ITree is the set of possible disorderings of its traces (defined using `network_reorder`). Finally, the ITree `impl_model` *network refines* the `linear_spec` when the former's network behavior is included in the latter's; see Figure 11.

Proving network refinement In order to prove that our implementation model network refines the linear specification, we establish logical proof rules for a generalization of

```
Definition impl_behavior (impl : itree implE unit) :
    network_trace → Prop :=
  fun tr ⇒ ∃ tr_impl, is_impl_trace impl tr_impl ∧
      network_reordered tr_impl tr.

Definition spec_behavior (spec : itree specE unit) :
    network_trace → Prop :=
  fun tr ⇒ ∃ tr_spec, is_spec_trace spec tr_spec ∧
      network_reordered tr_spec tr.

Definition network_refines impl spec : Prop :=
  ∀ tr, impl_behavior impl tr → spec_behavior spec tr.
```

Figure 11. Definition of network refinement in Coq. The functions `is_impl_trace` and `is_spec_trace` are thin wrappers around `is_trace` that convert between traces of different (but isomorphic) event types.

```
Record state := { get_ns : network_state;
                  get_spec : itree specE unit; ... }.

Definition nrefines_ (z : nat) (s : state)
                     (impl : itree implE unit) : Prop :=
  ∀ tr, is_impl_trace_ z s impl tr →
    ∃ dstr : network_trace,
      network_reordered_ (get_ns s) dstr tr ∧
      is_spec_trace (get_spec s) dstr.
```

Figure 12. Refinement relation generalized for reasoning

`network_refines`, named `nrefines_` (Figure 12). The `nrefines_` relation is step-indexed (`z : nat`) to handle the server's nonterminating loop; it relates a subtree of the implementation model `impl` to a record `s` of the current state of the network (`get_ns s : network_state`) and a subtree of the specification ITree (`get_spec s : itree specE unit`).

Two example proof rules are shown in Figure 13. When the server performs a network operation, for example when it receives a byte on a connection `c`, we use a lemma such as `nrefines_recv_byte_`: we must prove that the connection `c` is open, and we then prove the `nrefines_` relation on the continuation `k b`, with an updated network state in `s'`.

At any point in the proof, we can also generate part of the reordered trace from the linear specification ITree `get_spec s`, using the lemma `nrefines_network_transition_`. We actually use this rule at exactly two "linearization points" in the implementation model: right after the server accepts a new connection, and after it receives a complete message from a client and swaps it with the last stored message.

Using these rules, we prove the proposition `∀ z, nrefines_ z s0 impl_model`, where `s0` is defined so that `get_spec s0 = linear_spec` and `get_ns s0` is the initial network state, where all connections are closed; we can show this implies the second clause of the correctness theorem (Figure 4).

5 Verification

Embedding ITrees in VST VST is a framework for proving separation logic specifications of C programs, based on the C semantics of the CompCert compiler. Its separation logic comes with a proof automation system, Floyd, that supplies tactics for symbolically executing a program while maintaining its pre- and postcondition [Cao et al. 2018]. To support reasoning about external behavior in general—and the swap server's invocations of OS/network primitives in particular—we extend VST's logic with two *abstract predicates* [Penninckx et al. 2015]; these are separation logic predicates that behave like resources but do not have a footprint in concrete memory. Instead they connect to VST's model of *external state*, which in this case represents the allowed network behavior of the program. To make this possible, we made a small modification to the internals of VST to enable it to refer to the external state in assertions.

```
Lemma nrefines_recv_byte_ z s
  (c : connection_id) (k : byte → itree implE unit)
  : In (get_status s c) [PENDING; ACCEPTED] →
    (∀ b s', s' = append_inbytes c [b] s →
             nrefines_ z s' (k b)) →
    nrefines_ z s (b ← recv_byte c;; k b).

Lemma nrefines_network_transition_ z s obs' ns' t
      (dtr : network_trace)
  : (∀ dtr', is_spec_trace obs' dtr' →
             is_spec_trace (get_observer s)
             (dtr ++ dtr')) →
    server_transitions dtr (get_ns s) ns' →
    nrefines_ z (set_ns ns' (set_observer obs' s)) t →
    nrefines_ z s t.
```

