Verifying Distributed Systems: the Operational Approach

Thomas Ridge
University of Cambridge
thomas.ridge@cl.cam.ac.uk

Abstract

This work develops an integrated approach to the verification of behaviourally rich programs, founded directly on operational semantics. The power of the approach is demonstrated with a state-of-the-art verification of a core piece of distributed infrastructure, involving networking, a filesystem, and concurrent OCaml code. The formalization is in higher-order logic and proof support is provided by the HOL4 theorem prover.

Difficult verification problems demand a wide range of techniques. Here these include ground and symbolic evaluation, local reasoning, separation, invariants, Hoare-style assertional reasoning, rely/guarantee, inductive reasoning about protocol correctness, multiple refinement, and linearizability. While each of these techniques is useful in isolation, they are even more so in combination. The first contribution of this paper is to present the operational approach and describe how existing techniques, including all those mentioned above, may be cleanly and precisely integrated in this setting.

The second contribution is to show how to combine verifications of individual library functions with arbitrary and unknown user code in a compositional manner, focusing on the problems of private state and encapsulation.

The third contribution is the example verification itself. The infrastructure must behave correctly under arbitrary patterns of host and network failure, whilst for performance reasons the code also includes data races on shared state. Both features make the verification particularly challenging.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Distributed programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Performance, reliability, theory, verification.

Keywords Distributed, infrastructure, persistent queue, operational semantics, HOL, OCaml, network protocol, ground and symbolic evaluation, local reasoning, separation, invariants, Hoare-style assertions, rely/guarantee, inductive reasoning, refinement, linearizability.

1. Introduction

The verified computing stack is gradually becoming a reality. At the bottom of the stack, (partial) processor verification is routine. Higher up, verified operating systems and compilers are emerging. However, at the top of the stack, there is a huge gap between abstract mathematical models of programs and implementations in real code. Distributed infrastructure exemplifies this gap. The idealised models of distributed components may be reasonably clean, but implementations, forced to contend with failing hosts, failing network connections etc, while retaining good performance, are often significantly more complex. Formal techniques are rarely capable of addressing the full complexity of such implementations. This paper describes the successful application of operational methods to mechanically verify a core piece of distributed infrastructure. This sets a new high-water-mark for the verification of executable code in a behaviourally rich language and environment, demonstrating that such verification is feasible. At the same time it establishes a challenge to make such proofs more automatic, and to develop verification techniques that can address even richer programs.

The distributed infrastructure in question is a persistent message queue [29]. Persistent message queues provide reliable message delivery in the presence of host and network failure. As such, they are a core piece of enterprise computing infrastructure and are widely deployed in many large companies. They are included in several enterprise application stacks, such as J2EE [8], and marketed by competing vendors such as TIBCO and IBM.

The implementation discussed here was written by the author to exhibit many of the issues that would be expected in a production implementation, albeit on a somewhat smaller scale. Its performance is reasonably good and competitive with other implementations. Excerpts from the code are given in Section 2.

To verify such code, one needs a formal model of the system. This includes a model of the implementation language or languages (in this case OCaml), a model of a live host (in this case including mutexes, condition variables, the store, network connections, and the filesystem), and a model of the network, including hosts which may crash and network connections which may fail.

Section 3 describes these models. In this work, all models are expressed using operational semantics, as popularized by Plotkin [24, 23]. The models build on much previous work, both formal and informal, including the model of OCaml in higher-order logic by Owens [18]; the very detailed model of TCP/IP, also in higher-order logic, by the NetSem team (which includes the current author) [2, 3, 27]; a previous operational verification of OCaml code by the author [26]; and POSIX manuals and OpenGroup specifications [1].

A key choice was to model the code directly, as a program in an executable programming language equipped with an operational semantics, instead of some idealised state-machine or algorithm. Thus, the executable code is verified, subject only to the assumptions that the compiler is correct with respect to the operational semantics, and that the models of the filesystem, host and network are accurate.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-379-2/09/01…$5.00.

Section 4 contains a brief overview of the verification, which sets the scene for the later sections. The verification involves reasoning directly about the operational semantics, rather than the more usual approach via a program logic, primarily for the pragmatic reason that operational reasoning can support all the techniques that were needed for the verification. The overhead of using these techniques is very low: one can omit the separate definition of a program logic and the accompanying soundness proof and instead work within the flexible environment of higher-order logic. Moreover, the uniform foundation allows these techniques to be integrated together cleanly, as described later.

In Section 5 the specification of the persistent queue is presented. This specification is further refined in Section 6. The queue is not a whole program that runs in its own process, but a concurrent library intended for use by user applications. The infrastructure must perform correctly whatever actions the user application performs. The specification therefore needs to address library correctness in arbitrary context, which is a form of compositionality. The infrastructure is distributed, so the specification needs to detail what happens when other hosts crash or the network fails. The specification crucially also needs to support proofs of correctness of user applications which use the library, since this is one of the main purposes of a specification. The specification establishes complex properties far removed from simple generic safety properties such as memory safety.

To prove that the implementation meets the specification involves several existing techniques, which need to be integrated and used in combination. In Section 7 I recall these existing techniques, show how they are expressed formally in an operational setting (as lemmas and proof idioms about the operational semantics rather than as proof rules in some program logic), and most importantly show how they are integrated together to provide strong support for verification. In Section 8 a particularly interesting part of the correctness proof involving data races on shared state is discussed.

In Section 9 several new techniques are developed, which capture intuitive notions of privacy and encapsulation. These are used when reasoning about the composition of the queue library code (which involves private state shared between several library functions) and arbitrary user code. These notions are simple and general: several of the main lemmas are applicable to arbitrary OCaml code, and can therefore be reused in future verifications.

The verifications of the queue API methods are independent of each other, and these verifications are composed with a verification of arbitrary user code to give a correctness proof for the whole system. This process of composition is described in Section 10. References to related work are included in the body of the paper, and further references appear in Section 11. Finally I evaluate what has been achieved, draw conclusions, and look to the future in Section 12.

Reasoning directly about the operational semantics is only feasible with machine assistance. Most of the arguments described in this paper, including the refinement of the abstract queue to the alternating bit protocol, the verification of the individual OCaml functions, and the verification of the privacy metatheory, have been mechanized in HOL4. The part of the proof that deals with host and network failure at the implementation level has so far not been been mechanized, although these behaviours are dealt with at the intermediate level of the alternating bit protocol.

2. Implementation code

The queue is written in OCaml and makes use of the OCaml Unix and Thread libraries. The signature for the queue is given in Fig. 1, the code shared between sender and receiver in Fig. 2, and the code for the sender in Fig. 3. The code for the receiver is similar to that for the sender and so is omitted.

```
type filename = string
type ip = Unix.inet_addr
type port = int
type quad = ip*port*ip*port

type rqueue = ...
type squeue = ...

val listen    : quad → filename → rqueue
val available : rqueue → bool
val peek      : rqueue → string
val remove    : rqueue → unit
val connect   : quad → filename → squeue
val send      : squeue → string → unit
```

Figure 1. Queue signature (OCaml)

```
type queue = {
  (* shared state *)
  lock : Mutex.t ;
  cond : Condition.t ;
  msgs : (string list) ref ;
  (* active thread local state *)
  b    : bool ref ;
  fd   : Message.conn option ref ;
  (* constant *)
  quad : quad ;
  fn   : string ;
}

let mk_queue quad fn is_sender = {
  lock = Mutex.create () ;
  cond = Condition.create () ;
  msgs = ref [] ;
  b    = ref is_sender ;
  fd   = ref None ;
  quad = quad ;
  fn   = fn ;
} in

let init q =
  let _ = Mutex.lock q.lock in
  let _ =
    try
      let ( b :: msgs ) = File.read q.fn in
      let _ = q.b := bool_of_string b in
      let _ = q.msgs := msgs in
      ()
    with _ → () in
  let _ = Mutex.unlock q.lock in
  () in

let save q =
  let b = string_of_bool ( ! ( q.b ) ) in
  try
    File.write q.fn ( b :: ( ! ( q.msgs ) ) ) ; None
  with
  | File.Exception → Some File.Exception in ...
```

