Verifying concurrent software using movers in CSPEC

Tej Chajed, M. Frans Kaashoek, Butler Lampson[†], and Nickolai Zeldovich
MIT CSAIL and [†]Microsoft Research

Abstract

Writing concurrent systems software is error-prone, because multiple processes or threads can interleave in many ways, and it is easy to forget about a subtle corner case. This paper introduces CSPEC, a framework for formal verification of concurrent software, which ensures that no corner cases are missed. The key challenge is to reduce the number of interleavings that developers must consider. CSPEC uses mover types to re-order commutative operations so that usually it's enough to reason about only sequential executions rather than all possible interleavings. CSPEC also makes proofs easier by making them modular using layers, and by providing a library of reusable proof patterns. To evaluate CSPEC, we implemented and proved the correctness of CMAIL, a simple concurrent Maildir-like mail server that speaks SMTP and POP3. The results demonstrate that CSPEC's movers and patterns allow reasoning about sophisticated concurrency styles in CMAIL.

1 Introduction

Achieving high performance on a single computer requires concurrency, such as running on multiple cores or interleaving disk and network I/O with computation. Concurrent software, however, is difficult to get right because threads can interleave in many ways, and reasoning about all possible interleavings is hard. Furthermore, testing is insufficient, because there are usually too many interleavings to consider, and because it is difficult to reproduce a bug unless the developer knows the precise interleaving that caused it. By contrast, formal verification can prove that a system behaves correctly (i.e., satisfies its specification) in every possible interleaving, including all corner cases.

There has been some prior work on machine-checked verification of concurrent systems software on a single computer. For example, CertiKOS has verified spinlocks for protecting scheduling queues [13, 21]. As we discuss in detail in §2, that work focuses on lock-based concurrency. Systems in which concurrency takes the form of multiple processes sharing a file system tend to avoid the use of locks because they interact badly with crashes. This requires reasoning about many possible interleavings, since there is no lock enforcing sequential execution during critical sections. Work on a concurrent garbage collector [17, 18] supports reasoning about lock-free shared-memory concurrency, but relies on pen-and-paper proofs for key theorems, and does not support important proof patterns needed for CMAIL.

This paper presents CSPEC, a framework for specifying, implementing, and proving the correctness of concurrent systems. CSPEC supports reasoning about concurrent processes that share a file system, as well as about concurrent threads that share data structures in memory. All of CSPEC is implemented and proven in the Coq proof assistant [36].

To show that CSPEC makes it fairly easy to prove the correctness of concurrent software, we used it to develop a simple concurrent mail server, CMAIL. Typical mail servers such as Maildir do not use file locks for mail delivery, since locks are fragile if a process is killed or suspended while holding the lock [3]. Instead, Maildir relies on careful reasoning about atomicity and ordering of file system operations (e.g., writing data to a temporary file before renaming it into the user's mailbox directory). Mail delivery must interact safely with mail pickup (e.g., retrieving mail via POP3)—for instance, retrieving mail from a mailbox in the presence of concurrent deliveries to the same mailbox. Finally, other parts of the mail server do use POSIX file locking—for example, to ensure that a message cannot be retrieved and deleted at the same time.

The key challenge in CSPEC is to reduce the number of interleavings that the developer must consider in code like CMAIL's lock-free delivery. To achieve this, CSPEC uses the notion of *mover types* [27], which exploits the fact that certain operations are left- or right-commutative with respect to concurrent operations by other processes. CSPEC uses mover types to re-order operations so that processes appear to execute longer blocks of sequential code. This reduces the problem of reasoning about all interleavings to reasoning about just the atomic execution of these longer sequential blocks. CSPEC builds on prior work that used mover types to reason about concurrency [11, 16, 18], and provides the first general mover framework with a machine-checked proof of its implementation (see §2).

CSPEC allows the developer to separately tackle different aspects of the design by structuring the overall system as a stack of layers. Each layer has a formal specification and its own implementation and proof. CSPEC provides a library of patterns for different kinds of proofs that a layer might need, such as mover types, retry loops, abstracting state, partitioning state, and proving that the code follows

a protocol (i.e., a set of rules), such as accessing memory only while holding a lock.

We evaluate two key aspects of CSPEC: whether CSPEC makes it possible to do correctness proofs for sophisticated concurrent software, and whether the resulting concurrency translates into actual speedup. CMAIL is our primary case study of verifying concurrent software. CSPEC allowed us to handle its challenging concurrency patterns, such as a delivery process that modifies a mailbox directory at the same time the user is picking up mail, multiple delivery processes that write messages into the same mailbox, and concurrent processes sharing the same temporary directory to store partially received messages.

CSPEC's layering allowed us to decompose the overall correctness argument for CMAIL into smaller steps, each layer addressing a specific aspect of CMAIL's concurrency and using a CSPEC proof pattern to formally verify it. All of CSPEC's proof patterns were important in CMAIL, and most patterns were used multiple times. Designing and building CSPEC and CMAIL took two people approximately 6 months, on top of another 12 months of experimenting with several failed alternative designs. Experiments show that CMAIL's concurrency makes it run faster on a multi-core machine.

CMAIL's concurrency model is based on processes sharing a file system. CSPEC also allows developers to reason about other concurrency models. To demonstrate this, we specified a model of x86-TSO [34], consisting of a shared memory with per-core write buffers. On top of this model, we implemented and proved the correctness of an atomic counter. We used 10 layers to verify this counter, re-using proof patterns that we developed for CMAIL.

To summarize, the contributions of this paper are:

- CSPEC, a framework for verifying concurrent systems using mover types, which is fully machine-checked in Coq.
- A modular approach that simplifies proofs using layers and a library of proof patterns.
- An evaluation that uses CSPEC to formally prove the correctness of a concurrent mail server on top of a POSIX file system, and an atomic counter on top of a weak shared-memory model. The results demonstrate that CSPEC allows reasoning about a wide range of concurrency styles.

The source code of CSPEC and the example applications are publicly available at https://github.com/mit-pdos/cspec. Our prototype has several limitations. CMAIL does not include verified parsing or protocol implementations of SMTP or POP3. CSPEC uses Coq's extraction to generate executable code, which means the executable programs rely on either Haskell or OCaml at runtime; hence, one of these is part of the trusted computing base. Finally, CSPEC cannot be applied to existing software, since it requires the program to be written in CSPEC's framework.

2 Related work

CSPEC adopts many ideas from previous research in specification and verification of concurrent shared-memory and distributed-systems software.

Verification approaches. There are many ways to verify concurrent software. After experimenting with several different approaches (including several versions of concurrent separation logic [5] and rely-guarantee [12, 20]), we settled on using the state machines and refinement that underlie TLA and I/O automata [23, 24, 30, 31], combined with the proof pattern of movers [27].

CIVL [17, 18] (and its predecessor QED [11]) is the work most closely related to CSPEC, and CSPEC borrows many ideas from it. CIVL uses the state-machine approach with support for atomic actions, movers, a mover pattern inspired by CIVL's yield sufficiency automaton, and location invariants to reduce the proof burden. It is implemented as an extension to Boogie, and the authors used it to specify and verify a concurrent garbage collector that uses an algorithm by Dijkstra et al. [10] that has tricky concurrency reasoning. Subsequent work used CIVL to reason about concurrent programs on x86-TSO [4].

CSPEC borrows atomic actions and movers from CIVL, but differs in two ways. First, many of CIVL's proofs (e.g., all the proofs in §4 of [18]) are done with pen and paper [15], whereas all parts of CSPEC are machine-checked in Coq. Second, CSPEC supports some patterns not found in CIVL, such as retry loops, which were important for reasoning about concurrency in CMAIL. Furthermore, this paper reports on our experience in using CSPEC for a different application (namely, a file-system-based mail server rather than a concurrent garbage collector), which exhibits different styles of concurrency.

The advantage of the fact that CSPEC has machine-checked proofs, compared to CIVL's pen-and-paper proofs, is that it gives us confidence that all of the proof patterns are correct (once we prove them). This, in turn, makes it easier to experiment with proof patterns. For example, during the development of CSPEC, we added (and modified) a number of proof patterns (see §6). Having machine-checked proofs gave us confidence that we did not introduce any bugs.

Like CSPEC, CertiKOS's CCAL [14] organizes reasoning about concurrent execution into layers and has a linking theorem to "compile" top-level operations into bottom-layer operations. CCAL has been used for fully machine-checked proofs of several lock implementations and of CertiKOS's concurrent scheduling queue [13, 21].