Figure 13. Example proof rules for `nrefines_`

The first abstract predicate, `ITree(t)`, injects an interaction tree `t` into a VST assertion (an `mpred`):

```
Definition ITree {R} (t : itree implE R) : mpred :=
  EX t' : itree implE R, !!(t ⊑ t') && has_ext t'.
```

`ITree t` asserts that the observation traces of `t` (*i.e.*, the traces that may be produced by a program satisfying the assertion `ITree t`) are included in the traces that are permitted by the external environment (here, the OS). The `has_ext` predicate asserts that the external state (here representing the network behavior the OS expects from the program) is exactly t'. The notation `!!p` lifts an ordinary Coq predicate `p` to a VST separation logic predicate, and `&&` and `EX` are logical conjunction and existential quantification at the level of separation logic assertions.

While a detailed description of VST's support for external state is beyond the scope of the present paper, we give some key properties of this embedding. Internal code execution does not depend on or alter external state, so every program step that is not a call to the socket API leaves the `ITree` predicate unchanged. The monad and equivalence laws from the abstract theory of interaction trees are reflected as (provable) entailments between `ITree` predicates (recall the refinement relation of Figure 7):

$$\frac{\texttt{t} \sqsubseteq \texttt{u}}{\texttt{ITree u} \vdash \texttt{ITree t}}$$

This rule is *contravariant* because we can conform to the ITree `u` by producing some subset of its allowed behavior.

```
{ SOCKAPI st * ITree t *
  data_at_ alloc_len buf_ptr *
  !! ((r ← recv client_conn (Z.to_nat alloc_len) ;; k r)
       ⊑ t) *
  !! (consistent_world st ∧ lookup_socket st fd =
       ConnectedSocket client_conn) *
  !! (0 ≤ alloc_len ≤ SIZE_MAX) }
ret = recv(int fd, void* buf_ptr, unsigned int
     alloc_len, int flags)
{ ∃ (result : unit + option string) st' ret contents,
  !! (0 ≤ ret ≤ alloc_len ∨ ret = -1) *
  !! (ret > 0 → (∃ msg, result = inr (Some msg) ∧ ...) ∧
       st' = st) *
  !! (ret = 0 → result = inr None ∧ ...) *
  !! (ret < 0 → result = inl tt ∧ ...) *
  !! (Zlength contents = alloc_len) *
  !! (consistent_world st') *
  SOCKAPI st' *
  ITree (match result with
  | inl tt ⇒ t
  | inr msg ⇒ k msg end) *
  data_at alloc_len contents buf_ptr}
```

Figure 14. VST axiom for the `recv` system call.

External calls to network and OS functions are equipped with specifications that reflect the evolution of interaction trees, in resource-consuming fashion: actions are "peeled off" from the ITree as execution proceeds, so that the interaction tree in the postcondition of an external function specification is a subtree of the tree in the precondition. The ITree found in the outermost precondition of a program is thus a sound approximation of all the program's external interactions.

Hoare-logic specifications of system calls This use of the `ITree` predicate can be seen in the VST axiom for the `recv` system call in Figure 14. The precondition of this rule requires that the ITree (`r ←recv client_conn (...);; k r`), which starts with a `recv` event, be among the allowed behaviors of `t`, so a legal implementation of this specification is allowed to perform a `recv` call next. The postcondition either leaves the interaction tree `t` untouched, in the case that the call to `recv` failed, or says that the implementation may continue as `k msg`, in the case that the call to `recv` successfully returned a message `msg`.

Most of the remaining constraints relate the program variables and the variables in the interaction tree to the corresponding state in memory. For example, the predicate `data_at_ alloc_len buf_ptr` says that `buf_ptr` points to a buffer of length `alloc_len`. The constraint `lookup_socket st fd = ConnectedSocket client_conn` says that the socket with identifier `fd` is in the CONNECTED state according to the API and is associated with the connection identifier `client_conn` appearing in the interaction tree.