Figure 2. Queue shared sender/receiver code (OCaml)

The queue endpoints communicate over TCP/IP, using a protocol based on the alternating bit protocol, and log their state to persistent store. To abstract slightly from the details of TCP/IP and the filesystem, we use two libraries for messaging (`Message`) and file access (`File`). The messaging library allows communication using strings rather than a byte stream. The file library allows atomic file update by first writing to a temporary file and then renaming to the real target (POSIX-compliant filesystems provide atomic file rename). Although small, these libraries are written above rather complex APIs, so that their correctness is far from obvious. Ideally they should also be verified, but in this work their correctness is assumed.

```
let private_send q =
  let _ = Mutex.lock q.lock in
  let _ =
    while ! ( q.msgs ) = [] do
      Condition.wait q.cond q.lock
    done
  in
  let _ = Mutex.unlock q.lock in
  let msgs = [ string_of_bool ( ! ( q.b ) ) ;
                List.hd ( ! ( q.msgs ) ) ] in
  let _ = Message.send ( dest_Some ( ! ( q.fd ) ) ) msgs in
  () in

let private_recv q =
  let msg = List.hd ( Message.recv ( dest_Some ( ! ( q.fd ) ) ) ) in
  if ! ( q.b )  = bool_of_string msg then (
    let _ = Mutex.lock q.lock in
    let _ = q.msgs := List.tl ( ! ( q.msgs ) ) in
    let _ = q.b := not ( ! ( q.b ) )  in
    let e = save q in
    let _ = Mutex.unlock q.lock in
    maybe_raise e
   ) else () in

let sender q =
  while true do
    try
      q.fd := Some ( Message.connect q.quad ) ;
      while true do
        private_send q ;
        private_recv q
      done
    with
    | Message.Exception → (
        match ! ( q.fd ) with
        | None → ()
        | Some fd → ( Message.close_noerr fd ; q.fd := None ) )
    | e → ( raise e )
  done in

let _connect quad fn =
  let q = mk_queue quad fn true in
  let _ = init q in
  let _ = Thread.create sender q in
  q in

let _send q s =
  let _ = Mutex.lock q.lock in
  let _ = q.msgs := ( ( ! ( q.msgs ) ) @ [ s ] ) in
  let e = save q in
  let _ = Mutex.unlock q.lock in
  let _ = Condition.broadcast q.cond in
  let _ = maybe_raise e in
  () in

let connect quad fn =
  let q = _connect quad fn in
  _send q in

let send q s = q s in ...
```

Figure 3. Queue sender code (OCaml)

A queue endpoint is created when the user calls `listen` or `connect`. In both cases, the user supplies a quad, identifying the local and remote endpoint addresses, and the filename of the log file used to store persistent information about the endpoint state. In the case of `connect`, a call is made to the auxiliary function `_connect`, which creates and initializes the queue and then starts up the active management thread which runs the `sender` function. The thread sits in an outer loop, initializing and reinitializing the TCP/IP connection to the other endpoint, and then running an inner loop until an exception is raised. On each iteration of the inner loop, the sender sends the first pending message to the other endpoint (`private_send`) then waits for the subsequent acknowledgement (`private_recv`).

```
'a exp =
  Wrap of 'a
  | Var of var
  | Lam of (var # exp)
  | App of (exp # exp)
  | LetVal of (var # exp # exp)
  | ...

expr = unit exp

closure = Cl of expr#((var#closure)list)

hole_or_clo = Hole | Clo of closure

context = hole_or_clo exp

framestack = context list
```

Figure 4. Core OCaml datatypes (HOL)

The queue is asynchronous: a call to `connect` may return a queue before any network communication has taken place, whilst a call to `send` affects only the local endpoint. The active management thread handles all communication with the other endpoint.

3. Formal models

The correctness of the queue is a formal statement in higher-order logic. Before this statement can be constructed, the various parts of the system must be formally defined. At the heart of the model is an operational semantics for a pure OCaml expression. This is extended to a model of arbitrarily many individual threads executing in the context of a host. Threads can create other threads dynamically. Threads share access to a store, mutexes and condition variables, a filesystem, and a set of network connections. At the next level up, a network consists of many hosts communicating using messages sent over TCP/IP. To capture the transient nature of hosts and network connections, the model allows them to fail at any time, although host filesystems persist. All code excerpts from now on are written in the HOL4 syntax of higher-order logic, which is similar to the syntax of OCaml. The pair type constructor is written `#`. Finite map update is written `FUPDATE f (arg,result)` or alternatively `f |+ (arg,result)`. Records are written `<| fld:=val; fld':=val' |>`. List append is written `xs++ys`. Logical negation is written as a tilde.

OCaml expressions Previous work by the author [26] used flat expressions, whereas here the operational semantics for OCaml is based closely on the CEK machine [5], which uses closures and a framestack. The use of closures and a framestack introduces more structure into the representation of program state, which is helpful for verification. For example, substitution instances of a function are easier to identify because, using a closure representation, the body of the function remains constant. The fragment of OCaml that is modelled is sufficient to express the implementation code given previously. The most important omission is the OCaml module language.

The HOL datatype for a core subset of OCaml expressions `exp` is given in Fig. 4. If we ignore the `Wrap` constructor, this gives a standard "flat" datatype for expressions. The `Wrap` constructor is used to model `closures`, `contexts`, and `framestacks`.

The reduction rules for a closure `cl` in a framestack `fs` are given in Fig. 5. A thread is a pair where the second component is a framestack, and the first component is either the currently executing closure or a blocking system call. System calls are the interface between threads and the rest of the host. System calls include those related to the store (eg `SC_Assign`), locks and condition variables (eg `SC_Mutex_Lock`), the filesystem (eg `SC_File_Write`), and network

```
reduce_raise (cl,fs) =
  case fs of [] → NONE || f::fs → (
  case f of
    TryWith(Wrap(Hole),Wrap(Clo(cl2))) → (
        case push_env cl of Raise(Wrap(cl)) →
            SOME(cl2,App(Wrap(Hole),Wrap(Clo cl))::fs)
            || _ → NONE)
    || _ → SOME(cl,fs))

reduce_nonval (cl,fs) =
  let add_fs = λ (cl,f). (cl,f::fs) in
  let new_cl_fs push_env_cl = case push_env_cl of
    App (Wrap(cl1),Wrap(cl2)) →
        SOME(cl2, App(Wrap(Clo(cl1)),Wrap Hole))
    || LetVal (x,Wrap(cl1),Wrap(cl2)) →
        SOME(cl1, LetVal(x,Wrap Hole,Wrap(Clo(cl2))))
    || While (Wrap(cl1),Wrap(cl2)) → (
        let cl' = mk_cl (LetVal("_",Var"do",Var"while"))
          [("do",cl2);("while",cl)] in
        SOME(cl1,IfThenElse(Wrap Hole,Wrap(Clo(cl')),
                            Wrap(Clo unit))))
    || ...
  in
  ...

reduce_val (cl,fs) = ...

reduce (cl,fs) =
  let e = cl_to_e cl in
  if is_Raise e then reduce_raise (cl,fs)
  else if is_val e then reduce_val (cl,fs)
  else reduce_nonval (cl,fs)
```

Figure 5. OCaml reduction (HOL)

```
host = <|
  cs : (connectionid,connection)finite_map;
  ts : (threadid,thread)finite_map;
  s : store;
  m : (mutexid,threadid option)finite_map;
  w : condition set;
  f : filesystem
|>
```

Figure 6. Host type (HOL)

communication (eg SC_Listen). Finally the trans_t function ties these components together to give the transitions for a thread.

```
trans_t t = case (t:thread) of
  (T_Run(cl),fs) → (case dest_Call (cl_to_e cl,fs) of
      NONE → (OPTION_MAP (λ (cl,fs). (T_Run(cl),fs))
                (reduce (cl,fs)))
      || SOME(call,fs) → SOME(T_Block(call),fs))
  || (T_Block(call),fs) → failwith NONE "trans_t"
```

The host In Fig. 6 the type host includes threads, a store, a set of mutexes and condition variables, a filesystem, and a set of network connections. The behaviour of the host is also defined using small-step operational semantics; an excerpt describing mutex transitions appears in Fig. 7.