CCAL has no notion of movers; it uses rely-guarantee-style reasoning to prove atomicity for operations in a shared log. The only case in which CCAL can avoid reasoning about interleaving is when a thread accesses only thread-private memory. This is insufficient for CMAIL, which accesses shared files and directories all the time: for instance, mail pickup can read a message that was just written by a concurrent delivery process.

Another notable example of verifying concurrent systems software is Microsoft's HyperV verification, which used VCC [7–9], but the work on VCC and HyperV appears to have stopped after verifying about 20% of HyperV [7]. In contrast to CSPEC, the VCC approach did not use mover types for reasoning about concurrency.

Distributed systems. Related work in verifying distributed systems focuses on network protocols (message loss and re-ordering) as well as node failures and network partitions, while assuming a static partitioning of state across nodes [16, 26, 33, 37]. The focus of CSPEC, in contrast, is on dynamic sharing of state between processes on a single node, and on the patterns that help developers construct proofs for different styles of concurrency. CSPEC does not address node failures.

IronFleet [16] uses the notion of trace inclusion and movers in their reduction argument, which has been machine-checked [19]. However, IronFleet's verified reduction argument is specialized for IronFleet's specific use case, and has a hard-coded list of movers: sending and receiving UDP packets, and acquiring and releasing locks [28]. In contrast, CSPEC is a general-purpose mover framework.

Mail servers. Affeldt and Kobayashi verified a part of a mail server written in Java, by manually translating the Java program into a Coq function, and verifying properties of the Coq function [1, 2]. They verified the SMTP receiver part of the mail server, but do not model the interaction between the mail server and the file system. We use CSPEC to verify CMAIL, which includes both delivery via SMTP as well as pickup via POP3, and prove that CMAIL correctly uses the file system.

Ntzik [32] developed a concurrent specification for POSIX file systems using a concurrent separation logic, and used it to reason about snippets of mail server code for spam filtering. In contrast, CMAIL is a fully operational concurrent mail server, with a complete specification and machine-checked proof of its implementation.

3 Goal and challenges

The goal of CSPEC is to allow developers to write specifications for concurrent systems software such as the mail server and to prove that an implementation satisfies the spec. The proof should ensure that every possible interleaving, no matter how unlikely, is handled correctly.

```
def deliver(user, msg):
  tmpname = "/tmp/%d" % getpid()
  f = open(tmpname, "w")
  f.write(msg)
  f.close()

  while True:
    mboxfn = "/var/mail/%s/%d" % (user, random())
    if link(tmpname, mboxfn) == ok:
      unlink(tmpname)
      return
```

Figure 1: Pseudocode for delivery in a Maildir-like mail server.

```
def pickup(user):
  files = readdir("/var/mail/%s" % user)
  messages = []
  for fn in files:
    f = open("/var/mail/%s/%s" % (user, fn))
    messages.append(f.read())
    f.close()
  return messages
```

Figure 2: Pseudocode for pickup in a Maildir-like mail server.

To illustrate why this is hard, consider a mail server running on top of a file system, as a prototypical example of concurrent systems software. A mail server performs two main operations: `deliver`, which accepts incoming messages and writes them to the file system, and `pickup`, which allows users to download their messages. A mail server typically runs many processes, which concurrently perform deliveries and pickups.

For instance, consider the Maildir-like [3] server shown in Figures 1 and 2. In Maildir, each user's mailbox is a directory containing one file for each message. Maildir does not use locks for most concurrency control; instead, `deliver` and `pickup` choose file names and issue file system operations that are carefully designed to avoid races.

`deliver` first writes the incoming message into a temporary file with a unique filename (based on the process ID) and then links the file into the user's mailbox directory with a randomly chosen name. If the link fails because the filename already exists (which can happen because of another delivery that chose the same random name), `deliver` retries it with a different filename. To read a user's messages, `pickup` calls `readdir` to list the files in the user's mailbox, and reads them one at a time.

Even though the code appears to be simple, it is carefully designed to handle many subtle interleavings of file system operations that arise when multiple concurrent processes invoke `deliver` and `pickup`. For example, `pickup` will not return any partially written messages to the user, because `deliver` will link a message into the user's mailbox only after it has been fully written to a file. As another example, two `deliver` processes will not overwrite each other's messages, because they choose distinct filenames in the temporary directory, and because they use `link` to atomically place a message into the mailbox directory if and only if the filename does not already exist.

```
Definition message := string.

(* Defines a new type, [Op], representing operations, where
   running an [Op retT] returns a value of type [retT]. *)
Inductive Op : forall (retT : Type), Type :=
  (* one operation is [Deliver], which takes two arguments,
     [u] and [msg], and returns a [bool] *)
| Deliver : forall (u : user) (msg : message), Op bool
| Pickup : forall (u : user),  Op (list (msgid * message))
| CheckUser : forall (u : user), Op bool
| Delete : forall (u : user) (id : msgid), Op unit.

(* The abstract state is a two-level map: from users to
   mailboxes, which are maps from IDs to messages. *)
Definition State := Map.t user (Map.t msgid message).

(* The semantics, defining valid transitions for operations. *)
Inductive step :
  (* Transitions depend on the operation being executed, the
     current PID, and the initial state .. *)
  forall `(op : Op retT) (pid : nat) (st : State)
  (* .. and determine the operation's return value (whose
     type depends on the operation) and the final state *)
         (r : retT) (st' : State), Prop :=
| StepDeliverOK : forall u msg pid id st mbox,
  (* if user [u]'s mailbox is [mbox] *)
  Map.MapsTo u mbox st ->
  (* .. and message ID [id] is not used in [mbox] *)
  ~ Map.In id mbox ->
  (* .. then the following is a valid transition: *)
  (Deliver u msg, pid, st) |->
      (true, Map.set u (Map.set id msg mbox) st)
| StepDeliverErr : forall u msg pid st,
  (Deliver u msg, pid, st) |-> (false, st)
(* Some transitions omitted for space reasons *)
| StepDelete : forall u id pid st mbox,
  Map.MapsTo u mbox st ->
  (Delete u id, pid, st) |->
      (tt, Map.set u (Map.remove id mbox) st)
where "(op, pid, st) |-> (r, st')" := step op pid st r st'.
```

Figure 3: Specification of the mail server. Code snippets in this paper have been simplified for readability; the full code of CSPEC and CMAIL is available at https://github.com/mit-pdos/cspec.

4 Approach to proving atomicity

CSPEC's approach to verifying concurrent software is to specify the atomic semantics of operations such as `deliver` and `pickup`, and then prove that their implementations, such as the code shown in Figure 1 and Figure 2, meet their specs.

To use CSPEC, a developer first specifies the desired behavior of each operation if it were to execute atomically; then writes code in CSPEC to achieve this behavior, even when running concurrently; and finally the developer proves that the code indeed meets the atomic spec in all possible cases, with the help of CSPEC's proof patterns.

For example, Figure 3 shows the atomic spec of the main operations in CMAIL. The first statement in Figure 3 defines the set of allowed operations, using a Coq inductive type called `Op`. The next statement defines the abstract state. The last statement defines the semantics, by describing the allowed transitions using a Coq inductive type. For example, the first allowed transition, `StepDeliverOK`, states one legal way for a `Deliver` operation to execute with some arguments `u` and `msg`. Namely, if `u`'s mailbox is `mbox`, then `Deliver` adds the incoming message with a new identifier `id` in the user's mailbox. Here, `Deliver` denotes the primitive operation in the semantics, whereas

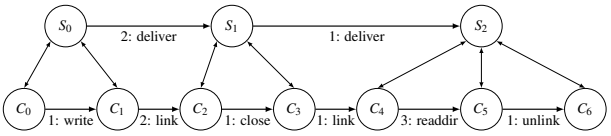

Figure 4: Example diagram of a simulation proof, connecting code from Figure 1 with the spec from Figure 3. In the example, processes 1 and 2 each deliver a message concurrently, while process 3 is running `pickup`.

the pseudocode of `deliver` from Figure 1 describes a possible implementation of `Deliver`.

To understand why proving correctness is hard, consider the approach based on a *simulation* proof [30], used by many frameworks [16, 25, 35]. The idea is to define an *abstraction relation* that connects the spec-level states with code-level states, and to show that this relation is preserved by every possible transition at the code level.