This socket information is tracked by a second abstract predicate, `SOCKAPI(st)`, which asserts that the external socket API memory can be abstracted as `st`, mapping file descriptors to socket states CLOSED, OPENED, BOUND, LISTENING, or CONNECTED. Bound and listening states are associated with an endpoint identifier in the network model, and connected states are associated with a connection identifier in the network model. The reason for modularly separating socket states from interaction trees is that the latter describe truly external behavior while the former concern the (private) contract between the server program and the OS. Specifically, the functions for creating sockets, binding them to addresses, and closing sockets are not visible at the other end of the network and are hence specified to only operate over `SOCKAPI` abstract predicates. In general, system calls like `recv` that affect the network state carry specifications of the form

```
{ SOCKAPI(st) * ITree (x ← op(a₁, ...); k x) * ... }
op(a1, ...)
{ EX st' t'. SOCKAPI(st') * ITree(t') * ... ∧
  (ϕ(r) → t' = k r) ∧ (¬ϕ(r) → t' = t)}
```

where ϕ is a boolean predicate distinguishing ITree-advancing (successful) invocations from failed invocations (which leave the ITree unmodified), by inspection of the implicitly quantified return value `r`.

Verifying the C implementation Having defined the abstract predicates we need to describe the network behavior of the server, we can now prove that the C implementation refines the implementation model using VST's separation logic. The goal is to prove that the implementation model `impl_model` is an *envelope* around the possible network behaviors of the C program, *i.e.*, every execution of the C program performs only the socket operations described in `impl_model`; this is expressed by the predicate `ext_behavior C_prog impl_model`. This proof then composes with the network refinement proof between `impl_model` and the linear specification to give us the main theorem in Figure 4.

We prove `ext_behavior C_prog impl_model` by specifying and proving a Hoare triple for each function in the C implementation. We begin with axiomatized Hoare triples for the library functions, in particular those from the POSIX socket API; these triples modify the `SOCKAPI` state and possibly consume operations from the `ITree`, as described above.

We then specify Hoare triples for functions in the program, including embedded interaction trees where appropriate. Verification proceeds as in standard Hoare logic, including formulating an appropriate invariant for each loop. The most interesting invariant is for the main loop, shown in Figure 5; among other things, the invariant states that `head` points to a linked list l of connection structures, `last_msg_store` points to a buffer storing a message M, and the interaction tree under `ITree` is an infinite loop of `select_loop_body` (Figure 8)) started on (l, m); the server address and buffer size are constants.

Note that it is not immediate that the C loop body refines `select_loop_body`. The former iterates over all ready connections in `process_connections`, while the latter works on only one connection per iteration. However, each iteration in `process_connections` is itself an iteration of `select_loop_body`, so the inner invariant carries the same interaction tree. Conceptually, one iteration of the main loop in C corresponds to multiple iterations of the model.

6 Testing

Our overall approach to verifying software includes testing for errors in code and specifications before we invest too much effort in verification. For the swap server, we used QuickChick [Lampropoulos and Pierce 2018], a property-based testing tool in Coq, to test both whether the C implementation satisfies the linear specification, and whether the implementation model refines the linear specification. These tests help establish confidence in all three artifacts.

Test setup Our testbed consists of a simple hand-written client, the server to be tested, and the linear specification that the server should satisfy. The client opens multiple TCP connections to simulate multiple clients communicating with the server over the network.

The testing process is straightforward: First, the client generates a random sequence of messages along randomly chosen TCP connections. The client then collects a trace of its interactions with the server—the messages that it sent and the responses that it received in return on each connection. Finally, the checker attempts to "explain" this trace by enumerating all of the possible reorderings of the real trace and checking whether any of them is, in fact, a trace of the linear specification. If such a trace is found, this test case passes, and another trace is generated. If none of the reorderings satisfies the specification, the tester reports that it has found a counterexample. Before actually displaying the counterexample, the tester attempts to *shrink* it using a greedy search process modeled on the one used in Haskell's QuickCheck tool, successively throwing away bits of the counterexample and rechecking to see whether the remainder still fails.

We can also test that the implementation model refines the linear specification. The setup here is similar to the one for the C program, but simpler because we can execute both the client and server within a single Coq program rather than extracting a client from Coq and running it with the server and a network.

