From C to Interaction Trees
Specifying, Verifying, and Testing a Networked Server

Nicolas Koh
University of Pennsylvania
Philadelphia, PA, USA

Yao Li
University of Pennsylvania
Philadelphia, PA, USA

Yishuai Li
University of Pennsylvania
Philadelphia, PA, USA

Li-yao Xia
University of Pennsylvania
Philadelphia, PA, USA

Lennart Beringer
Princeton University
Princeton, NJ, USA

Wolf Honoré
Yale University
New Haven, CT, USA

William Mansky
University of Illinois at Chicago
Chicago, IL, USA

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA

Steve Zdancewic
University of Pennsylvania
Philadelphia, PA, USA

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.

CCS Concepts → Software and its engineering → Software verification; Formal software verification;

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 “from internet RFCs down to transistors” of real-world systems such as web servers. 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 allowed by the linear specification. We also show how property-based random testing using Coq’s QuickChick plugin [Lampropoulos and Pierce 2018] can be deployed in this setting, both for detecting disagreements between the implementation and specification and for validating the specification itself against legacy servers. 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.

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 we leave completing the formal proofs as 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.

## 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

![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 socket interface; the axioms in VST, in turn, are validated by a lower-level axiomatization in the style of CertiKOS, which is manually compared to the (unverified) TCP implementation. The implementation model “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.](image)

![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 most recently 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. The network socket may send one message in multiple chunks, messages from different clients may be received in any order, and messages may be delayed indefinitely (dotted arrow). The “explanation” of the two runs in terms of the linear specification is the same.](image)
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.

This theorem constrains the implementation 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 the server 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 also prevents the server from sending a message from a client over some established connection, or doing a receive and then a send on some existing connection picked in the list conns. The linear specification is initialized with an empty set of connections and a message filled with zeros.

The left-hand side of Figure 2 shows a simple example of correct behavior of a swap server.

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). We will discuss the types used here in more details in Section 3.

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.

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.
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 main body of the server is a non-terminating loop (Figure 5); in each iteration, it uses the `select` system call\(^1\) 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.

Our C code is compiled by CompCert and should run on any operating system with POSIX sockets. We have tested it on CertiKOS, OSX, and Linux; our long-term aim is deeper integration with CertiKOS’s own formal verification.

**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 library functions like `memset` and the `fdset` macros. We rely on the soundness of the Coq proof assistant, plus the standard axioms of functional and propositional extensionality and proof irrelevance [Coq development team 2017].

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, 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 plugin 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.

Supporting property-based testing requires executable specifications of the properties involved. Happily, interaction trees, which play a crucial role throughout our development,
From C to Interaction Trees

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.

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 environments 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 environment. 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 [Chlipala 2017]:

Cofixpoint spin (E R) : itree E R := Tau spin.

The final, and most interesting, way to build 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 environment) and X is the type of data that the environment 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 environment. Vis creates branches in 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 [Moggi 1989; Wadler 1992], which makes it convenient to structure effectful computations using the conventions and notations of functional programming. We wrap the Ret constructor as a ret (return) function 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.

One could, of course, simply consider such an extracted implementation to be the final artifact (as in, for example, Verdi [Wilcox et al. 2015]). However, we are interested in a verified C implementation for two main reasons. First, extracting Coq to OCaml generally involves a certain amount...
The definitions of trace verified. Moreover, the extracted code relies on OCaml’s run-
alleviate at least some of these concerns. 2

that uses an effect monad to capture network interactions.

specification itself looks like a standard functional program
swap request and response messages, which are events that
include nondeterministic choice as well as observations of
linear specification of the swap server in Figure 3. The
action trees provide a convenient yet rigorous way of for-
ing directly in a low-level imperative language that may, in
the long run, be important for our eventual goal of verifying
a high-performance web-server.

Equivalence and Refinement Intuitively, ITrees that en-
code 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 re-
cieve 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 Fig-
ure 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 for-
malizing 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 effect monad to capture network interactions.

2Compiling directly to native code using CertiCoq [Anand et al. 2017] would alleviate at least some of these concerns.

Definition select_loop_body

Figure 7. Trace refinement and equivalence for ITrees.

Figure 8. Loop body of the implementation model.

ITrees as specifications: the implementation model We use the same itree datatype, this time instantiated with an
event type implE which contains nondeterministic choice and
a networking interface (e.g., accept, send, recv), 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 send and receive buffers. In each iteration of the loop,
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 lin-
ear 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. Neverthe-
less, we can prove that the C behavior is a refinement of
the implementation model, as we describe in Section 5.
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.