Figure 4 shows a simulation argument for one execution of the mail server: two processes concurrently delivering a mail message. At the bottom are code-level states, representing the states and transitions of the file system, corresponding to code from Figure 1 (in this example, the mail server is handling an incoming SMTP message). At the top are spec-level states, representing the abstract state and transitions of the mail server, corresponding to the specification in Figure 3. The abstraction relation, shown as vertical arrows, captures the correspondence between the abstract spec-level state (set of messages in each user mailbox) and the concrete code-level state (files and directories representing the mailboxes). For each code-level transition, the simulation proof shows that the new code-level state corresponds to a spec-level state after zero or more spec-level transitions.

Proving atomicity using simulation turns out to be hard, because it requires the developer to consider many possible interleavings, such as the one shown at the bottom of Figure 4 among others. This leads to a secondary complication: the abstraction relation must describe all reachable code-level states, including ones in which many processes are halfway through executing their updates.

We would like to reduce the problem of reasoning about concurrent execution to reasoning about sequential execution as much as possible. To provide some intuition for why this might work, consider `deliver` from Figure 1. We would like to ignore interleavings with other processes before `link` (lines 2-8) and after `link` (lines 9-11), because operations on lines 2-8 affect that process's temporary file, which is specific to that process's PID. Other processes will not interfere with it. However, the interleavings with respect to `link` (line 9) do matter, because the message created by `link` in the shared mailbox can now affect other processes running `deliver` or `pickup`.

To formalize this intuition, CSPEC uses the idea of left-, right-, and non-movers [27], which captures the notion that operations from different processes might (or might

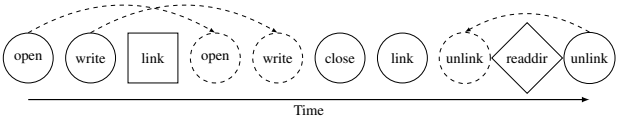

Figure 5: Example interleaving of file system operations executed by 3 separate processes: circles correspond to a process running `deliver`, squares correspond to another process running `deliver`, and diamonds correspond to another process running `pickup`. Dotted operations and arrows indicate re-ordering with the help of mover types.

not) commute with one another. Movers help CSPEC reason about atomicity, by proving that certain sets of interleavings all produce the same outcome, and hence that it suffices to consider just the interleaving where the code executes atomically.

Consider the interleaving in Figure 5, which depicts the code-level steps from the bottom of Figure 4. (Ignore the dashed elements for now.) In this example, the `open`, `write`, and `close` operations from process 1 (denoted by circles) are *right-movers*, which means that moving their execution to the right in the diagram (past the transitions of other processes) produces the same outcome. This is because `open` and `write` modify a temporary file that's named by the process ID and hence not accessed by any other process, and because `close` does not interact with other processes at all. Similarly, the `unlink` operation from process 1 is a *left-mover*, which means that it can be moved earlier (left) in the execution (past the transitions of other processes) without changing the outcome. However, note that the `link` operation from process 1 is neither a left- or right-mover (i.e., a non-mover), since moving it to the left or right can change the outcome by affecting a `readdir` from a concurrent `pickup`.

By using left- and right-movers in Figure 5, we can re-order the execution of `deliver` in process 1 to be atomic, as shown by the dashed elements in Figure 5. This re-ordering corresponds to a sequential execution of `deliver`, and allows us to prove that `deliver` can be thought of as executing atomically. We do the same style of reasoning for `pickup`, showing that we can rearrange operations so that they form a sequential execution of `pickup`, and then proving that the implementation preserves the atomicity of `pickup`. This further allows us to prove correctness of arbitrary interleavings of processes by considering just the sequential executions of `deliver` and `pickup`.

5 Design of CSPEC

This section provides an overview of CSPEC's design by describing what a layer is, how CSPEC defines correctness, and how a developer proves an implementation correct.

5.1 Layers

CSPEC's workflow involves defining *layers*. The spec of a layer has three parts: the set of operations (`Op`), the state manipulated by those operations (`State`), and the

```
Definition pathname := list string.

Inductive Op : forall (retT : Type), Type :=
| Read : forall (pn : pathname), Op (option string)
| Link : forall (src : pathname) (dst : pathname), Op bool
| Unlink : forall (pn : pathname), Op unit
| List : forall (dirpn : pathname), Op (list string)
(* Some operations omitted for space reasons *).

Inductive State : Type :=
| ST : forall (Files : Map.t pathname string)
              (Locks : Map.t pathname bool), State.

Inductive step : forall `(op : Op retT) (pid : nat)
  (st : State) (r : retT) (st' : State), Prop :=
| StepReadOK : forall pn fs pid msg locks,
  Map.MapsTo pn msg fs ->
  (Read pn, pid, ST fs locks) |-> (Some msg, ST fs locks)
| StepReadNone : forall pn fs pid locks,
  ~ Map.In pn fs ->
  (Read pn, pid, ST fs locks) |-> (None, ST fs locks)
(* Some transitions omitted for space reasons *)
| StepLinkOK : forall fs pid dst data pn locks,
  Map.MapsTo pn data fs ->
  ~ Map.In dst fs ->
  (Link pn dst, pid, ST fs locks) |->
      (true, ST (Map.set dst data fs) locks)
| StepLinkErr : forall fs pid dst pn locks,
  (Link pn dst, pid, ST fs locks) |-> (false, ST fs locks)
where "(op, pid, st) |-> (r, st')" := step op pid st r st'.
```

Figure 6: Low layer for the mail server example.

semantics, describing how each operation updates this state and what value it returns (the `step` relation).

For example, the top layer of the mail server is the spec shown in Figure 3. The bottom layer is the file system, partially shown in Figure 6. This layer defines the file system operations, the file system state (a tuple, called `ST`, consisting of a map representing the contents of all files, and a map representing whether each file is locked using POSIX file locking), and the results of each operation: how it updates the state and what values it returns.

Layers are an important modularity technique. Many proofs in CSPEC require considering all possible transitions made by other processes (e.g., when proving that an operation is a right- or left-mover). Doing so directly on top of the file system layer would be tedious, because there are many possible transitions (corresponding to many operations), and because the transitions operate in terms of low-level file system state. Re-defining the operations and state in an intermediate layer can simplify the proof, because the state is smaller and there are fewer operations to consider. For instance, to prove CMAIL, we decomposed it into 13 layers as shown in Figure 7, with each layer (except the bottom) implemented using the operations of the layer below it. As an example, Figure 8 shows the implementation connecting the MailboxTmpAbs and Deliver layers.

Connecting two layers requires writing code for every higher-level operation that uses only lower-level operations, and a proof that this code meets the layer's spec. CSPEC then links multiple layers together by chaining their implementations and proofs. It's much easier to do the proofs if they map onto CSPEC's proof patterns.

	Layer name	Operations	State	Pattern
	MailServerComposed	Deliver, Pickup, Delete, CheckUser (Figure 3)	Messages in user mailboxes (Figure 3)	
Per-user layers	MailServerPerUser	Per-user Deliver, Pickup, Delete	Messages in one user's mailbox	Part
	MailServerLockAbs	*same as above*	+ Lock on mailbox for serializing Pickup and Delete	Abs
	Mailbox	+ List and Read; - Pickup	*same as above*	Mov+Prot
	MailboxTmpAbs	*same as above*	Additional temporary directory	Abs
	Deliver	+ Create, Link, and Unlink; - Deliver	*same as above*	Mov
	DeliverListPid	+ Filtered List returning files with caller's PID	*same as above*	Mov+Prot
	MailFS	+ GetPID; - Filtered List	*same as above*	Mov+Prot
	MailFSStringAbs	*same as above*	File names are strings instead of pairs	Abs
	MailFSString	Operations now in terms of string names	*same as above*	Mov
	MailFSPathAbs	*same as above*	Per-user file system	Abs
	MailFSPath	Per-user file system operations	*same as above*	Mov
	MailFSMerged	File system operations (Figure 6)	File system (Figure 6)	Part+Abs

Figure 7: Layers used for verifying the mail server. The operations column describes the `Op` type for that layer. The state column describes the abstract state, `State`, over which the layer's semantics are defined. The pattern column lists the CSPEC proof patterns (described in §6) used for connecting two layers. This layering corresponds to "plan 1" described later on in §8.1; not shown is one intermediate layer used for "plan 2."