Testing the tester The proofs connecting our C implementation, implementation model, and linear specification were well along before we completed the testing framework; this meant that these artifacts were already thoroughly debugged, and testing was not able to find any additional bugs.

To assess how effective testing *might* have been if it had been deployed earlier in the process, we used QuickChick's *mutation testing* mode [DeMillo et al. 1978] to inject 12 different "plausible bugs" (of the sort commonly found in C: pointer errors, bad initialization, off-by-one errors, *etc.*) into the code and check that each could be detected during testing. The bugs are added to the C program as comments marking a section of "good code" and a "mutant" that can be substituted for it. QuickChick performs this substitution for each of the mutants in turn, generates random tests as usual, and reports how many tests it took to find a counterexample for each of the mutants.

We analyzed the running time and number of tests needed to capture the bugs, by repeating QuickChick for 29 times on each mutant. For five of the 12 mutants (changing the initial message buffer, sending extra bytes from the response buffer, responding with wrong message, computing wrong connection state, and skipping the completeness check), the wrong behavior was caught by the very first test in each run. Six of the mutants passed the first test in some runs, but always failed by the second test (sending wrong number of bytes, storing to wrong message buffer, handling partial messages incorrectly, dropping one byte of message, copying response from wrong buffer, and skip populating response). The most interesting mutant was changing the return value of the `recv` call. 3/4 of the runs caught the bug within four rounds, but others took up to nine rounds. This mutant sometimes causes the server not to respond, which is trivially correct because our specification does not deal with liveness. As a result, the tester discarded up to three thousand test cases where the server did not respond, and ran for up to five minutes before failing. The other mutants could fail within 0.4 second with 95% confidence.

It is hard to draw definite conclusions about the effectiveness of testing from a case study of this size, but the fact that we are able to detect a dozen different bugs, most quite quickly, is an encouraging sign that this approach to testing will provide significant value as the codebase and its specification become more complex. Reports in the literature of property-based random testing of similar kinds of systems (*e.g.*, Dropbox [Hughes et al. 2016]) are also encouraging.

7 Connecting to CertiKOS

A key pillar of the proof of correctness of the C implementation is the specification of the socket operations such as `send` and `recv`. We took these specifications as axioms when proving the implementation model, but because we are running the server on top of CertiKOS, which has its own formal specification, we should be able to go one step better: we would like to prove that the socket operations as specified by CertiKOS satisfy the axioms used in the VST proof. This part of the case study is still in progress; we report here on what we've achieved so far and identify the challenges that remain.

The Socket API in CertiKOS CertiKOS provides its own axiomatized specifications for the POSIX socket API. Unlike VST specifications, which are expressed as Hoare triples, CertiKOS specifications are written as state transition functions on the OS abstract state. This state is a record with a

```
Definition recv_spec (fd maxlen : Z) (d : OSData)
  : option (OSData * Z) :=
  let pid := d.(curid) in
  (* Check that the ITree allows this behavior *)
  match ZMap.get pid d.(itrees) with
  | Vis (recv fd' maxlen') k⇒
    if (fd = fd' && maxlen = maxlen') then
      (* Query the oracle for the next network message *)
      match net_oracle (ZMap.get pid d.(net)) with
      | RECV msg⇒
        (* Take up to maxlen bytes *)
        let msg' := prefix maxlen msg in
        let len := length msg'⇒
        (* Update the ITree based on len *)
        let res := if (len > 0) then inr (Some msg')
                   else if (len = 0) then inr None
                   else inl tt in
        let itree' := match res with
          | inl tt⇒ ZMap.get pid d.(itrees)
          | inr msg⇒ k msg end in
        (* Update the OS state and return len *)
        Some (d {itrees: ZMap.set pid itree' d.(itrees)}
                {rbuf: ZMap.set pid msg' d.(rbuf)}
                {net: RECV msg :: d.(net)}, len)
      | _⇒ None end
    else None
  | _⇒ None end.
```

Figure 15. CertiKOS specification of `recv`

field for each piece of real or ghost state that the OS maintains. This includes, for example, buffers for received network messages, or socket statuses. To provide a common language with VST for expressing allowable network communications, we have modified CertiKOS' state to also include an ITree for each user process.

