ChanLib: Reconciling Choice, Rendezvous, Broadcast, and Buffering in a Mechanized Specification of Go Channels

Levi Redlin[†]
lredlin@wisc.edu
University of Wisconsin-Madison
Madison, Wisconsin, USA

Upamanyu Sharma
upamanyu@mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Jonas Kastberg Hinrichsen
jkhi@cs.aau.dk
Aalborg University
Copenhagen, Denmark

Tej Chajed
chajed@wisc.edu
University of Wisconsin-Madison
Madison, Wisconsin, USA

Abstract

Go is a widely used language with good support for concurrency, making it an attractive target for verification. Go programs frequently use *channels* to coordinate concurrent work, but existing verification frameworks for Go, such as Perennial and Gobra, have limited specifications for the Go channel API. Channels support a range of features with simple syntax but subtle semantics: unbuffered channels involve multiple linearization points, close is a broadcast primitive that can be used only once without a runtime panic, and `select` multiplexes channels using any of these features. Iris has a powerful methodology for specifying linearizable data structures where each operation is given a logically atomic triple describing a single atomic transition of an abstract state, but these features and their idiosyncrasies are particularly challenging to capture in this style.

This paper presents ChanLib, a library that expressively models channel behavior and provides specifications integrated with Perennial, which supports reasoning about all other aspects of Go verification. We verified a core specification based on Iris atomic updates against an executable model written in userspace Go, validated empirically against the Go runtime's own tests to capture the semantics faithfully. Using ChanLib, we verified a wide variety of programs such as the Mixtris theory of mixed-choice multiparty message passing, which hadn't been verified on a production language, and concurrent data structures spanning disparate uses of channels: `unbounded.go`, a buffer used in production in gRPC-Go, and an elimination stack that uses an unbuffered channel as its synchronization point. All results are mechanized in the Rocq prover, within the Perennial framework.

CCS Concepts: • **Software and its engineering → Concurrent programming structures**; • **Theory of computation → Program verification**; *Separation logic.*

Keywords: message passing, Go, verification, concurrency

ACM Reference Format:
Levi Redlin, Upamanyu Sharma, Jonas Kastberg Hinrichsen, and Tej Chajed. 2027. ChanLib: Reconciling Choice, Rendezvous, Broadcast, and Buffering in a Mechanized Specification of Go Channels. In *Proceedings of the 16th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '27), January 11–12, 2027, Mexico City, Mexico.* ACM, New York, NY, USA, 27 pages. https://doi.org/10.1145/3871849.3871890

1 Introduction

Go is a widely used language known for its efficiency, productivity, and good support for concurrency. It is commonly used for concurrent and distributed systems, making it an attractive target for verification. Go programs extensively use *channels*—the language's built-in primitive for message-passing concurrency—and `select`, a statement that waits on multiple channel operations and executes whichever is ready first: an empirical study of 865 popular Go projects found that 76% use channels and 66% contain at least one `select` statement [6]. Despite this, existing verification frameworks for Go provide only limited support for channels. Gobra [30] can verify basic send and receive operations but lacks support for `select` statements. Perennial [3], a state-of-the-art verification framework for Go built on top of the Iris separation logic [16], does not support channels.

Go concentrates in one primitive what is usually spread across several concurrent data structures. A *buffered* channel, made with `make(chan T, n)` for a capacity n, acts as a bounded queue where `ch <- v` enqueues and `<-ch` dequeues. An *unbuffered* channel, made with `make(chan T)`, instead

[†]Performed while working as a software engineer at Google LLC.



This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '27, Mexico City, Mexico
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3146-4/2027/01
https://doi.org/10.1145/3871849.3871890

couples sending and receiving. The two operations synchronize in tandem with no intermediate storage, in what is often called a *rendezvous*. `close(ch)` is a one-shot broadcast that every receiver observes. `select` provides a nondeterministic choice across multiple channel operations. Each has been studied on its own, and no one account covers them together. In this paper we give a single specification that covers all four and the interactions between them. We state it in Iris, a higher-order concurrent separation logic mechanized in Rocq, and prove it against an operational model of Go channels, which we validate against the test suite of Go's own channel implementation.