For example, some of the lower layers in Figure 7 deal with how the mail server state is encoded using directories and file names. That is, these layers have a different definition of `State`, but typically the same list of operations (i.e., same `Op`) as higher layers. All of the layers above, however, assume that the mail server's mailbox is completely disjoint from the temporary directory, and assume that file names are pairs of process ID and message ID (i.e., their `State` is just a map, as in Figure 3). As a result, the code and proofs at higher layers need not worry about file name encoding, pathnames, traversing directories, etc.

5.2 Defining correctness

CSPEC's definition of correctness revolves around the observable behaviors allowed by the specification, and the observable behaviors that can be produced by the implementation. CSPEC uses a standard notion of correctness: it requires that the behaviors of the implementation be a subset of the behaviors allowed by the spec.

More formally, CSPEC models the interaction with the outside world using the notion of *events* [24, 30]. The idea is to annotate operations that interact with the outside world (e.g., accepting a connection, reading or writing network messages, closing a connection, etc) as producing *events*. These events reflect the external behavior of our system: SMTP messages coming in and being acknowledged, and POP3 requests coming in and getting responses. The sequence of events produced by a system thus defines its externally visible behavior, which we call a *trace*.

CSPEC defines correctness of an application by requiring that the traces of events produced by the application when using the concurrent implementation of operations (e.g., `deliver` and `pickup`) must be a subset of the traces that can be produced by the application using the specification of those operations (e.g., Figure 3 for the mail server). In other words, if the actual implementation of the system can exhibit some behavior, then this behavior must be allowed by the atomic specification.

```
Definition deliver_core (msg : message) :=
  ok <- Call (DeliverOp.CreateWriteTmp msg);
  match ok with
  | true =>  ok <- Call (DeliverOp.LinkMail);
             _  <- Call (DeliverOp.UnlinkTmp);
             Ret ok
  | false => _ <- Call (DeliverOp.UnlinkTmp);
             Ret false
  end.

Definition compile_op `(op : MailboxOp.Op T) :=
  match op with
  | MailboxOp.Deliver msg => deliver_core msg
  | MailboxOp.Read pn     => Call (DeliverOp.Read pn)
  | MailboxOp.Delete pn   => Call (DeliverOp.Delete pn)
  ...
  end.
```

Figure 8: Implementation connecting the MailboxTmpAbs and Deliver layers.

Trace inclusion is a good fit for specifying concurrent systems, compared to some of the alternative approaches that have been used by recent systems, such as postconditions [6]. Postconditions allow specifying the return values from a procedure, but this does not help with procedures that never return, such as the mail server that accepts incoming connections in an infinite loop.

CSPEC also uses the notion of trace inclusion to define the correctness of intermediate layers, such as the 13 layers used in CMAIL. Transitively, if each layer produces a subset of traces allowed by the layer above it, the entire stack of layers is correct: the traces produced by the bottom-most code are a subset of traces allowed by the top-most specification.

5.3 Implementation

An implementation is a module that provides one function, `compile_op`, which implements higher-level operations in terms of lower-level operations. For instance, CMAIL has 12 such implementations, connecting its 13 layers. Implementations of multiple layers can be chained together; for instance, CMAIL chains together its implementations to translate high-level operations like `Deliver` and `Pickup`

```
Lemma createwritetmp_right_mover : forall data,
  right_mover DeliverRestrictedAPI.step
              (DeliverOp.CreateWriteTmp data).
Proof.
  unfold right_mover; intros.
  ...
Qed.

Lemma unlinktmp_left_mover :
  left_mover DeliverRestrictedAPI.step
             (DeliverOp.UnlinkTmp).
Proof.
  split; eauto.
  ...
Qed.
```

Figure 9: Example lemmas about movers that arise in verifying the implementation of the MailboxTmpAbs layer on top of the Deliver layer. `DeliverRestrictedAPI.step` refers to a restricted version of the semantics of the `Deliver` layer (using the protocol pattern from §6.2), where the filename of any file linked into a user's mailbox must contain the PID of the process that called `link()`.

into low-level file system operations from Figure 6, such as the pseudocode shown in Figure 1 and Figure 2 (except that our actual implementation is in Coq, which is not as easy to read as the Python-like pseudocode).

To produce runnable code, CSPEC extracts this code to Haskell using Coq's code extraction facility, and replaces the low-level operations with actual file system calls. An unproven driver, written in Haskell, interfaces with the network (e.g., accepts connections using sockets) and calls the appropriate top-level operations. To verify the driver would require verifying the parsing of SMTP and POP3 messages, which we didn't do because it has little to do with concurrency. Finally, the Haskell compiler produces an ELF executable.

5.4 Proving

Verifying the implementation entails proving that the code generated by `compile_op` correctly implements every high-level operation in terms of the lower-level operations. This includes proving that the code preserves the atomicity of high-level operations, given the atomicity of the lower-level operations. To make this task easier, CSPEC provides several proof patterns that encapsulate proof techniques to prove theorems about the behavior of a concurrent system.

For instance, the mover approach described in §4 is one such technique. It allows the developer to prove that certain operations are atomic. Figure 9 shows the lemmas needed to prove the atomicity of `deliver_core` of Figure 8. The lemmas state that `CreateWriteTmp` is a right mover and `UnlinkTmp` is a left mover, and the developer must write a proof in Coq (and checked by Coq) to show that this is true. CSPEC provides a general-purpose theorem (discussed in §6) that translates these developer-proven lemmas into a proof that the entire implementation of `deliver_core` executes atomically.

Note that `compile_op` in Figure 8 translates many operations one-to-one to lower-level operations. It is trivial to prove that they are atomic because the lower-level operations are atomic, and CSPEC does this automatically.

CSPEC chains the proofs of each layer's `compile_op` to provide an end-to-end proof that the resulting executable system meets the top-level atomicity specification.

6 CSPEC's proof patterns

CSPEC provides a library of proof patterns that help in proving that the code connecting two layers is correct. This section presents each proof pattern in turn.

6.1 Mover pattern

The key pattern provided by CSPEC for reasoning about concurrency is the *mover pattern*. As we saw in §4, this reduces the problem of reasoning about many interleavings (i.e., concurrent execution) to a combination of reasoning about just one interleaving (i.e., atomic sequential executions) and proving that certain operations are left- or right-movers.

For instance, consider the implementation of `Deliver` shown in Figure 8 as `deliver_core`. Running this implementation concurrently with other `Deliver` and `Pickup` operations can produce many interleavings, since there are no locks. It is not even possible to enumerate all the possible interleavings, since there can be an arbitrary number of concurrent processes.

Intuitively, we can reason about the execution of `deliver_core` by observing that the `link` operation is the commit point. That is, before `link` other processes are not affected (e.g., they cannot observe partially delivered messages), and after `link` other processes can observe the delivered message (if `link` succeeds).

Equivalence. To formalize this line of reasoning, CSPEC reasons about *equivalent executions*—that is, two interleavings that must produce the same trace of events. For example, changing the order of the `unlink` in `Deliver`, with respect to other processes, produces the same trace.

To prove the atomicity of a procedure using CSPEC's mover pattern the developer shows that certain operations are left- or right-movers with respect to other operations, and that the code of the procedure consists of these left- and right-movers operations in a certain order. Given these lemmas, CSPEC provides a theorem that proves the equivalence of other interleavings.

Movers. CSPEC models the concurrent execution of an overall system by repeatedly executing one operation from some process, which leads to a particular *execution sequence*. Treating an entire operation as a single transition captures the idea that operations are atomic. The choice of the process whose operation is executed at each point in this execution sequence determines a particular interleaving. By considering all processes at

```
Definition right_mover step `(opA : Op TA) :=
  forall `(opB : Op TB) st0 st1 st2 pidA rA pidB rB,
    pidA <> pidB ->
    step opA pidA st0 rA st1 ->
    step opB pidB st1 rB st2 ->
      exists st1',
        step opB pidB st0  rB st1' /\
        step opA pidA st1' rA st2.
```

Figure 10: Definition of the right mover.

```
Theorem trace_incl_movers : forall `(p : proc Op T),
  right_left_mover_pattern p -> trace_incl p (Atomic p).
```

Figure 11: Mover pattern theorem. `proc Op T` is a type denoting a procedure that returns a value of type `T` and can invoke operations described by `Op`. `Atomic p` denotes a procedure that atomically executes `p`.

each point in the execution sequence, CSPEC considers all possible interleavings between concurrent processes.