The network A network consists of hosts communicating using TCP/IP. As mentioned previously, the model assumes a thin messaging layer on top of TCP/IP that allows hosts to communicate using strings rather than a byte stream. The transitions of the network consist of transitions of host threads, transitions where a connection on the host sends a message to the network, transitions where a message is received from the network, and transitions representing host and connection failure. An excerpt is given in Fig. 8.

4. Verification overview

In Section 5 an abstract model of a queue is defined. Informally the OCaml implementation is said to be correct if it behaves in the same way as this abstract queue. Formally an abstraction function maps concrete implementation states to abstract queue states. Every transition of the implementation must map, via the abstraction function, to a transition of the abstract queue.

```
trans_mutex (h,tid,call) = case call of
  SC_Mutex_Create → (
      let l = free (FDOM h.m) in
      let m' = FUPDATE h.m (l,NONE) in
      [(h with <| m:=m' |>, SC_Ret(mk_con "Mut" l))])
  || SC_Lock(l) → (
      option_case [] (λ x. case x of
        NONE → (
            (* acquire the mutex *)
            let m' = FUPDATE h.m (l,SOME tid) in
            [(h with <| m:=m' |>, SC_Ret unit)])
        || SOME _ →
            (* mutex owned by another thead *)
            [])
        (FLOOKUP h.m l))
  || SC_Unlock(l) → (
      option_case [] (λ x. case x of
        (* mutex is not held *)
        NONE → [(h,SC_Ret mutex_exception)]
        (* mutex is held *)
        || SOME tid' → (
            if tid = tid' then
              (* we hold the mutex, so unlock it *)
              let m' = FUPDATE h.m (l,NONE) in
              [(h with <| m:=m' |>, SC_Ret unit)]
            else
              (* mutex owned by another thead *)
              [(h,SC_Ret mutex_exception)]))
        (FLOOKUP h.m l))
  || _ → []
```

Figure 7. Host mutex transitions (HOL)

```
msg_trans ((quad:quad),(msg:msg)) (n:net) =
  let (i1,p1,i2,p2) = quad in
  let hid = i2 in
  let h1 = FLOOKUP n.hs hid in
  case h1 of
  NONE → {}
  || SOME(H_Dead _) → {}
  || SOME(H_Alive h) → (
      (* expected connection state, given msg *)
      let st = case msg of
        SYN →        LISTEN
        || SYNACK → SYN_SENT
        || ACK →     SYN_RECV
        || DATA _ → ESTABLISHED
      in
      (* relevant (connectionid,connection) pair *)
      let cidc = get_cidc quad st h.cs in
      (* updated connection and new messages *)
      let g (cid,c) =
        let quad' = rev_quad quad in
        let (c',msgs) = case msg of
          SYN →
              (c with <| st:=SYN_RECV       |>,[(quad',SYNACK)])
          || SYNACK →
              (c with <| st:=ESTABLISHED    |>,[(quad',ACK)])
          || ACK →
              (c with <| st:=ESTABLISHED    |>,[])
          || DATA(ss) →
              (c with <| in_:=(c.in_++[ss]) |>,[])
        in
        ((cid,c'),msgs)
      in
      (* updated network *)
      option_case {}
        (set_eta o cmsgs_in_n n (hid,H_Alive h) o g)
        cidc)
```

Figure 8. Network transitions for messages received (HOL)

The proof is factored into an abstraction function from the concrete OCaml implementation to an intermediate specification, and a further abstraction function from the intermediate specification to the abstract queue. The composition gives the single abstraction function we seek. In Section 6 this intermediate specification is defined. It captures the communication protocol used by the queue, which is a version of the alternating bit protocol (ABP). The function from the ABP to the abstract queue is also defined, but the standard proof that it is an abstraction function is omitted. The interesting part of the verification involves the abstraction function from the implementation to this intermediate specification. Defining this function is straightforward because the states of the implementation and the ABP are closely related: the OCaml code is a direct implementation of the ABP. The main proof obligation is to check that this function is indeed an abstraction function. This is an invariant property, that is, a property of all reachable states.

We examine all reachable states using symbolic evaluation. The reachable states are too complicated to use symbolic evaluation directly, so we use rely/guarantee to rephrase the transition system from the point of view of some arbitrary thread t. Whereas the original system used an interleaving model of concurrency, in this new system, steps of t alternate with steps of interference from other threads, the host, and the network.

We are now in a position to execute through a trace of the system from the point of view of a thread t. There are two possibilities for t. Either it is the active management thread, or it is some other user thread. If it is the active management thread, the thread state is largely known, eg for the sender endpoint it is the function `sender`, with a symbolic value for the `q` parameter. In this case, we can execute the code, checking the reachable states as we go.

The other possibility is that t is a user thread. There are two further possibilities. Either the user thread calls a queue API method, or it executes some arbitrary user code. For each queue API method, the state of the thread t is again largely known: it is the code for the queue method itself. As before, we execute the code, checking each state in turn. Verification ends when the method returns to user code. For a transition involving arbitrary user code, queue resources are private and inaccessible, so the state of the queue is unchanged.

5. Abstract specification

The abstract queue is formed by concatenating the pending messages at the sender endpoint (as recorded on the host filesystem) to those at the receiver endpoint, taking care to avoid duplicate messages. The abstract specification of a queue is straightforward.

```
abstract_queue_trans xs xs' = ∃ msg msgs.
  (* msg appended to end *)
  ( (xs =msgs       ) ∧
    (xs'=msgs++[msg])) ∨
  (* msg removed from front *)
  ( (xs =[msg]++msgs) ∧
    (xs'=       msgs))
```

How is this a useful specification? In the simplest scenario, user code establishes a send queue and `send`s a message `msg` to the receiver endpoint. During the execution of `send`, the `msg` is written to the log file on the sender endpoint. The change to the sender's filesystem translates, via the abstraction function, to a change in the abstract queue, so that the abstract queue now contains `msg`. On the receiver endpoint, user code establishes the corresponding receive queue and then `peek`s at the contents. If `peek` returns, it is because the head of the pending messages on the receiver endpoint was non-empty. The head of the pending messages corresponds, via the abstraction function, to the head of the abstract queue. Since the front of the abstract queue is the message `msg`, the `peek` must return `msg`. This reasoning is completely independent of the internal functioning of the queue, including the details of how the message makes its way from the sender endpoint to the receiver endpoint.

```
abp_host = <| b:bool; ss:string list |>

abp_net = <|
  s    : abp_host;
  msgs : (bool#string) list;
  r    : abp_host;
  acks : bool list
|>

abp_trans_sender n n' = ∃ n0 s0 b s ss acks msgs.
  (* a user thread makes a send call transition *)
  ((n =n0 with <| s:=s0 with <| ss:=ss        |> |>) ∧
   (n'=n0 with <| s:=s0 with <| ss:=(ss++[s]) |> |>)) ∨
  (* a msg moves to the network *)
  ((n =n0 with <| s:=s0 with <| b:=b; ss:=(s::ss) |>;
       msgs:=msgs            |>) ∧
   (n'=n0 with <| s:=s0 with <| b:=b; ss:=(s::ss) |>;
       msgs:=(msgs++[(b,s)]) |>)) ∨
  (* an ack for previous msg is received *)
  ((n =n0 with
     <| s:=s0 with <| b:=b |>; acks:=(~b::acks) |>) ∧
   (n'=n0 with
     <| s:=s0 with <| b:=b |>; acks:=     acks  |>)) ∨
  (* an ack for current msg is received *)
  ((n =n0 with
    <| s:=s0 with <| b:= b; ss:=   ss |>; acks:=(b::acks) |>) ∧
   (n'=n0 with
    <| s:=s0 with <| b:=~b; ss:=TL ss |>; acks:=(   acks) |>))

abp_trans_receiver n n' = ...

abp_trans n n' = ∃ n0 xs ys zs.
  abp_trans_sender n n'
  ∨ abp_trans_receiver n n'
  ∨ (* host or connection failure, transient messages lost *)
     ( (n  = n0 with <| msgs:=(xs++ys++zs) |>) ∧
       (n' = n0 with <| msgs:=(xs++zs)     |>))
```