A function like `recv` presents a challenge in that it depends on nondeterministic behavior by the network, but the specification must be a deterministic function. The standard solution used in CertiKOS is to parametrize the specification by an "environment context" [Gu et al. 2018], which acts as a deterministic oracle that takes a log of events and returns the next step taken by the environment. Because the only restriction on the environment context is that it is "valid" (*e.g.*, for networks this could mean that receive events always have a corresponding earlier send event), properties proved about the specifications hold regardless of the particular choice of oracle. Equipped with such a network oracle, the specification of `recv` is fairly straightforward (Figure 15).

Bridging VST and CertiKOS memories The other major gap between VST and CertiKOS is their treatment of memory. Both VST and CertiKOS build on CompCert's memory model to describe the state of memory, but the changes they make to it are unrelated and incompatible. VST builds a step-indexed model on top of CompCert memories [Appel 2014], to allow for "predicates in the heap"-based features, including recursive predicates and lock invariants. Hoare triples are interpreted as assertions on these step-indexed memories. On the other hand, the CompCert model corresponds to virtual memory, and treats independent memory allocations as belonging to separate, nonoverlapping "blocks", while CertiKOS uses a "flat" memory model in which there is only one block to more accurately represent the kernel's view of physical memory. To bridge this gap, we need to translate VST pre- and postconditions into assertions on ordinary, step-index-free CompCert memories (and vice versa), and transform predicates on multiple-block CompCert memories into predicates on CertiKOS's flat memories (and vice versa).

Performing this translation in general is an interesting research problem, but for this application, the specifications to be connected have a very particular form. The pre- and postconditions `send` and `recv` functions are each divided into two parts: a memory assertion on a single buffer, an array of bytes meant to hold the message, and an ITree assertion describing the external network behavior. This simplifies the task of connecting the VST and CertiKOS specs: we just need to relate the interaction tree to some component of the OS state, and translate an assertion on a single piece of memory into the flat memory model and back. (The other socket operations do not involve any changes to user memory, though they do modify kernel memory, which is abstracted to the C program via the **`SOCKAPI`** predicate.)

We have explored this approach by sketching the correspondence between the VST specification of `recv` and its CertiKOS specification. We translated the VST pre- and postcondition for `recv` into step-index-free predicates on CompCert memories and interaction trees by hand, and proved the correctness of the translation using the underlying logic of VST. We then wrote functions that transfer a single block of memory between the CompCert model and the flat model, and adapted the CertiKOS OS component representing the network state to use interaction trees, so that the two systems have a common language to describe network operations. The network component of the CertiKOS OS state is now a map that, for each user process, holds an interaction tree describing the network communication that that process is allowed to perform. Finally, we are in the process of proving that the CertiKOS specification for `recv` satisfies the step-index-free, flattened versions of the VST pre- and postcondition. This gives us a path to validating the axiomatized specifications of the socket API that we rely on for the correctness of the C implementation: they can be substantiated by connection to the (axiomatized) behavior of the socket operations in the underlying operating system.

8 Related Work

Interaction trees As mentioned in Section 3, our "interaction trees" are a Coq-compatible variation of ideas found elsewhere. Kiselyov and Ishii [2015] present a similar concept under the name "freer monad". It is proposed as an improvement over a "free monad" type, which one might hope to define in Coq as follows:

```
Inductive free (E : Type → Type) (R : Type) :=
| Ret : R → free E R
| Vis : E (free E R) → free E R.  (* NOT PERMITTED!! *)
```

Unfortunately, the recursive occurrence of `free` in the `Vis` constructor is not strictly positive, so this definition will be rejected by Coq. Thus in a total language, the choice for the `Vis` constructor to separate the effect `E X` from the continuation `X → itree E R` is largely driven by necessity, whereas the work on freer monads proposes it as a matter of convenience and performance.

The McBride [2015] variant, which builds on earlier work by Hancock [2000], is called the "general monad." It is defined inductively, and its effects interface replaces our single `E : Type → Type` parameter with `S : Type` and a type family `S → Type` to calculate the result type. It was introduced as a way to implement general recursive programs in a total language (Agda), by representing recursive calls as effects (*i.e.*, `Vis` nodes). Our coinductively defined interaction trees also support a general (monadic) fixpoint combinator.