Figure 10 formally defines what it means for a certain operation, `opA`, to be a right-mover. Specifically, it considers every possible execution where `opA` is followed by some other operation, `opB`, from a different process (with process ID `pidB`). In this execution, `opA` changes the state from `st0` to `st1`, and `opB` changes the state from `st1` to `st2`. In order for `opA` to be a right-mover, it must be possible to swap `opA` with `opB` in this execution: that is, if `opB` ran first, it must produce some state `st1'` such that `opA` will then produce `st2`, and `opB` and `opA` produce the same return values `rB` and `rA` respectively. Left movers are defined similarly (there are some subtle differences that we discuss later). As an example, Figure 9 showed how one layer of CMAIL uses these definitions.

Showing that an operation O is a left- or right-mover requires considering how O interacts with every possible operation from another process. Layers help by making it possible to define the operations in a way that makes it easier to prove that other operations commute.

Composing movers. In order to reason sequentially about the execution of a procedure, its code must consist of a sequence of right-movers, followed by zero or one non-movers, followed by a sequence of left-movers. This structure allows CSPEC to show that any possible execution sequence is equivalent to one where the procedure executes sequentially, with no intervening operations from other processes. Specifically, CSPEC provides a theorem, shown in Figure 11, stating that any trace produced by procedure `p` is also produced by procedure `Atomic p` (which executes `p` in a single atomic step), as long as `p` follows the above mover pattern.

CSPEC proves this theorem by moving all of the right-movers to the right and all of the left-movers to the left, so that they appear to execute sequentially with the optional non-mover in the middle. The non-mover (for example, `link` in `Deliver`) is the commit point of the operation.

Left-mover challenges. Proving the theorem in Figure 11 is difficult, and required addressing several challenges with the formalization of left-movers.

First, operations can be left-movers just with specific arguments or just in specific states. For example, the implementation of `Pickup` first lists the files in a user's mailbox directory and then opens and reads the files one at a time, as shown in Figure 2. Here, the `open` operation is a left-mover only if called with the pathname of a file in the user's mailbox. (The implementation of `Pickup` holds a lock to prevent concurrent deletes, but does not prevent concurrent deliveries.) It is not a left-mover if called with a filename in the temporary directory, because opening a temporary file might succeed or fail depending on what a concurrent `Deliver` does in the temporary directory. Furthermore, `open` is a left-mover only if the file already exists *before* the other operation (i.e., the operation being re-ordered with respect to the `open`). Otherwise, this other operation might be `Deliver` creating the file in question in the mailbox.

CSPEC supports state- and argument-dependent left movers by restricting the states and arguments that have to be considered by the left-mover. Specifically, CSPEC requires the left-mover to consider only those states and arguments that can arise after executing the prefix of the operations leading up to the left mover. For instance, the procedure `p` shown in Figure 11 may be composed of several right-movers, followed by a non-mover, followed by several left-movers. The left-movers have to consider only the states that can arise after the right-movers and the non-mover have executed.

To take advantage of state-dependent left-movers, the developer first states an invariant that is established by executing the right-movers followed by the non-mover. The developer then proves a lemma that, starting from any state, executing the right-movers followed by the non-mover establishes this invariant. Finally, when reasoning about a left-mover, the developer can invoke this lemma to prove that the state observed by the left-mover satisfies the invariant.

Second, CSPEC's model of operations allows for operations to be *disabled*: that is, the semantics forbids an operation to execute in a given state. This is represented by a `step` relation that does not provide any legal transitions for a particular operation and a particular state. In the top-level and lowest-level layers such restrictions do not appear, because CMAIL can always deliver and pickup mail and the file system can always execute an operation (even if only to return an error). However, disabled operations are helpful in intermediate layers, in order to prove that other processes follow certain rules.

The simplest example is a lock that protects memory accesses. Reads and writes to memory protected by a lock commute with other threads, because those threads cannot access the locked memory. By taking advantage of the fact that reads and writes are disabled for other threads that do not hold the lock, CSPEC allows a proof that

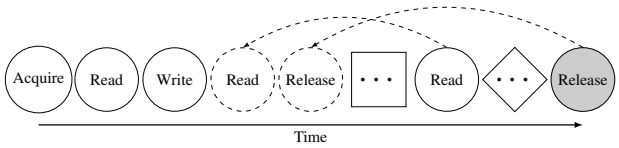

Figure 12: Use of left movers in an example process that acquires a lock, reads a variable, writes a variable, then reads a variable again and releases the lock. The gray shading of the `Release` on the right indicates that, although the `Release` is part of the code for the round process, the execution sequence shown is one where the `Release` never gets around to executing (e.g., due to other threads preempting it).

reads and writes are movers. (CSPEC's protocol pattern, described in §6.2, allows a developer to show that it is correct to assume that certain operations are disabled.)

Disabled operations complicate the notion of a left-mover, because moving an operation to the left requires showing that it can be executed earlier, which requires showing that it is enabled earlier. Consider a simple example shown in Figure 12. Reading from a locked memory region requires that the caller hold the lock. Moving the second read earlier requires showing that the caller holds the lock at that point. CSPEC deals with this by requiring a proof that a left-mover is *stably enabled*. This means that if the operation was enabled in a certain state (e.g., at the point where the second read actually ran in Figure 12), then it must be enabled in a prior state before another operation from a different process ran (e.g., in its dashed location in Figure 12). The read is stably enabled because the process must have held the lock, and no other process can acquire or release this process's lock.

The final challenge has to do with liveness. For example, the `Release` in Figure 12 is a left-mover, and we would like to use this fact to make the entire sequence of five operations into an atomic step. However, in our example, `Release` never actually ran (i.e., it is not part of the execution sequence). This might be because the scheduler is not fair and repeatedly ran other processes instead. How can we re-order the `Release` if it does not appear in the execution sequence to begin with?

To deal with this problem, CSPEC's proof considers all possible execution sequences. If an execution sequence contains the `Release`, the proof uses the fact that it is a left-mover to move it left. However, if an execution sequence does not contain the `Release` (i.e., if the `Release` never runs), then it is safe to insert that `Release` into the execution sequence. Stable enablement of left movers guarantees that `Release` is enabled at the point where we would like to insert it (i.e., the `Release` cannot have been waiting for another process to do something), and `Release` being a left-mover guarantees that other operations from this execution sequence will not be affected by inserting this `Release` (because they never saw the lock being released in the first place).

```
Definition op_abs :=
  forall '(op : Op T) st st' ST pid r,
    absR st ST ->
    lo_step op pid st r st' ->
      exists ST',
        absR st' ST' /\ hi_step op pid ST r ST'.
```

Figure 13: Definition of abstraction.

6.2 Protocol pattern

Proving that operations are left- or right-movers sometimes requires reasoning about what other processes will do, not just about what operations they have. In the lock example above, proving that memory accesses are movers while holding the lock requires knowing that other processes will not access the same memory while this process is holding the lock. To reason about such examples, CSPEC requires the developer to define a *protocol*, which is a restricted version of the `step` execution semantics that disables certain transitions. In the lock example, this restricted semantics requires that the caller hold the lock in order to read or write memory. With this restricted `step` relation memory accesses are movers, because other processes are not allowed to access the same memory location while not holding the lock.

In reality, nothing prevents another process from accessing memory without holding the lock. Thus, a proof that is sound to use the restricted semantics requires a proof that all users of the API correctly follow this protocol. Specifically, this entails proving that any execution of a process's code on the unrestricted semantics is also a valid execution on the restricted semantics.

In theory, this requires reasoning about many interleavings. In practice, however, the reason that a procedure follows a protocol is often simple (e.g., syntactically, the program never calls `Release` unless it called `Acquire` first). Thus, the proof needs only limited reasoning about the execution of other processes. In the locking example, proving that a process reads or writes memory only while holding a lock requires just one helper lemma: that other processes will not release a lock held by this process.

6.3 Abstraction pattern

To connect layers with different types of states, CSPEC provides an abstraction pattern. The abstraction pattern requires the developer to define an *abstraction relation* that connects low-level and high-level states, and to prove that every operation preserves this relation. This pattern is a specialized version of a standard simulation proof: it requires that the operations remain the same.

Figure 13 formally defines the proof obligation for the abstraction pattern. It requires a proof that, for every operation `op`, if `op` runs from state `st` to `st'` in the low-level semantics, and low-level state `st` corresponds to high-level state `ST` according to the abstraction relation `absR`, then there's a state `ST'` that corresponds to `st'` such

that the same op runs from ST to ST' with the same return value.