Figure 9. Alternating bit protocol (HOL)

6. The alternating bit protocol

The queue endpoints use a version of the alternating bit protocol (ABP) to communicate. The ABP is described in [13]. The protocol can be verified independently of the implementation, so we introduce an intermediate system between the abstract specification and the concrete implementation which captures the ABP. This system is defined in Fig. 9. The relationship between the ABP and the abstract queue is expressed as an abstraction function.

```
abp_to_abstract_queue n =
  n.r.ss++(if n.s.b=n.r.b then TL n.s.ss else n.s.ss)
```

The abstraction function takes the pending messages at the receiver, `n.r.ss`, and appends the pending messages at the sender, `n.s.ss` to form the abstract queue. If the `b` values at the endpoints are equal, then the message at the head of the sender's queue has already been accepted onto the receiver's queue, and as a result the message is omitted when forming the abstract queue. The proof that this abstraction function respects transitions uses inductive reasoning to establish protocol correctness, in the style of Paulson [20]. Because it is a well understood technique, the details of the proof are omitted, and inductive reasoning about protocol correctness is not discussed further.

How does the ABP relate to the OCaml implementation? The endpoint state in the ABP, `b` and `ss`, corresponds to the endpoint state in the implementation, `b` and `msgs`, as recorded on the host filesystem. The `msgs` and `acks` at the ABP level correspond to transient messages, on the network, in connection objects on hosts, and even in the active management threads before pending changes to state have been logged to disk. Thus, a connection failure may result in messages being lost from the network, but messages in the connection object on the receiving host, and in the active thread, remain. Similarly, a single endpoint failure still leaves messages on the network and on the other endpoint that can be received by that endpoint's active thread. For space reasons, details of the abstraction function `abstract` from the implementation to the ABP model are omitted.

The main property we want to prove is that transitions of the OCaml implementation, when mapped by the abstraction function, are respected by the ABP. Formally we have the invariant `inv_main` below. This invariant is parameterized by `nps`, which records information such as the quad for the queue that we are interested in. Where this invariant is used, the state `n'` is a successor of `n`.

```
inv_main nps n n' =
  let trns = RC abp_trans in (* reflexive closure *)
  let abstrct = abstract nps in
  trns (abstrct n) (abstrct n')
```

This invariant is further decomposed into several invariants covering common situations. For example, the case where a thread on the sender endpoint takes a step is dealt with by the following invariant. The parameter `ps` records information about the endpoint. For example, `ps.tid` is the thread id of the active management thread.

```
inv_main_sender ps h h' =
  let trns = RC abp_trans_sender in (* reflexive closure *)
  let abstrct = abstract_sender_to_abp_net ps in
  trns (abstrct h) (abstrct h')
```

7. Proof techniques and their integration

This section describes several existing proof techniques, how they were used in the verification, and how they were integrated together on top of the operational foundation.

Basic setup At the heart of the operational approach to verification is symbolic evaluation. Program execution deals with ground terms. Symbolic evaluation deals with parametric terms, where subterms are replaced by variables (logical, not program). While ground evaluation can enumerate the reachable states of a particular instance of a program, such as `fact 5`, symbolic evaluation can deal with the reachable states of all possible instances of a program, such as `fact n`.

Symbolic execution can be automated fairly easily (although the current implementation in HOL4 is rather slow). This affects the structure of proofs: rather than describe the behaviour of a function in a way that echoes the operational semantics, we can simply execute the function. For example, consider an `increment` function that takes a mutable variable and increments its value by one. The behaviour of `increment` as given by the operational semantics cannot be abstracted in any meaningful way. Such functions are handled directly rather than by separating out their properties as a lemma. In the case of the queue, the functions `mk_queue`, `init` and `save` are like this.

The queue code is structured into functions, some of which are part of the queue API, and some of which are internal to the queue itself. It is natural to structure the proof similarly, and so the bulk of the verification consists of separate lemmas, with each lemma corresponding to a particular function in the code. This makes the proof modular, and, since the verification of each function is independent of the others, one can hope that the effort scales with the number of functions.

Each function is verified by symbolically executing it. Global invariants are assumed to hold initially, and verification must establish that they hold at successor states. The operational semantics uses an interleaving model of concurrency, and other threads may interfere with the thread executing the function. Rely/guarantee style reasoning can be used to rephrase the transition systems so that every step of the thread executing the function is followed by a single "rely" step which represents interference from other threads. The cost of this transformation is that the corresponding "guarantee" of the thread in question must be show to hold at each step.

Symbolic execution Symbolic execution is used to explore the set of reachable states. A reachable state lies at the end of a finite sequence, or trace, whose head is a start state, and whose consecutive states are related by the transitions of the system. Symbolic execution works with such traces p, where p_n is a symbolic representation of the system at step n. It is important to note that these symbolic states are characterized by arbitrary HOL formulae. Each of the positions in the trace is dealt with in turn, using information about p_n to derive p_{n+1}.

If the system is non-deterministic, ie there is more that one successor state, then there is a corresponding branch in the proof. In fact, a network involving a queue is highly concurrent and non-deterministic, so rely/guarantee is used to mitigate this non-determinism: as mentioned previously, every step of a thread is followed by a step representing interference from other threads and the environment.

Loops and recursion are handled using induction. Typically, traces are allowed to start in any state that may recur, and before symbolic execution commences, there is an outer induction on the length of the trace p. If a state recurs as the head of some suffix p' of p, the induction hypothesis is invoked to deduce that the invariant holds on the remainder p' of p.

Auxiliary variables and Hoare-style assertions History variables [15] are used to record facts about previous states. For example, a proof might note the value of p_m, which is later used when examining p_{m+n}. At its simplest, p_m is used to determine p_{m+1}. Since the whole trace is directly accessible at any point in the proof, prophecy variables can also be used freely: when examining p_m one is free to case split on the value of p_{m+n}. If, as is often the case, n is not known exactly, one can case split on the first n such that some useful property P holds of p_{m+n}. Prophecy variables are not used in this work, but history variables are used extensively.

Hoare-style assertions [7] are used in a similar way to history variables. Rather than record the exact value p_m of a previous state, the assertion $P(p_m)$ is established, where P is some predicate of interest. This information is used at some later stage, typically to derive some further assertion $P_1(p_{m+1})$, which is itself used to derive $P_2(p_{m+2})$, and so on. In Section 8 there is an example of the use of history variables and Hoare-style assertions.

Invariants Invariants are properties which hold of every reachable state. As an example, the following invariant describes how the state of the queue in memory relates to the state of the queue as recorded on the host filesystem.

```
inv_mem_disk_none ps h =
  let q1 = mem_queue_of_host ps h in
  let q2 = disk_queue_of_host ps h in

  (h.m ' ps.lock = NONE)
   ⟶ ((q1.msgs,q1.b) = (q2.msgs,q2.b))
```

The variable `ps.lock` identifies the queue lock. Given this, the invariant may be paraphrased "if the queue lock is not held, then the in-memory queue[1] and the on-disk queue are the same". Because the queue code contains data races, a further invariant is required to characterize the relationship between the in-memory and on-disk representations when the lock is held by a user thread.

```
inv_mem_disk_some ps h =
```

[1] Rather, the fields `msgs` and `b` of the relevant queues.

```
  let q1 = mem_queue_of_host ps h in
  let q2 = disk_queue_of_host ps h in

  ∀ tid. tid IN FDOM h.ts ∧ ˜ (tid = ps.tid)
  ⟶ (h.m ' ps.lock = SOME tid)
  ⟶ (q1.msgs = q2.msgs) ∨ (∃ msg. q1.msgs = q2.msgs++[msg])
```

The other interesting invariant relates to encapsulation and the contents of the store, and is described in Section 9. Further invariants deal with wellformedness conditions. All host invariants are combined in a single invariant `inv_h`.