Letan et al. [2018] present the "program monad" to model components of complex computing systems. Like the general monad, it is defined inductively. Whereas our interpretation of ITrees is based on traces, they use a coinductively defined notion of "operational semantics" to provide the context in which to interpret programs, describing the state transitions and results associated with method calls/effects.

Our choice to use coinduction and the `Tau` constructor gives us a way to account for "silent" (internal) computation steps, and hence allows us to semantically distinguish terminating from silently-diverging computations (which is not easy with trace-based semantics, at least not without adding a "diverges" terminal component to some of the traces). Although liveness is explicitly not part of our correctness specification in this project (the spec is conditioned on there being visible output), it is conceivable to strengthen the specifications and account for `Tau` transitions as part of the C semantics, which might allow one to prove liveness properties (although VST does not currently support that). However, there are also costs to working with coinduction: our top-level programs are defined by **`CoFixpoint`**, and coinduction is generally not as easy to use in Coq as it could be [Hur et al. 2013].

Verifying effectful systems A common approach to reasoning about effectful programs is to provide a model of the state of the outside world, with access mediated strictly through external functions. These functions may be given (possibly non-deterministic) semantics directly [Chlipala 2015], or indirectly through an oracle [Férée et al. 2018; Gu et al. 2016]. For example, in Férée et al. [2018], external functions are called through a Foreign Function Interface (FFI), and specification/verification is done with respect to an instantiated FFI oracle that records external calls and defines the state of the environment and the semantics of external functions. In their work, a `TextIO` library was verified with respect to a model of the file system. Similarly, our specifications in terms of Hoare triples assume a model of external socket API memory, *i.e.*, the state under the **`SOCKAPI`** predicate, and describe how this state is transformed.

Stronger specifications of effectful programs can involve dynamics (*"what has happened"*) rather than statics (*"what is the final state"*). In such cases, a model of the external state is commonly extended with (or taken to be) a *trace* or *history* of past events, and specifications involve these traces. Chajed et al. [2018]; Hawblitzel et al. [2015]; Leroy [2009]; Malecha et al. [2011], *etc.* use this approach.

Our specifications are based on interaction trees (which can be construed as sets of traces), with one major difference: interaction trees specify *"what is allowed to happen"*. Rather than reasoning about *lists* of events that have occurred in the *past*, our reasoning is based on the *trees* of events that are allowed to be produced in the *future*. One main advantage of using interaction trees is that it gives us a unifying structure for specification, testing, and verification, as detailed in Section 3. A similar underlying structure to interaction trees is used as specifications of distributed systems in an early version of F* [Swamy et al. 2011], but they did not show how to use them for testing or how to do refinement.

Linearizability Network refinement is closely related to linearizability [Herlihy and Wing 1990], a correctness criterion for concurrent data structures. A data structure implementation is *linearizable* if, for every possible collection of client threads, the behavior of the data structure is indistinguishable from the behavior of a sequential implementation of the structure. Filipovic et al. [2009] related linearizability to contextual refinement. Network refinement is essentially this same idea of contextual refinement, but with network effects playing the role of relaxed memory. Our network model closely resembles TSO, and network refinement is similar to TSO-linearizability [Burckhardt et al. 2012].

Verifying networked servers In one early attempt at server verification, Black [1998] verified security properties of the `thttpd` web server, based on axiomatized C semantics. That work did not establish the functional correctness of the web server, the axiomatic semantics was not testable, and it did not consider the effects of network reordering.