The rest of this subsection describes two stylized uses of the abstraction pattern that we have found particularly useful in developing CMAIL and the x86-TSO locked counter example.

Invariant. The abstraction pattern allows a developer to prove that a layer follows an invariant: some property of states at that layer that is maintained by that layer's semantics. This in turn can help the developer apply other patterns, such as movers or the protocol pattern.

Operationally, the developer first specifies an invariant by defining a layer whose semantics require the invariant to hold in the initial and final state of every operation; the operations and the type of state remain the same. The developer then defines an identity abstraction relation (connecting states one-to-one). Finally, the proof of the abstraction relation shows that, if the invariant holds in some state, running any operation results in a state that also satisfies the invariant.

Error state. The abstraction pattern can also allow a developer to defer reasoning about unreachable states by defining an explicit error state. This is useful at lower-level layers, which have insufficient information to prove that certain states are unreachable (e.g., because it is up to the implementation of higher layers to avoid those states). This is simpler than an alternative plan that fully describes what happens in these states, and allows subsequent layers to treat all of these error states identically.

Operationally, the developer defines a protocol that they expect to follow (much as in the protocol pattern from §6.2), and augments the state with a designated *error* state. The developer then modifies the execution semantics so that, if the protocol is not followed, the execution transitions into the error state. Once the execution enters the error state, it remains in that state forever.

To connect an error-state layer to a lower layer without an error state, the developer defines an abstraction relation that allows the high-level error state to correspond to any low-level state. To connect two layers with error states, the developer defines an abstraction relation that connects the error states at the two layers. To finally dismiss the error state, the developer uses the protocol pattern to show that an implementation never enters the error state, and thus the error state is unreachable.

6.4 Other patterns

Retry loop. CSPEC provides a specialized pattern for reasoning about retry loops. For example, when the mail server is delivering a message into a mailbox, it guesses a name that is unlikely to exist (using the current timestamp), and attempts to link the new message under that name. If `link` returns an error (i.e., the name already exists), CMAIL guesses a new filename and retries.

Component	Lines of code/proof
Core: processes, layers, etc.	4,594
Proof patterns	2,117
Helper: Maps, Sets, etc.	2,869
Total	9,580

Figure 14: Combined lines of code and proof for CSPEC components

The retry loop pattern requires a proof that the body of the loop either has the correct effect (such as delivering the message into a mailbox) and exits the loop, or has no effect and retries. This allows CSPEC to prove that executing the loop is equivalent to just running the body once, at exactly the right time (when it finally succeeds), because it can provably ignore all previous attempts (since they must have had no effect).

Partitioning. CSPEC provides a partitioning pattern to reason about disjoint parts of the state. For example, CMAIL has a separate mailbox for every user. Without explicit support for partitioning, the developer would need to reason about pairs of users at every layer of CMAIL—for instance, showing that an operation is a right-mover would require considering concurrent operations both on the same mailbox and on other mailboxes.

To use CSPEC's partitioning pattern, the developer implements and proves layer `A` on top of layer `B`, using CSPEC's other patterns, where `A` and `B` represent a single shard of the overall system state. For example, the core of CMAIL implements the per-user `MailServerPerUser` layer on top of the per-user `MailFSPath` layer, as shown in Figure 7. The developer must also specify how these shards are named (e.g., by string username in the case of CMAIL). The partitioning pattern turns this proven single-shard implementation into a proven implementation for many shards (e.g., all users in CMAIL).

As shown in Figure 7, cross-mailbox operations show up just at the top and bottom layers of the CMAIL stack. At the bottom layer, the proof must show that mailboxes are correctly partitioned in the file system—that is, each mailbox gets its own directory that is independent of all other mailbox directories. At the top level, the developer must specify and prove how the entire state of the system can be decomposed into per-user partitions. This is straightforward for CMAIL because the top-level abstract state (Figure 3) consists of a mailbox per user.

7 Implementation

We implemented CSPEC in Coq. Figure 14 shows the lines of code, specification, and proof for the major components. Developers implement, specify, and prove their concurrent software in Coq, and CSPEC produces executable code using Coq's extraction support to Haskell. Our prototype of CSPEC and CMAIL is available at https://github.com/mit-pdos/cspec.

One technical difficulty in implementing CSPEC is that Coq (like many other formal reasoning systems) makes it cumbersome to reason about infinite objects (i.e., Coq's `CoInductive`), as opposed to arbitrary-sized objects (i.e., Coq's `Inductive`). This made it hard for us to model the possibly infinite traces of events produced by the execution of a concurrent system.

To deal with this, we borrowed an idea from Lynch [29: §13], taking advantage of the fact that CSPEC is targeting only safety properties. A violation of safety can be observed in a finite prefix of the trace. Thus, we define trace inclusion in Coq for possibly infinite traces as trace inclusion for every finite prefix of that infinite trace.

8 Evaluation

This section answers five questions to evaluate CSPEC:

- Can CSPEC enable developers to specify, implement, and verify concurrent software? §8.1 answers this in the context of CMAIL, and §8.2 demonstrates that CSPEC's patterns are also applicable for a different style of concurrency: namely, weak shared memory.
- Can software developed using CSPEC actually achieve speed-ups by taking advantage of concurrency? (§8.3)
- How much effort is required to use CSPEC? (§8.4)
- How important are CSPEC's patterns? (§8.5)
- What are the trusted components of CSPEC and CMAIL? (§8.6)

We answer the above questions by exploring two case studies built using CSPEC: a concurrent mail server (CMAIL) and a concurrent counter that uses locks implemented on top of an x86-TSO memory model.

CMAIL is a simple but complete mail server that supports SMTP and POP3. It runs on top of any file system on Linux and we have tested its compatibility with several SMTP and POP3 clients, including the SMTP library in Go, and the postal and rabid mail server benchmarks. CMAIL lacks sophisticated features found in standard mail servers, such as spam filtering, logging, TLS support, etc.

8.1 Verifying CMAIL

To show that CSPEC enables reasoning about concurrency, we give examples of concurrency from our two case studies. This subsection describes the examples of concurrency from CMAIL, and the next subsection describes our experience verifying an atomic counter on top of x86-TSO weak memory.

Figure 15 summarizes the examples of concurrency from CMAIL, by describing pairs of processes that might run concurrently, the state that they might access concurrently, the plan for dealing with this concurrent execution, and how we as developers were able to use CSPEC to formally reason about the correctness of this concurrent interaction. The rest of this subsection describes these examples in more detail.

Deliver/Deliver: temp directory. Accepting an incoming message requires CMAIL to first write it to a temporary directory. However, there can be concurrent deliveries writing to the same directory at the same time. For correctness, a CMAIL process includes its PID in the names of its temporary files, which ensures two processes never conflict on files in the temporary directory. In CSPEC, we formally reason about this by showing that operations on the temporary directory always commute between different processes, because they have different PIDs in the filenames.

Pickup/Delete. If a user has two connections to CMAIL, and deletes a message on one connection while picking up messages via another connection, then the code for pickup, which lists and picks up messages, might discover halfway through that it cannot read a message file because the file has been deleted. CMAIL deals with this by acquiring a lock (using POSIX `flock`) on the user's mailbox, in both pickup and delete (but not in deliver; concurrency between deliver and pickup will be discussed next). We reason formally about this in CSPEC by first proving that CMAIL follows a protocol that requires holding a lock to delete any messages, and then showing that reading an existing message file is a both-mover while the lock is held.

Deliver/Pickup by another user. When CMAIL delivers or picks up mail for different users in different processes the concurrency plan is easy: these operations are independent because they operate on different mailboxes. In CSPEC, we show that operations on different mailboxes are commutative.

Deliver/Pickup by same user. A user can pick up (list and read) the messages in their mailbox while CMAIL is concurrently delivering new messages to that same mailbox (by creating files). CMAIL handles this like Maildir: it first creates new messages in a temporary directory, and then atomically renames them into the mailbox directory. When a user picks up their mail, CMAIL first calls `readdir` to list the files in the mailbox, and then reads the files in a loop. This is correct even in the presence of concurrent deliveries, because deliveries never delete existing files. To reason formally about this in CSPEC, we show that creating temporary files during delivery is a right-mover, and the atomic rename by delivery is a non-mover. On the pickup side, `readdir` is a non-mover, but all subsequent reads of existing files are left-movers.