Rely/guarantee Rely/guarantee [9] is a core technique for reasoning about concurrent systems. The standard reference is Jones' PhD thesis [9]. Jones helpfully maintains an annotated bibliography on rely/guarantee online[2]. Given a particular thread of interest, the idea is to characterize the interference that may be caused by other threads. As an example, the following rely condition describes how interference affects the value of the in-memory list of pending messages at the sender endpoint.

```
rly_msgs_non_empty ps tid h h' =
  let q = mem_queue_of_host ps h in
  let q' = mem_queue_of_host ps h' in

  (tid = ps.tid)
  ⟶ ˜ (q.msgs = [])
  ⟶ ˜ (q'.msgs = []) ∧ (HD q'.msgs = HD q.msgs)
```

A rely condition characterizes the interference caused by a thread transition from a host `h` to a host `h'`. This rely condition is parameterized by two variables, `ps` and `tid`. The variable `tid` is the thread identifier of the thread that may assume the rely condition, while `ps.tid` identifies the active management thread on the host. The condition `tid = ps.tid` implies that this rely is trivial unless the thread identified by `tid` is the active management thread. The rely condition can therefore be paraphrased: "If you are the active management thread, you may assume that if `msgs` is non-empty and other threads take steps and so interfere with the system state, then `msgs` will still be non-empty, and moreover the head of `msgs` will be preserved". If the queue correctly maintains the privacy of its internal data structures, then this is obvious from the code: user threads on the sender endpoint can call the queue API function `send` to add messages to the end of `msgs`, but the active management thread is the only one that can remove messages from the front of `msgs`.

A rely condition should be reflexive and transitive because it represents zero or more steps of interference.

```
is_rly_gty rg = reflexive rg ∧ transitive rg

is_rly_gty (rly_msgs_non_empty ps tid)
```

There is no need to state the guarantee conditions separately. A rely condition `rly` is parameterized by the thread identifier `tid`. If the host state is `h`, and other threads take steps causing the state to change to `h'`, then thread `tid` can rely on property `rly tid h h'`. Conversely, when thread `tid` itself takes a step from state `h` causing the state to change to `h'`, it must be sure to guarantee `rly tid' h h'` for all other threads `tid'`. There is an example of this in the statement of `send_lemma` below. Thus, if the thread identifier is made explicit, then the rely and guarantee conditions become identical. This clarifies the often observed symmetry between rely/guarantee conditions.

What is the benefit of using rely/guarantee style reasoning? Usually a thread t executes in parallel with other threads. Instead of interleaving steps of other threads with those of t, one can instead interleave steps of interference. If the interference is reflexive and transitive, then one can follow each step of t with a single step of interference. The reachability of this system is at least that of the original. Thus, any invariant of this system is an invariant of the original. Crucially, this approach abstracts from the details of the other threads, making verification significantly more manageable.

[2] homepages.cs.ncl.ac.uk/cliff.jones/ftp-stuff/rg-hist.pdf

Linearizability Linearizability [6] is the requirement that an action composed of multiple atomic actions appear to happen at a single point in time. The use of an abstraction function from a concrete implementation to an abstract specification often requires reasoning about linearizability, since an atomic action of the specification may require several steps in the implementation. In this paper, linearizability is dealt with using invariants, auxiliary variables, Hoare-style assertions, and rely/guarantee. The proof of the correctness of the code with data races in Section 8 illustrates linearizability in detail.

Integration The verification is only made possible by using these techniques in combination. The key to integration is to express all the techniques using the core techniques of symbolic evaluation and invariants.

Symbolic execution is used to explore the state space of the system. During symbolic evaluation, Hoare-style assertions are used to abstract from the details of a particular state, and history variables are used to record details of previously seen states.

The property that the OCaml implementation refines the ABP specification is also an invariant: recall that the implementation refines the ABP specification if, for every reachable implementation state and every transition from that state there is a corresponding transition of the specification.

Rely/guarantee states that for a given thread `tid` and for all reachable implementation host states `h`, any successor state `h'` arising from a transition of a thread `tid'`≠`tid` satisfies `rly tid h h'`. Again, this is an invariant of implementation states `h`. As with refinement, the invariant makes reference to successor states.

There are therefore three different kinds of invariant. For each transition of a thread `tid`, we must prove the guarantee corresponding to the rely, the correctness of the abstraction function, and the other basic invariants. For example, the part of the verification dealing with the `send` queue API method involves the following typical goal, parts of which have been omitted for clarity.

```
send_lemma p = ∀ ps tid tid' h h'.
  ...      (* variables are set appropriately *)
  ⟶ inv_h ps h
  ⟶ inv_h ps h' ∧ rly_h ps tid' h h' ∧ inv_main_sender ps h h'
```

The aim is to prove that for any path `p`, if the last two states `h` and `h'` in the path arise as a transition of thread `tid`, then assuming the basic invariants `inv_h` hold of state `h`, we must show that they hold of state `h'` (`inv_h ps h'`), that the transition of thread `tid` guarantees the relies of other threads `tid'` (`rly_h ps tid' h h'`), and that there is a corresponding transition of the ABP model (`inv_main_sender ps h h'`). In this way, all techniques used in the proof, including privacy (part of `inv_h`), rely/guarantee, and refinement, are cleanly combined.

8. Data races

Races on shared resources are dangerous, but they can be prevented using locks. For performance reasons, it is important to hold locks for the shortest time possible. The queue uses locks wherever shared state may be accessed concurrently, with one interesting exception. In order to improve performance, the `private_send` function permits `q.msgs` to be dereferenced outside the locked region (`private_send` is reproduced below). In the rest of this section, I discuss why this might cause problems, why it does not cause problems in this case, and how this is handled in the proof.

Recall that the main property we are proving is that, from a reachable state, every step of the implementation corresponds to a step of the alternating bit protocol. The possible problem arises in the last line of `private_send`, when a message is sent to the network (in fact, its immediate destination is the relevant connection object on the host). The alternating bit protocol requires that the message that is sent at this point is the first message pending *at this point*. However, the dereference of the head of the pending messages, `List.hd ( ! ( q.msgs ) )`, occurs some steps before the message is sent. Potentially the head of `q.msgs` could have changed in the intervening time, in which case the dereferenced value would not be current, and there would be no corresponding step of the alternating bit protocol. The problem is one of linearizability.

It is important to realise that even if `q.msgs` were dereferenced inside the locked region, the fact that the send occurs outside the locked region still causes problems. If both the dereference and the send occur inside the locked region, then there is no problem. However, depending on the size of the message, the send may take a long time, and thus impact performance considerably.

Fortunately the code is correct as it stands. The function `private_send` is executed only by the active management thread. Although other threads may read and update `q.msgs`, they can only add messages to the end. Once `q.msgs` is non-empty, the head remains constant until the active management thread itself removes it in response to a new acknowledgement received from the other endpoint. Thus, it does not matter that `q.msgs` is dereferenced some time before the send occurs since the value will not change in the intervening time. The situation is only slightly more complicated by the fact that the abstraction to the alternating bit protocol uses the on-disk rather than in-memory queue. Fortunately the `inv_mem_disk_none`, `inv_mem_disk_some` invariants guarantee that the two representations are the same (or, at least, that the head of the in-memory queue is the same as the head of the on-disk queue, since a user thread may be calling `send`, which has appended a message to the in-memory queue, but not yet logged the results to disk).