IronFleet [Hawblitzel et al. 2015] is a methodology for verifying distributed system implementations and it is similar to our approach in several ways: both verify the functional correctness of a networked system; both use a "one client at a time" style specification at the top-level; and both verify the correctness of a system implementation which interleaves its operations via linearizability. However, there are several major differences between IronFleet and our work: (1) We are concerned with testing, as it allows us to find implementation bugs early, and it also allows us to use the same specification for blackbox-testing of existing implementations. For these reasons, we choose the executable interaction trees to represent the specification. IronFleet focuses instead on reducing the burden of verification. It uses non-executable state machines, and it relies on tools such as IDEs to support rapid verification. (2) Our work verifies C implementations. VST and CompCert ensure that the properties we have proved at the source-code level are preserved after the program has been compiled to assembly code. IronFleet verifies programs written in Dafny [Leino 2010], and extracts them to C#. This means that both the extraction engine and the .NET compiler must be trusted. The authors of IronFleet also suggest an alternative strategy to reduce the trusted computing base, by first translating the programs to assembly code, and verifying the assembly code using an automatically translated specification [Hawblitzel et al. 2014]. However, that still requires the specification translator to be trusted. (3) IronFleet is based on UDP, while our works is based on TCP. Nevertheless, we both need to consider packet reordering. The difference is that messages will not be reordered on each individual connection. (4) IronFleet uses TLA+ [Lamport 2002] to prove liveness properties. The partial-correctness approach of separation logic makes it more difficult to reason about liveness.

CSPEC [Chajed et al. 2018] is a framework for verifying concurrent software. CSPEC focuses on reducing the number of interleavings a verifier must consider. To do that, it provides a general verification framework built on *mover types* [Lipton 1975]. We may be able to use mover types to simplify the process of proving network refinement.

Verdi [Wilcox et al. 2015] is a framework for verified distributed systems that work under different fault and network models. Verified System Transformers transform a distributed system verified under one model to one that works in another. In particular, the Raft system transformer [Woos et al. 2016] transforms a given state machine (server) into a distributed system of servers that synchronize state using Raft messages, over a network that may drop, reorder, or duplicate messages. Any trace of Raft I/O messages produced by the distributed system can then be linearized to an I/O trace of the input state machine. Distributed systems and transformers are written in Coq and extracted to OCaml.

Ridge [2009] verified the functional correctness and linearizability of a networked, persistent message queue written in OCaml using the HOL4 theorem prover. In contrast to Verdi and Ridge's work, our methodology focuses on testing and verifying C implementations, dealing with the full complexity of low-level programming including memory allocation and pointer aliasing.

For simplicity, our work builds on a small subset of axiomatized TCP specifications. A rigorous and experimentally-validated specification of TCP can be found in Bishop et al. [2005a,b]; Ridge et al. [2009].

Testing There is more research on testing linearizability of concurrent or distributed systems than we can summarize here, including Burckhardt et al. [2010]; Scott et al. [2016]; Shacham et al. [2011]; Vechev et al. [2009]. Our work is distinguished by the focus on uniting testing and verification in the same framework. The QuickChick property-based testing methodology has been shown to be useful in formal verification [Lampropoulos and Pierce 2018]. There are also many accounts of successfully applying property-based random testing to real-world systems. For example, Hughes and Bolinder [2011] used QuickCheck to test for race conditions in dets, a vital component of the Mnesia distributed database system; Arts et al. [2015] have applied the methodology to test the AUTOSAR Basic Software for Volvo Cars; and Hughes et al. [2016] tested the linearizability of Dropbox, the distributed synchronization service.

9 Conclusions and Future Work

Starting from a C implementation and a "one client at a time" specification of swap server behavior, we have proved that every execution of the implementation correctly follows the specification. The proof breaks down into layers of refinements: from the C program to an implementation-level interaction tree, and from there, via *network refinement* to the linear interaction tree. We use VST to verify the C code, pure Coq to relate the trees, QuickChick to test our specifications and implementations, and CertiKOS to validate our specifications of network communication. The result is a proof of the correctness of the swap server from the linear specification down to the interface between the C program and the operating system.

Although this work represents significant progress towards the Deep Specification project's goal of formally-verified systems software, much remains to be done. The verification of the swap server has tested the limits of VST, in terms of both scale and style of specifications. Previous VST verifications were self-contained libraries, but this swap server interacts with the OS through the socket API, requiring us to develop new features (the external assertions) that should be useful for verifying a variety of more realistic programs.

A clear next step is to fully verify the socket API used by the server, by completing the proof that each VST socket axiom follows from the specification of the corresponding operation in CertiKOS. Doing so will require several more proofs along the lines of our verification of `recv`, bridging the gap between VST's step-indexed memory and CertiKOS's flat memory, as well as defining a suitable C-level abstraction of the kernel memory related to the socket operations. This will further extend the reach of our result, so that we rely only on the correctness of the operating system's model of the socket API.