Deliver/Deliver: files in mailbox, plan 1. Concurrent deliveries into the same user mailbox must ensure they pick different file names for the new messages. CMAIL implements two plans for this scenario, to demonstrate

Process 1	Process 2	State	Concurrency plan	CSPEC approach
Deliver message	Deliver message	Temp. directory	File names based on PID	Both-movers due to commutativity
List mailbox	Delete a message	Files in mailbox	Lock the mailbox directory	Protocol: both-movers while holding lock
Deliver to one user	Pickup by another user	Files in mailbox	Per-user directories	Both-movers due to commutativity
Deliver to one user	Pickup by same user	Files in mailbox	Atomic rename / readdir	Non-mover rename, non-mover readdir
Deliver to one user	Deliver to same user	Files in mailbox	List files by PID and pick next	List-per-PID is a mover
Deliver to one user	Deliver to same user	Files in mailbox	Retry link to random filename	Retry loop

Figure 15: Examples of concurrency from CMAIL supported by CSPEC.

how different approaches can work. In the first approach (which differs from Figure 1), filenames in the mailbox directory are based on the PID of the process that delivered the message. To pick an available filename, the delivery process calls `readdir` to list the directory, and chooses the next available filename that contains its PID.

Formally reasoning about this turns out to be tricky in two ways. First, `readdir` is not a mover, because its results can be affected by concurrent deliveries. To use mover-based reasoning, we implemented a function that filters the output of `readdir` and returns only the filenames of the caller's PID. This PID-filtered `readdir` function is a both-mover, because concurrent deliveries by different processes have filenames with different PIDs.

Second, in the presence of concurrent message deletion, even PID-filtered `readdir` is not quite a mover. We solve this by allowing it to return a superset of files that exist: that is, it must return all files that exist but can also return some non-existent files. This suffices because a filename that is not in the superset is guaranteed to not exist. This PID-filtered `readdir` is a right-mover in the presence of deletion (though not a left-mover), and so we can use the mover pattern to reason about its concurrent execution.

Deliver/Deliver: files in mailbox, plan 2. The second plan we implemented for concurrent deliveries to the same mailbox is to pick a random filename and try using it. POSIX `link` returns an error if the file already exists, so in case of an error CMAIL picks a new random filename (actually, it uses the current timestamp) and retries. To reason about this we use the retry pattern, showing that `link` either succeeds or returns an error and has no effect.

8.2 Verifying a counter on x86-TSO

CMAIL's concurrency model is based on processes with private memory sharing a file system. To demonstrate that CSPEC can also be used to reason about other concurrency models, we developed a model of x86-TSO [34], the predominant memory model of x86 processors. On top of x86-TSO, we implemented a lock, and used the lock to implement a counter. The lock implementation is a loop around an atomic test-and-set instruction, which includes an implicit write barrier (on x86, this corresponds to a `LOCK` prefix on the test-and-set instruction). We used 10 layers to verify this counter, as shown in Figure 16.

The top layer is a counter with two atomic operations: increment and decrement. The bottom layer, `TSO`, models x86-TSO: there is a shared memory and a per-core store buffer, and individual cores can issue reads, writes, or atomic test-and-set instructions, as well as perform a write barrier to flush that core's store buffer. Every operation at this bottom layer allows any core to flush any part of its store buffer at any time.

One challenge in the `TSO` layer is that background flushes of store buffers can happen on any core at any time. To help address this challenge, we showed that the `TSO` layer is equivalent to the `TSODelayNondet` layer which does not allow store buffer flushes on write (instead, postponing them to a subsequent read, barrier, or test-and-set).

The `LockOwner` layer introduces abstract state to keep track of which core owns the lock, using the abstraction pattern. Our intention is that the lock protects reads and writes to a shared memory location. However, this proper use of the lock is not established until a higher layer (namely, `Lock`). As a result, the `LockOwner` layer uses an explicit error state (§6.3) to indicate when the locking rules are not being followed. This error state is proven to be unreachable in the `Lock` layer (using the protocol pattern).

The `LockInvariant` layer additionally tracks the previous lock owner as part of the state. This is necessary because the implementation of lock release does not issue a write barrier. As a result, even though the lock may have been released, the lock value in shared memory may still appear to be locked, and pending writes to shared data are also in some core's store buffer. By tracking the previous lock owner, the `LockInvariant` layer states an invariant that either there are no pending writes to the lock or shared data in any core's store buffer, or they are in the previous lock owner's store buffer. The next layer, `SeqMem`, builds on this invariant to present a sequentially consistent view of shared memory, abstracting away the store buffer details.

The `RawLock` layer introduces an `Acquire` operation that waits until it can acquire the lock. This layer is implemented on top of `SeqMem` by repeatedly trying to acquire the lock in a loop. The proof is constructed with the help of CSPEC's loop pattern.

Layer name	Operations	State	Semantics	Pattern
Counter	Inc, Dec	Counter value	Atomic Inc and Dec	Abs
LockedCounter	Inc, Dec	Counter value + lock	Atomic Inc and Dec	Mov
Lock	Read, Write, Acquire, Release	SC memory + lock	Read/Write allowed only while holding lock	Prot
RawLock	Read, Write, Acquire, Release	SC memory + lock	Read/Write allowed any time	Loop
SeqMem	Read, Write, TryAcquire, Clear	SC memory + lock	Single value in memory, no SBs	Abs
LockInvariant	Read, Write, TryAcquire, Clear	Mem + SBs + cur/prev LOs	SBs empty except current or prev lock owner	Abs
LockOwner	Read, Write, TryAcquire, Clear	Mem + SBs + current LO	TSO + error state for violating lock protocol	Abs
TAS_TSO	Read, Write, TryAcquire, Clear	Mem + SBs	TryAcquire grabs lock; Clear releases lock	Mov
TSODelayNondet	Read, Write, TestAndSet, Barrier	Mem + SBs	Reduced number of background SB flushes	Abs
TSO	Read, Write, TestAndSet, Barrier	Mem + SBs	SB may choose to flush on every operation	

Figure 16: Layers used for verifying the x86-TSO locked counter. The operations column describes the `Op` type for that layer. The state column describes the abstract state, `State`, over which the layer's semantics are defined. The semantics column describes the semantics. The pattern column indicates which CSPEC proof pattern is used in connecting adjacent layers. "SC" stands for sequentially consistent. "SB" stands for store buffer. "LO" stands for lock owner.

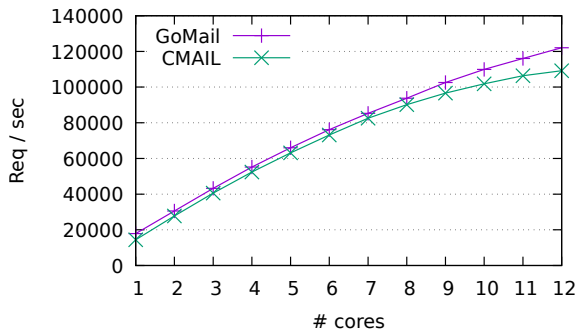

Figure 17: Throughput of CMAIL with a varying number of cores.

8.3 Speedup

To demonstrate that CMAIL can take advantage of multiple cores because it executes concurrently, we run a mixed workload of SMTP deliveries of new messages and POP3 requests that read and delete messages. The mix is an equal ratio of new messages being delivered and existing messages being read and deleted. Each request (delivery or pickup) chooses one of 100 users at random. Although CMAIL supports full-fledged SMTP and POP3 over the network, we simulated SMTP and POP3 requests on the same machine to stress CMAIL's scalability. We ran the experiment on a server with two Intel Xeon CPU, each with 6 cores running at 3.47 GHz. To keep the disk from being the bottleneck, we ran CMAIL on Linux tmpfs. To compare the performance of CMAIL to that of an unverified implementation, we implemented an equivalent mail server in Go, called GoMail, and measured its performance in the same setup.

Figure 17 shows the performance (in requests per second) for different numbers of cores of both CMAIL and GoMail. The results show that CMAIL scales well with more cores. This is because tmpfs can execute the file system calls of the different CMAIL processes in parallel. In terms of absolute performance, CMAIL achieves 81-97% of GoMail's throughput, depending on the number of cores.