The reasoning may be presented slightly more formally using Hoare-style assertions. Because the invariants `inv_mem_disk_none`, `inv_mem_disk_some` are involved in the reasoning, I also show how they are preserved.

```
let private_send q =
    (* 1 *)
  let _ = Mutex.lock q.lock in
    (* 2 *)
  let _ = while ! ( q.msgs ) = [] do
    Condition.wait q.cond q.lock done in
    (* 3 *)
  let _ = Mutex.unlock q.lock in
    (* 4 *)
  let msgs = [ string_of_bool ( ! ( q.b ) ) ;
               List.hd ( ! ( q.msgs ) ) ] in
    (* 5 *)
  let _ = Message.send ( dest_Some ( ! ( q.fd ) ) ) msgs in
    () in
```

At 1, the global invariant holds by assumption. In particular, we restrict attention to the relevant part of the global invariant, `inv_mem_disk_none` and `inv_mem_disk_some`. At 2, the thread has successfully taken the lock. Immediately prior to this step, the lock is not held, so the condition of the invariant `inv_mem_disk_none` is satisfied, and the in-memory and on-disk queues are the same. This is not altered by the lock being taken, so immediately after the lock is taken, the in-memory and on-disk queues are still the same. Other threads may interfere at this point, but they guarantee not to alter (the shared parts of) the queue state while the lock is taken. Thus, while the lock is held, the conclusion of `inv_mem_disk_none` remains true. `inv_mem_disk_some` is trivially true since the condition is false, and this remains the case while the lock is held. At 3, the lock is still taken, and `q.msgs` is non-empty. Again, other threads may interfere, but the active thread can rely on `rly_msgs_non_empty` to ensure that the head of `q.msgs` is constant. In fact, the head of the queue remains constant also because the lock is held, indicating that `rly_msgs_non_empty` is really only necessary outside the locked region. This is related to the fact that the mutex and condition variable are used here primarily for inter-thread communication, not to protect shared access to resources. At 4 the lock is released. The conclusion of `inv_mem_disk_none` holds immediately before the lock is released, so `inv_mem_disk_none` holds immediately after the lock is released. `inv_mem_disk_some` is trivially true because the condition is false. From this point on, user threads may interfere, and the lock may be free, or taken by a user thread. However, `rly_msgs_non_empty` guarantees that the head of the list remains constant. At 5, `q.b` and `q.msgs` have been dereferenced. The only thread that can alter the value of `q.b` is the sender thread, which we are currently executing, and `rly_msgs_non_empty` ensures `q.msgs` remains current. When the send finally occurs, the lock can either be free, or held by one of the user threads (nothing can make the lock be held by the active sender thread). The invariants `inv_mem_disk_none`, `inv_mem_disk_some` ensure that, regardless of whether the lock is held or not, the previously read values `q.b` and `List.hd ( ! ( q.msgs ) )` correspond to the values currently on disk. There is thus a corresponding transition of the alternating bit protocol respecting the abstraction function.

9. Context, privacy and encapsulation

The queue is intended to be used as a library by other applications. It should behave correctly regardless of the context in which it is called. This is achieved by keeping queue resources private. If resources are private, then invariants on the resources can be enforced. In this work, resources are unforgeable references to host state, typically store locations, mutex identifiers, condition variable identifiers and network connection identifiers. Even a thread identifier might be considered a resource, although in the current model of OCaml there is no way to manipulate a thread via its identifier, so whether thread identifiers are private or not is immaterial. The exact nature of a resource is orthogonal to privacy concerns— nothing is lost by considering all resources to be store locations.

The idea of privacy is very simple. A single resource r is private to a function f if, wherever it occurs, it is syntactically within the body of f. For example, location `Loc i` is private to function λ `x. Loc i` in `App(`λ `x. Loc i, unit)`. In general there may be more than one resource and more than one function. For example, the queue API methods for the receiver all share the same resources.

A resource r may be private to f initially, but subsequently f may leak r to user code. Possible ways f can leak r are by making r accessible via shared store, or by returning r to user code (either normally, or during exceptional return). To ensure that r remains private throughout an execution, f must be *privacy-preserving*. In particular, f must not return r to user code (directly, or during an exceptional return). In a single-threaded setting, f must ensure that r is not accessible via other locations in the store when f finishes executing. In a concurrent setting f must ensure that r is not accessible to other threads via other locations in the store at any point during execution. For the queue, resources are never accessible via other locations in the store. Thus, as far as privacy is concerned, verification must establish that the queue API functions do not return private resources to user code.

OCaml can support notions of privacy for in-memory data structures. However, the queue also uses the network and the filesystem. External restrictions must be placed on how these are used by the context. For example, no user thread should write directly to the log files otherwise chaos might ensue.

The definition of privacy The first definition gives all subclosures of a closure `cl`, omitting those that match one of the functions `fns`. This is then lifted to contexts and framestacks in the obvious way.

```
subcls fns cl =
  if cl IN fns then [] else cl::(case cl of Cl(e,env) →
    FLAT (MAP (λ (v,cl). subcls fns cl) env))
```

A set of resources `rs` are private to a set of functions `fns` if, when occurrences of `fns` are removed, there are no occurrences of `rs`.

```
private_cl fns rs cl = ∀ r. r IN rs ⟶ ~ (MEM r (subcls fns cl))

private_fs fns rs fs = ∀ r. r IN rs ⟶ ~ (MEM r (subcls_fs fns fs))

private_clfs fns rs (cl,fs) =
  private_cl fns rs cl ∧ private_fs fns rs fs
```

Recall that thread state consists of a pair, where the second component is a framestack, and the first component represents a running thread evaluating a closure, or a thread blocking on a system call.

```
private_thread fns res t =
  let rs = res_to_cls res in
  case t of
    (T_Run cl,fs) → private_clfs fns rs (cl,fs)
    || (T_Block call,fs) →
        private_fs fns rs fs ∧ case call of
          SC_Ref(cl) → private_cl fns rs cl
          || SC_Deref(l) → ~ (l IN res.locs)
          || SC_Assign(l,cl) →
              ~ (l IN res.locs) ∧ private_cl fns rs cl
          || ...
```

Store invariant Queue resources are never accessible from other store locations, either while user code executes or while queue API functions execute. This is captured by the following invariant. The definition `private_cl` is first lifted to the host store.

```
private_store fns res s =
  let rs = res_to_cls res in
  ∀ loc:loc. loc IN FDOM s ⟶ private_cl fns rs (s ' loc)

inv_private_store ps h =
  let res = ps_to_res ps in
  let fns = qfns ps in
  private_store fns res h.s
```

The invariant is checked while user code executes and while queue API functions execute. Since the resources are private to user code, there is no way user code can write them into the store (except by writing a queue API function itself into the store, but this preserves privacy). For the queue API functions it is clear from the code that this invariant is satisfied.

Privacy verification Verification starts by considering an arbitrary thread, with resources `res` private to functions `fns`. In this work, `fns` are the queue API functions. There are two cases. Either a function executes, or arbitrary user code executes. For arbitrary user code, by considering all possible cases, one can show that after a step of execution the resources remain private, and the state of the resources is unchanged. This is described in more detail below. Since the proof is independent of the resources `res` and the set of functions `fns`, the result is OCaml metatheory and may be reused in other verifications.

The second case arises when one of the queue API functions `fns` executes. For each function, verification must establish that resources are not written to the store while the function executes, nor returned to user code when the function finishes executing. The functions are specific to the code being verified, in this case the queue, and therefore this part of the verification cannot be reused.

Resources remain private while executing user code The following lemma describes the case that arbitrary user code executes with resources `res` private to functions `fns`.

```
user_lemma = ∀ tid h h' t t' fns res.
  ... (* variables are set appropriately *)
  ⟶ ~ (∃ fs f arg.
    f IN fns ∧ (t = (T_Run f,App(Wrap Hole,Wrap(Clo arg))::fs)))
  ⟶ private_thread fns res t  ∧ private_store fns res h.s
  ⟶ h' IN set (trans_h_tidt h (tid,t))
  ⟶ private_thread fns res t' ∧ private_store fns res h'.s
    ∧ (eval_res res h' = eval_res res h)
```

Thread `t` is identified by thread identifier `tid`. The first condition restricts thread `t` to user code rather than (an application involving) one of the `fns`. The inductive assumption is that the resources are private in both the thread state `t` and in the host store `h.s`. The next condition restricts `h'` to a successor of `h` (within `h`, the thread making the transition is `t`). The conclusion is that resources remain private in both the thread state `t'` and in the host store `h'.s`, and moreover the value of any resource is unchanged. This lemma is proved by analysing all the possible cases for the user code, and all the possible ways each case might evaluate.