The network We model a simple subset of the TCP socket interface, where connections carry byte streams (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, 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_reorderd 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_reorderd ts tc = network_reorderd_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.

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.

The network We model a simple subset of the TCP socket interface, where connections carry byte streams (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, 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_reorderd 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_reorderd ts tc = network_reorderd_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 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 c: 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 a part of the reordered trace from the linear specification ITree get_spec s, using the nrefines_network_transition lemma.
Lemma nrefines_recv_byte_z (
(c : connection_id) (k : byte → itree implE unit)
: In (get_status s c) [PENDING; ACCEPTED] →
∀: (V b s', s' = append_inbytes c [b]) s →
nrefines_z z s (b ← recv_byte c;; k b).

Lemma nrefines_network_transition_z s spec' ns' impl
(dtr : network_trace)
: (V dtr', is_spec_trace spec' dtr' →
is_spec_trace (get_spec s)
(dtr ++ dtr')) s →
server_transitions dtr (get_ns s) ns' →
nrefines_z (set_ns ns' (set_spec spec' s)) impl →
nrefines_z z s impl.

Figure 13. Example proof rules for nrefines_z.

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 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.

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 !!! 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 and will be reported elsewhere, 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):

\[
\begin{align*}
\text{SOCKAPI } &\text{ st } \times \text{ ITree } \text{ t } * \\
\text{data_at } \text{ alloc_len } \text{ buf_ptr } &\text{ * } \\
\text{ret } = \text{ recv(int fd, void* buf_ptr, unsigned int alloc_len, int flags)} &\text{ * } \\
&\text{ { } result : unit } + \text{ option string } \text{ st’ ret } \text{ contents, } \\
&\text{ (consistent_world st } \land \text{ lookup_socket st } \text{ fd } = \text{ ConnectedSocket client_conn) } \\
&\text{ (consistent_world st } \land \text{ alloc_len } \leq \text{ SIZE_MAX) } \\
&\text{ ret } = \text{ recv(int fd, void* buf_ptr, unsigned int alloc_len, int flags)} \\
&\text{ { } result : unit } + \text{ option string } \text{ st’ ret } \text{ contents, } \\
&\text{ (consistent_world st’ ) } \\
&\text{ consistent_world st’ } \\
&\text{ ITree (match result with } \\
&\text{ | inl tt }\Rightarrow t \\
&\text{ | inr msg }\Rightarrow k \text{ msg end) } \\
&\text{ data_at alloc_len contents buf_ptr }
\end{align*}
\]

Figure 14. VST axiom for the recv system call.

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

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 \( k \text{msg} \), in the case that the call to \texttt{recv} successfully returned a message \texttt{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 \texttt{data_at alloc len buf ptr} says that \texttt{buf ptr} points to a buffer of length \texttt{alloc len}. The constraint \texttt{lookup socket st fd = ConnectedSocket client conn} says that the socket with identifier \texttt{fd} is in the \texttt{CONNECTED} state according to the API and is associated with the connection identifier \texttt{client conn} appearing in the interaction tree.

This socket information is tracked by a second abstract predicate, \texttt{SOCKAPI(st)}, which asserts that the external socket API memory can be abstracted as \texttt{st}, mapping file descriptors to socket states \texttt{CLOSED}, \texttt{OPENED}, \texttt{BOUND}, \texttt{LISTENING}, or \texttt{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 (after shutdown) are not visible at the other end of the network and are hence specified to only operate over \texttt{SOCKAPI} abstract predicates. In general, system calls like \texttt{recv} that affect the network state carry specifications of the form

\[
\{(\texttt{SOCKAPI}(st) \times \texttt{ITree})(x \leftarrow \texttt{op}(a_1, \ldots); k \times) \times \ldots\}
\]

\[
\text{op}(a_1, \ldots)
\]

\[
\{(\texttt{EX st t' \text{SOCKAPI}(st')} \times \texttt{ITree}(t')) \times \ldots \land (\phi(r) \rightarrow t' = k \times) \land (\neg \phi(r) \rightarrow t' = t))\}
\]

where \( \phi \) is a boolean predicate distinguishing \texttt{ITree}-advancing (successful) invocations from failed invocations (which leave the \texttt{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 now prove that the implementation model \texttt{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 \texttt{impl model}; this is expressed by the predicate \texttt{ext behavior C prog impl model}. This proof then composes with the network refinement proof between \texttt{impl model} and the linear specification to give us the main theorem in Figure 4.