References

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers.* Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017). https://doi.org/10.1098/rsta.2016.0331

Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015.* 1–4. https://doi.org/10.1109/ICSTW.2015.7107466

M. Belshe, R. peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. RFC Editor. http://www.rfc-editor.org/rfc/rfc7540.txt

Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005a. *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 1: Overview.* Technical Report UCAM-CL-TR-624. Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-624.html 88pp.

Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005b. *TCP, UDP, and Sockets: rigorous and experimentally-validated behavioural specification. Volume 2: The Specification.* Technical Report UCAM-CL-TR-625. Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-625.html 386pp.

Paul E. Black. 1998. *Axiomatic Semantics Verification of a Secure Web Server.* Ph.D. Dissertation. Provo, UT, USA. AAI9820483.

Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010.* 330–340. https://doi.org/10.1145/1806596.1806634

Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. 2012. Concurrent Library Correctness on the TSO Memory Model. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* 87–107. https://doi.org/10.1007/978-3-642-28869-2_5

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reasoning* 61, 1-4 (2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying a concurrent mail server with CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/osdi18/presentation/chajed

Adam Chlipala. 2015. From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 609–622. https://doi.org/10.1145/2676726.2677003

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* 268–279. https://doi.org/10.1145/351240.351266

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O Myreen, and Son Ho. 2018. Program Verification in the Presence of I/O: Semantics, verified library routines, and verified applications. In *10th Working Conference on Verified Software: Theories, Tools, and Experiments.*

Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2009. Abstraction for Concurrent Objects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings.* 252–266. https://doi.org/10.1007/978-3-642-00590-9_19

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* 646–661. https://doi.org/10.1145/3192366.3192381

Peter Hancock. 2000. *Ordinals and interactive programs.* Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/376

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* 1–17. https://doi.org/10.1145/2815400.2815428

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 165–181. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. 1986. Data Refinement Refined. In *ESOP 86, European Symposium on Programming, Saarbrücken, Federal Republic of Germany, March 17-19, 1986, Proceedings.* 187–196. https://doi.org/10.1007/3-540-16442-1_14

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016.* 135–145. https://doi.org/10.1109/ICST.2016.37

John M. Hughes and Hans Bolinder. 2011. Testing a database for race conditions with QuickCheck. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011.* 72–77. https://doi.org/10.1145/2034654.2034667

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13).* ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015.* 94–105. https://doi.org/10.1145/2804302.2804319

Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq.* Electronic textbook. https://softwarefoundations.cis.upenn.edu/qc-current/index.html

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721. https://doi.org/10.1145/361227.361234

Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. 2011. Trace-based Verification of Imperative Programs with I/O. *J. Symb. Comput.* 46, 2 (Feb. 2011), 95–118. https://doi.org/10.1016/j.jsc.2010.08.004

Coq development team. 2018. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.8.1.

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*. 199–212. http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 158–182. https://doi.org/10.1007/978-3-662-46669-8_7

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Thomas Ridge. 2009. Verifying distributed systems: the operational approach. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. 429–440. https://doi.org/10.1145/1480881.1480934

Thomas Ridge, Michael Norrish, and Peter Sewell. 2009. *TCP, UDP, and Sockets: Volume 3: The Service-level Specification*. Technical Report UCAM-CL-TR-742. University of Cambridge, Computer Laboratory. 305pp.

Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George C. Necula, Arvind Krishnamurthy, and Scott Shenker. 2016. Minimizing Faulty Executions of Distributed Systems. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*. 291–309. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/scott

Ohad Shacham, Nathan Grasso Bronson, Alex Aiken, Mooly Sagiv, Martin T. Vechev, and Eran Yahav. 2011. Testing atomicity of composed concurrent operations. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 51–64. https://doi.org/10.1145/2048066.2048073

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 266–278. https://doi.org/10.1145/2034773.2034811

Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*. 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368. https://doi.org/10.1145/2737924.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*. 154–165. https://doi.org/10.1145/2854065.2854081