Resources remain private after executing queue API functions Recall that the proof consists of two cases: either arbitrary user code executes, or one of the queue API methods executes. Prior to this case split, there is an outer induction on the length of the trace p, as described in Section 7. In the case that a queue API method executes, resources are private to functions in the framestack context *fs*. The function executes, private queue resources may be manipulated, and eventually the method returns a value to the user code context *fs*. Providing the method has not returned private resources to user code, the remaining suffix p' of p satisfies the inductive assumption that at p'_0 resources are private to the queue API functions. The inductive assumption is invoked to conclude that on p', the remainder of p, the resources remain private.

10. Proof skeleton, composing the fragments

The previous sections detail the verification of the internal queue function `private_send`, the queue API method `send`, and arbitrary user code. How are these separate verifications combined?

When we talk about verifying a function, we really mean verifying a function executed by a thread. Similarly, verifying arbitrary user code means verifying a thread executing arbitrary user code. The task of composing the verification fragments involves assembling the individual thread verifications into a verification of the host.

The host is more than a set of threads. Most of the components of the host are passive, that is, they do not of themselves cause transitions to occur. The store is an example. Some parts of the host are active, such as the network connections. Network connections cause transitions, but they do not directly affect threads— a thread has to make an explicit system call to interact with a connection object. Some parts of the host are active and directly affect the threads. For example, a thread might be sleeping, waiting on a condition variable, and the system may decide to wake the thread up, even if the condition has not been signalled[3]. Moreover, the system behaviour is not just the behaviour of the hosts. Clearly the additional behaviours of the host and the behaviour of the network are important; however, in this section we limit the discussion to host thread transitions only. Further composition steps treat the additional host behaviours and the network behaviour.

[3] This is the reason `waits` are wrapped in `while` loops. Modern implementations *may* not exhibit this traditional behaviour, but of course, it is safer to assume they do.

We first define a state transition system whose reachability is at least that of the system we are interested in.

```
sender_endpoint_thread_starts ps tid (int,h) =
  let t = h.ts ' tid in
  let fns = qfns ps in
  let res = ps_to_res ps in
  (int = T)
  ∧ case tid = ps.tid of
    T → (t = sender_active_thread ps)
    || F → (private_thread fns res t)

sender_endpoint_thread_trans ps tid (int,h) (int',h') =
  let t = h.ts ' tid in
  (int' = ~ int)
  ∧ case int of
    T → ((rly_h ps tid) h h' ∧ inv_h ps h')
    || F → (h' IN set (trans_h_tidt h (tid,t)))

sender_endpoint_thread_sts ps tid =
  let s = sender_endpoint_thread_starts ps tid in
  let t = sender_endpoint_thread_trans ps tid in
  (s,t)
```

The `sender_endpoint_thread_sts` describes transitions of the network from the point of view of thread `tid`. As usual, we use symbolic execution to examine traces p of this system. The start states `sender_endpoint_thread_starts` constrain p_0. Recall that `ps.tid` is the active management thread. The first case split on `tid = ps.tid` (from `sender_endpoint_thread_starts`) determines whether we are executing the active thread, or a user thread. If we are executing the active thread we use the verification of the `sender` function. Otherwise we are executing a user thread. The start states are further constrained by `private_thread fns res t`. Recall that this predicate, defined in Section 9, says that resources `res` are private to queue API functions `fns` in thread `t`. Then either the currently evaluating closure is one of the functions `fns` or it is not. If it is not, then arbitrary user code executes, and we invoke the metatheory from Section 9. Otherwise we case split on which of the queue API functions from `fns` the closure is and invoke the appropriate queue API method verification.

11. Related work

Operational semantics is a standard technique for defining programming languages and proving metatheory, but is less often used directly as a basis for program verification. An example of a large operational semantics is the formal description of TCP/IP [27, 2]. Symbolic evaluation is a natural counterpart to operational semantics. For example, the work on TCP/IP involved significant testing using symbolic evaluation inside a theorem prover [3].

One researcher who has advocated reasoning directly about the operational semantics is Moore, although he explicitly recognizes that this approach has only recently become feasible: “had there been decent theorem provers in the 1960s, Floyd and Hoare would never have had to invent Floyd-Hoare semantics!”[4] His work [12] focuses on Java programs, which have first been compiled to bytecode. Correctness properties are phrased as properties of the bytecode, and reasoning occurs above the bytecode, not above the original Java program. The examples treated, such as an “add one” program and a Java function that implements factorial, are significantly simpler than the work presented here.

A rare example of operational reasoning applied to a high-level language is the work of Compton [4], who verifies a version of Stenning’s protocol for a restricted model of Caml and UDP. This work is similar, but again much simpler, than that presented here.

The most directly related piece of work is the author’s verification of Peterson’s algorithm for mutual exclusion [26]. This work was simpler than that presented here, but several of the core techniques, including symbolic evaluation and rely/guarantee, were used in the same way that they are here.

Hoare popularized the use of assertions for reasoning about programming languages [7]. Owicki and Gries extended Hoare’s work to treat concurrent systems [19]. Since then, many variations on the original Hoare logic have been proposed. For example, a recent mechanization of a novel Hoare logic for recursive procedures and unbounded nondeterminism is [16]. Hoare logic has been used to reason about real languages such as Java [17].

Completeness of the refinement approach is considered by Lamport and Abadi [15]. The authors note that the technique of refinement is not new and point to the slightly earlier application of refinement by Lynch and Tuttle [14] and even earlier work of Lamport [11] and Lam and Shankar [10].

In this work, linearizability arises because of the need to match many implementation transitions to a single specification transition. However, linearizability has been proposed as a form of specification independent of refinement [6]. This avoids the overhead of defining an abstract model; however, in this paper the abstract model (the alternating bit protocol) is also used to reason about protocol correctness.

Local reasoning, separation, privacy and encapsulation are currently areas of rapid growth in theoretical computer science. The most popular approaches derive from Reynolds’ separation logic [25]. Like its ancestor, Hoare logic, separation logic has been adapted in various directions, for example, to include rely/guarantee style reasoning [28]. A more operational approach to local state and privacy has been pursued by Pitts and others [22, 21].

12. Conclusion and future work

This work presented the operational approach to verification, including details of how it was applied to verify a persistent queue. The mechanization involves around 3000 lines of definitions, and 3000 of proof, representing approximately 6 months of effort. The proof scripts take about an hour to process, with most of that time spent evaluating symbolic expressions.

The proof was constructed to suit mechanization. Essentially all the proof obligations were reduced to checking a single invariant of the reachable states of a transition system. Symbolic evaluation was used to generate the reachable states, and invariant checking was based on HOL4’s rewriting and simplification.

The heavy reliance on symbolic evaluation had advantages and disadvantages. The main advantage was that mechanization was fairly straightforward. In the common case where a single step of evaluation does not affect the rest of the host state, all invariants are proved automatically. This makes the scripts robust against trivial changes to the queue code and the OCaml semantics. The main disadvantage of this approach is the slow speed of symbolic evaluation in HOL4. Much effort, was spent trying to address this problem, both in terms of writing specialized tactics and in reshaping the proof. Even so, it can take 10 seconds or more to execute a single step of the system (including automatically discharging invariants), and a single queue API function may require hundreds of steps to execute. Thus, it takes a long time to construct an initial proof, and to rework existing proofs. Moreover, waiting for symbolic evaluation to complete results in low productivity for the human being driving the proof process. An obvious conclusion is that to make the operational approach more feasible would require investment in theorem prover infrastructure, particularly in the areas mentioned.

Several aspects of the case study make it particularly suited to the operational approach. The abstract queue in Section 5 has a natural operational specification, as does the intermediate model of the alternating bit protocol. At the implementation level, operational semantics is the standard for defining realistic programming

[4] `http://www.cs.utexas.edu/users/moore/best-ideas/vcg/`.

languages such as OCaml, and can also be used to describe the rest of the system, including the hosts and the network. The OCaml implementation is a direct refinement of the alternating bit protocol, and checking the existence of such a refinement is well suited to mechanization. The techniques that were needed in the verification were all easily expressed and integrated above the operational semantics. Furthermore, new techniques, such as the simple approach to privacy and encapsulation, could also be developed above the operational foundation. For this case study at least, operational reasoning was a natural approach, which proved flexible, and imposed little overhead on the proof process. Presumably similar systems could be handled in a similar manner.