We prove \texttt{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 \texttt{SOCKAPI} state and possibly consume operations from the \texttt{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 \texttt{head} points to a linked list \texttt{l} of connection structures, \texttt{last msg store} points to a buffer storing a message \texttt{m}, and the interaction tree under \texttt{ITree} is an infinite loop of \texttt{select loop body} (Figure 8) started on \((1, m)\); the server address and buffer size are constants.

Note that it is not immediate that the C loop body refines \texttt{select loop body}. The former iterates over all ready connections in \texttt{process connections}, while the latter works on only one connection per iteration. However, each iteration in \texttt{process connections} is itself an iteration of \texttt{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 the possible reorderings of this 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 \texttt{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** Although we did not find any bugs, we assessed the effectiveness of testing using 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, 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. 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 is fairly straightforward (Figure 15).

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 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` 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.
The network component of the CertiKOS OS state is now receiving that the CertiKOS specification for allowed to perform. Finally, we are in the process of describing the network communication that that process is a map that, for each user process, holds an interaction tree have a common language to describe network operations.

We then wrote functions that transfer a single block of CertiKOS memories and interaction trees by hand, and proved and adapted the CertiKOS OS component representing the of memory between the CompCert model and the flat model, of VST. We then wrote functions that transfer a single block

\[ \text{recv} \]

SOCKAPI operations do not involve any changes to user memory, though 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 \text{recv} and its CertiKOS specification. We translated the VST pre- and post-condition for \text{recv} into step-index-free, flattened versions of the VST pre- and post-condition. 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:

\[ \text{Inductive free (E : Type → Type) (R : Type) :=} \]

\[ \text{Ret : R → free E R} \]

\[ \text{Vis : E (free E R) → free E R. (∗ NOT PERMITTED!! ∗)} \]

Unfortunately, the recursive occurrence of \text{free} in the \text{Vis} constructor is not strictly positive, so this definition will be rejected by Coq. Thus in a total language, the choice for the \text{Vis} constructor to separate the effect \text{E} \text{X} from the continuation \text{X} → \text{iTree} \text{E} \text{R} is largely driven by necessity, whereas the work on free 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 \text{E} : \text{Type} → \text{Type} parameter with \text{S} : \text{Type} and a type family \text{S} → \text{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., \text{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 \text{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 \text{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 \texttt{CoFixpoint}, and coinduction is generally not as easy to use in Coq as it could be [Chlipala 2017; 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 [Féréé et al. 2018; Gu et al. 2015]. For example, in Féréé 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 \texttt{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 that work did not show how to use the structure for testing or how to do refinement. Gu et al. [2018] use environment contexts to specify past events as well as future events, but rather than starting with all possible traces and consuming them, valid traces are generated one event at a time by consulting an oracle. Although using this step-based approach instead of explicitly coinductive ITrees leads to different specification styles, it is possible to connect them as we discussed in Section 7.

**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 \texttt{httpd} 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 tool support such as near-real-time IDE-integrated feedback for 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] and 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 its focus on uniting testing and verification in the same framework. QuickCheck’s property-based testing methodology has been shown to be useful in formal verification [Bulwahn 2012; 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] have 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 toward 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. The scale of this project forced us to debug and streamline VST’s existing automation.

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.

Many real-world web servers are multi-threaded, handling requests from different clients in separate threads. Some parts of our approach are already able to handle concurrency: the top-level specification ITree should be sequential regardless of the implementation, and VST and CertiKOS already support concurrent C programs [Gu et al. 2018; Mansky et al. 2017]. Other parts will require adjustment: for instance, the implementation model may need to explicitly represent the concurrency allowed in the C program.

Acknowledgements
This work was funded by the National Science Foundation’s Expedition in Computing The Science of Deep Specification under the awards 1521602 (Appel), 1521539 (Weirich, Zdancewic, Pierce), and 1521523 (Shao), with additional support by the NSF projects Verified High Performance Data Structure Implementations, award 1005849 (Beringer, Mansky), and Random Testing for Language Design, award 1421243 (Pierce). We are grateful to all the members of the DeepSpec project for their collaboration and feedback, and we greatly appreciate the reviewers’ comments and suggestions.

References