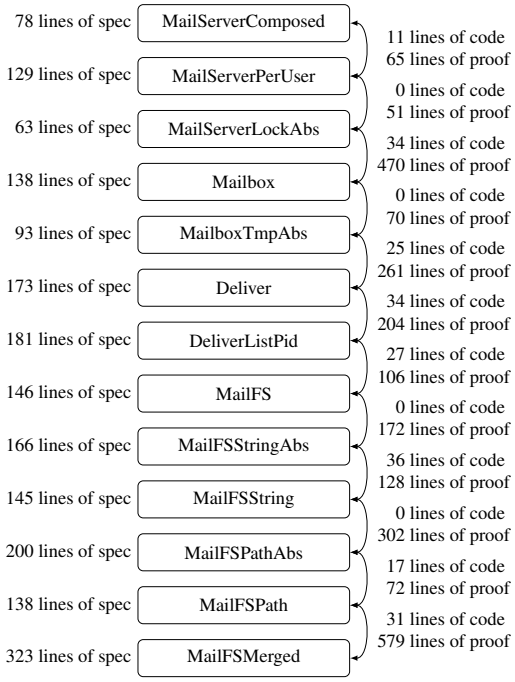

Figure 18: Combined lines of code and proof for CMAIL layers. The number next to arrow indicates number of lines of code and proof for the implementation connecting two layers.

8.4 Effort

Figure 18 shows the size of CMAIL: the lines of Coq code to specify each layer (i.e., define operations, state, and semantics) and the lines of Coq code required to connect layers (i.e., implement one layer in terms of a lower layer and prove the correctness of that code). Developing CSPEC and CMAIL took two people ∼6 months of part-time effort.

The figure shows that the effort required per layer is modest. Each layer spec is 100-200 lines of Coq code, which are largely repetitive, with only small differences between adjacent layers. Informally, the specs of adjacent layers differ in roughly half the lines, and even the differing lines are often similar (e.g., an extra state component is added everywhere). Better language support, perhaps

Proof pattern	# of uses in CMAIL	# of uses in x86-TSO
Movers	6	2
Abstraction	5	5
Protocol	3	1
Partitioning	2	0
Retry loop	1	1

Figure 19: Use of proof patterns in CMAIL and the x86-TSO example.

along the lines of CIVL's [22], could eliminate the repetition. A layer often maps a high-level operation directly onto a low-level operation, so it should be sufficient to write the spec only once. For example, CMAIL's `GetPID` is the same in each of the 13 layers.

The code and proof is sometimes shorter than the layer spec because some code takes advantage of CSPEC's patterns so well that it requires little additional proof effort. This is particularly true for the abstraction pattern that introduces additional state not seen at a higher layer (e.g., adding state for a lock that is hidden at a higher layer).

The most significant code and proof effort connects the MailServerLockAbs and Mailbox layers, where CMAIL implements atomic pickup. This requires a proof that pickup's file reads are left-movers, and inductive reasoning about a loop that reads all files. This is particularly hard because the file read is a state- and argument-dependent left mover, which requires reasoning about the set of files that exist in the system after `readdir` returns.

Evolution. To evaluate how hard it is to make incremental changes to a verified system in CSPEC, we report the effort it took us to make several significant changes to CMAIL as we were developing it. Initially our mail server supported POP3 retrieval but not deletion. Adding deletion support took about a day: we had to change some mover proofs because deletion made certain operations into non-movers. Our initial mail server used plan 1 to choose unique file names in a mailbox (see §8.1); implementing plan 2 using retry loops with `link` took us about a day. Finally, adding support for multiple users took us about a week. After a day, we realized that manually adding users to each layer was too tedious, and spent a week developing the partitioning pattern in CSPEC. Afterwards, supporting multiple users took about a day.

8.5 Patterns

Figure 19 shows the number of uses of a proof pattern in CMAIL and in the x86-TSO example. Typically each layer uses one proof pattern, but a few layers are split into several modules, each module using a distinct proof pattern. The results show that all patterns are important; that movers is the most commonly used pattern in CMAIL; and that abstraction is the most common pattern in x86-TSO.

8.6 Trusted computing base

Whether CMAIL does the right thing depends on several unverified assumptions and components. The first assumption is that the top-level specification (Figure 3) captures the right behavior. Second, the specification of the bottom layer (Figure 6) must be an accurate model of the underlying file system. Finally, the Haskell runtime and interpreter used to run CMAIL must behave appropriately.

CMAIL also requires Coq to be sound, but inside of Coq, CMAIL and CSPEC are fully proven. We used the `Print Assumptions` command in Coq to verify that the end-to-end theorem about correctness of CMAIL does not depend on any unproven axioms (aside from standard assumptions like Coq's functional extensionality).

9 Conclusion

CSPEC is a framework for verifying concurrent systems software. It uses mover types to simplify reasoning about both lock-based and lock-free concurrency, with the first fully machine-checked proofs. To further simplify proofs, CSPEC has layers and a library of proof patterns. CMAIL demonstrates that CSPEC can verify all the concurrency patterns in a Maildir-like mail server. Furthermore, we demonstrate that CSPEC's proof pattern can also be used to prove an atomic lock-based counter on top of x86-TSO shared memory. CMAIL achieves speedup on a multicore machine due to concurrency. We hope that CSPEC and its ideas will help others to verify concurrent software.

Acknowledgments

Thanks to Adam Chlipala, the PDOS research group, the anonymous reviewers, and our shepherd, Jon Howell, for improving this paper. This research was supported by NSF awards CNS-1563763 and CCF-1836712, and by Google.

References

[1] R. Affeldt and N. Kobayashi. Formalization and verification of a mail server in Coq. In *Proceedings of the 1st International Symposium on Software Security (ISSS)*, pages 217–233, Tokyo, Japan, Nov. 2002.

[2] R. Affeldt, N. Kobayashi, and A. Yonezawa. Verification of concurrent programs using the Coq proof assistant: A case study. *IPSJ Digital Courier*, 1: 117–127, Jan. 2005.

[3] D. Bernstein. Using maildir format, 2003. http://cr.yp.to/proto/maildir.html.

[4] A. Bouajjani, C. Enea, S. O. Mutluergil, and S. Tasiran. Reasoning about TSO programs using reduction and abstraction. arXiv:1804.05196

[cs.LO], Apr. 2018. Available at https://arxiv.org/abs/1804.05196.

[5] S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3), May 2007. Festschrift for John C. Reynolds's 70th Birthday.

[6] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–37, Monterey, CA, Oct. 2015.

[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Munich, Germany, Aug. 2009.

[8] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-2019, Microsoft Research, Feb. 2009.

[9] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, Edinburgh, UK, July 2010.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, Nov. 1978.

[11] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009.

[12] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, GA, Jan. 2009.

[13] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 653–669, Savannah, GA, Nov. 2016.

[14] R. Gu, Z. Shao, J. Kim, X. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[15] C. Hawblitzel. Personal communication, email, Mar. 2018.

[16] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Monterey, CA, Oct. 2015.

[17] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, San Francisco, CA, July 2015.

[18] C. Hawblitzel, E. Petrank, S. Qadeer, and S. Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, Feb. 2015.

[19] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving safety and liveness of practical distributed systems. *Communications of the ACM*, 60 (7):83–92, July 2017.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, Oct. 1983.

[21] J. Kim, V. Sjöberg, R. Gu, and Z. Shao. Safety and liveness of MCS lock–layer by layer. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 273–297, Nov. 2017. http://flint.cs.yale.edu/flint/publications/mcslock-tr.pdf.

[22] B. Kragl and S. Qadeer. Layered concurrent programs. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, Oxford, UK, July 2018.

[23] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] B. Lampson. Principles of computer systems, 2006. http://bwlampson.site/48-POCScourse/48-POCS2006Abstract.html.

[25] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[26] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 357–370, St. Petersburg, FL, Jan. 2016.

[27] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12), Dec. 1975.

[28] J. Lorch et al., 2016. https://github.com/Microsoft/Ironclad/tree/concur-tree/ironfleet/src/Dafny/Distributed/Common/Reduction.

[29] N. Lynch. *Distributed Algorithms*. Elsevier, 1996.

[30] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[31] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, Cambridge, MA, Nov. 1988.

[32] G. Ntzik. *Reasoning About POSIX File Systems*. PhD thesis, Imperial College London, 2017.

[33] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. In *Proceedings of the 45th ACM Symposium on Principles of Programming Languages (POPL)*, Los Angeles, CA, Jan. 2018.

[34] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[35] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[36] The Coq Development Team. *The Coq Proof Assistant, version 8.8.0*, Apr. 2018. URL https://doi.org/10.5281/zenodo.1219885.

[37] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.