This work addresses the question of what is needed to reason about systems that have been defined operationally. Since operational semantics is the standard for formally defining complex systems, this is a natural and important question. Since most verification is not based directly on operational semantics, it is natural to wonder why. Hopefully this work demonstrates some of the potential of operational reasoning, as well as indicating to some extent where problems lie. There is much scope for future work.

More complex examples There are many exciting opportunities for verifying implementations of interesting algorithms. For example, Amazon currently use an implementation of the Paxos algorithm as a core part of their network infrastructure. Reports suggest that most errors are errors introduced while refining the Paxos specification to production code. Using the techniques presented here, I believe that implementations of such complex algorithms are now well within reach of mechanized verification. Other interesting targets are concurrency libraries, such as Doug Lea's `java.util.concurrent`.

Other languages Although this work treats the case of two OCaml endpoints, the queue could just as easily have been written in, say, Java. The model of OCaml would be replaced by a model of Java, but the rest of the model would remain unchanged. Moreover, the ABP model and abstract specification are language neutral, and the notion of privacy would be the same, so the general structure of the proof should be preserved. Given a reasonable definition of the Java operational semantics, it should even be straightforward to treat a queue where one endpoint is implemented in OCaml and the other in Java, giving a verified proof of interoperability.

Other approaches to privacy The approach to privacy and encapsulation presented here is based on restricting access to (mutable) resources. In a typed setting it is more natural perhaps to restrict access to values through types ie by using abstract types or signatures to hide type information. There are obvious similarities between these approaches. The approach of this paper restricts resources to appear only within a known set of functions. The type-based approach allows resources to appear anywhere, but only within a known set of functions can they be accessed and manipulated. The type-based approach is supported by the language itself, which is one reason it is more natural. To support the type-based approach would require a model of OCaml modules, together with details of the module type system, so the initial overhead is higher. The higher initial overhead was the reason that the type-based approach was not taken here. However, modelling these parts of OCaml is certainly a long term aim of this work.

Models The models of OCaml, hosts and the network are abstract, but reasonably realistic. For example, the control messages on the network are modelled directly on those used by TCP/IP. However, the models could be improved further.

The model of core OCaml should be linked to that of Owens [18]. That model was not used directly because it is based on flat expressions, whereas this work required a more structured representation based on closures and framestacks. Owens' work includes a definition of the OCaml type system and a proof of type soundness. The verification described above makes no use of types, although type information can make verification easier. It would also be good to incorporate other static analyses, not only those based on types, into this framework.

The current model of the host includes several unrealistic assumptions. For example, the model assumes an infinite number of file descriptors, none of which are ever re-allocated. Consequently, errors involving file descriptor exhaustion, wrap-around or re-allocation are not addressed in this work. A long term aim is to make the model of the host more detailed and realistic. This should be possible without modelling the operating system or the network stack in detail (although these make interesting complementary projects).

The model of networking is based on the author's previous work [26], although there is no formal connection. The current model does not fully reflect the behaviour of TCP/IP; for example, it omits certain rare behaviours such as simultaneous connection. The previous work includes these behaviours, but is not abstract enough: the size of the specification alone makes it difficult to use in verification. The previous work needs to be revisited and abstracted even further with an eye to replacing the current network model.

Alternative queue implementations The current OCaml implementation of the queue could be improved in several ways. Rewriting the whole log file every time state changes is clearly unnecessary. A better approach would be to store individual updates in separate files. Lock contention could be reduced by splitting the endpoint queue in half, with API functions accessing one half, and the active management thread accessing the other. Only when the half used by the active thread becomes empty would a lock need to be taken, and the contents of the other half copied over. A final improvement would be to send more than one message at a time. This would involve changing from an implementation based on the alternating bit protocol to one based on the sliding window protocol. Unfortunately the sliding window protocol is significantly more complicated to work with because it requires restrictions on the rate at which messages are sent, in order to avoid wrapping the message identifier too quickly. Indeed, despite lots of attention, the sliding window protocol has yet to be verified satisfactorily, ie including precise conditions on the rate that messages are sent. On the other hand, the standard sliding window protocol assumes messages can be reordered, whereas the underlying TCP used here guarantees that no messages are reordered, so that it should be possible to avoid issues of identifier wrap-around altogether.

Liveness This work treats safety properties of the queue, but it would also be good to tackle liveness. Although the use of locks is fairly elementary (there is only one lock per queue) liveness is still non-obvious. Even during normal operation, liveness depends on invariants about the way the network is used. In the presence of host and network failure, liveness is not obvious. Liveness should be verified.

Denotational semantics In the functional programming community, there is a tradition of algebraic reasoning, using equalities between (purely functional) program fragments. The absence of side-effects and the restriction to terminating functions justifies this form of reasoning. Higher-order theorem provers, such as HOL4 and Coq, directly support such equational reasoning for their own internal (pure) languages. Operational reasoning stresses the step-by-step nature of computation, which handles side-effects well, but is ill-suited to this form of reasoning. For the pure fragment of OCaml it is important to support such reasoning. Therefore, a long term goal is to reason about the theorem prover equivalents of structures such as lists, and then to transfer the results directly to the OCaml code. For example, in the pure fragment of OCaml we can already prove that list append is associative. The next step is to show that any equality concerning lists, that is established in the theorem prover, is valid for the purely functional fragment of OCaml. This avoids the need to transfer results individually, instead making the full range of HOL equalities available to reason about OCaml code, thereby providing strong support for algebraic reasoning.

Acknowledgements I thank the anonymous referees for their comments, which improved the paper significantly. I also wish to thank Peter Sewell, who commented extensively on this and previous versions of the paper. I gratefully acknowledge the support of EPSRC grants GR/T11715 and EP/C510712.

References

[1] IEEE standards association, POSIX. Available online at http://standards.ieee.org/regauth/posix/.

[2] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proc. SIGCOMM 2005 (Philadelphia)*, Aug. 2005.

[3] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL'06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 2006. ACM Press.

[4] M. Compton. Stenning's protocol implemented in UDP and verified in Isabelle. In M. D. Atkinson and F. K. H. A. Dehne, editors, *CATS*, volume 41 of *CRPIT*, pages 21–30. Australian Computer Society, 2005.

[5] M. Felleisen and M. Flatt. Programming languages and lambda calculi. Available online at http://www.cs.utah.edu/plt/publications/pllc.pdf.

[6] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[7] C. Hoare. An axiomatic basis for computer programming. *CACM: Communications of the ACM*, 12, 1969.

[8] R. Johnson. J2EE development frameworks. *IEEE Computer*, 38(1):107–110, 2005.

[9] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Prgr.Res.Grp. 25, Oxford Univ., Comp. Lab., UK, June l981.

[10] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[11] L. Lamport. Specifying concurrent program modules. *ACM TOPLAS*, 5(2):190–222, Apr. 1983.

[12] H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. *Lecture Notes in Computer Science*, 3223:184–200, 2004.

[13] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, MIT, 1987.

[15] M. Abadi and L. Lamport. The Existence of Refinement Mappings. In *Proc. of the 3rd Symposium on Logic in Computer Science*, pages 165–175, Edinburgh, July 1988. IEEE.

[16] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471, pages 103–119, 2002.

[17] D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods Europe (FME 2002)*, volume 2391, pages 89–105, 2002.

[18] S. Owens. A sound semantics for OCaml light. In S. Drossopoulou, editor, *ESOP*, volume 4960 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[19] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[20] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[21] A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002.

[22] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[23] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming (JLAP)*, 60:3–15, 2004.

[24] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Progamming (JLAP)*, 60:17–139, 2004.

[25] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

[26] T. Ridge. Operational reasoning for concurrent Caml programs and weak memory models. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2007.

[27] T. Ridge, M. Norrish, and P. Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[28] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR'07: Conference on Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[29] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.