Iris is higher-order, so a specification can quantify over the resources a client chooses to send over a channel, and it provides ghost state and invariants for sharing those resources between goroutines.[1] We use both to give a channel a logical state that its clients reason about. A particular challenge is `select`. A single `select` may wait on buffered, unbuffered and closed channels at once, and whichever operation succeeds determines the control flow that follows. A case can succeed only if it is *ready*, which is a property of the state of the channel it names: a receive on an unbuffered channel, for instance, is ready only when a sender is already waiting. Channels resist the standard methodology for concurrent data structures [16, 26] in several ways.

Unbuffered channels are not linearizable. Send and receive on an unbuffered channel *rendezvous*: the value passes directly from sender to receiver with no intermediate storage, and each operation completes only after the other has begun. Neither takes effect at a single linearization point. Objects with this character, such as the *exchangers* used to build elimination stacks, are well studied [2, 5, 9]. We specify each unbuffered operation by two paths: a *fast path* that completes an exchange when a partner is already waiting, and a *slow path* that signals intent to exchange and then waits for notification that one occurred. A matched send and receive therefore each partly run after the other has begun. Blocking and non-blocking `select` have different semantics here, and we capture the difference by what a client has to show: a blocking `select` requires both paths to be safe, while a non-blocking one requires only the fast path, since it never signals intent of its own. Unbuffered channels are common: they account for 61% of channel creations in the projects studied by Dilley [6], and Go programs exploit the rendezvous. Our elimination stack case study (§ D.1) uses an unbuffered channel as the exchange point between concurrent push and pop, and its correctness rests on the guarantee that a matched pair overlaps in time.

[1] A goroutine is a lightweight thread of execution multiplexed to operating system threads by Go's runtime. For this work the distinction is not relevant, and a goroutine can be read as a thread.

Close is a broadcast primitive that can be used only once. Closing a channel signals that no more values will be sent: subsequent receives drain any buffered values and then return `(zero, false)`, where the `false` flag indicates the channel is closed. This makes close useful both as an EOF-like signal when streaming data and as a one-to-many broadcast notification—any number of receivers can observe that the channel is closed, with Go's `for range` loop or by checking the `ok` flag in `v, ok := <-ch`. Go's semantics reflect an intended usage of structured concurrency where values and signals are consumed exhaustively: receiving from a closed channel is always safe and returns a zero value, but `close(ch)` panics if `ch` is already closed (This is a recurring bug class in production Go [29]) and `ch <- v` on a closed channel panics unconditionally. A specification must therefore be asymmetric: receiving from a possibly-closed channel is freely available, while the *right to close* is an exclusive resource. We verify `unbounded.go` (§ 4), a production channel wrapper used in gRPC-Go that uses careful mutex discipline and two boolean fields to ensure the channel is closed exactly once. Panics are unsound in Perennial by design, so our specification requires proving that no close ever races with another close or with a send: verifying `unbounded.go` meant proving its protocol never leads to a double close or a send-after-close.

`select` *multiplexes unrelated channel operations.* The `select` statement waits on several channel operations—sends and receives, on different channels, buffered or unbuffered, possibly closed—and executes exactly one that is ready, chosen nondeterministically, running arbitrary code in the selected case's body. Every case must be safe to execute even though only one runs, which means the same separation logic resources must justify more than one case: requiring the separating conjunction of the cases' preconditions would be sound but unusable. Our rule instead requires the *overlapping conjunction* of one atomic update per case—the caller proves every case from shared resources, and soundness follows because the implementation consumes exactly one update. For a `select` such as

```
select {
    case ch1 <- v:   sendHandler()
    case v := <-ch2: recvHandler(v)
}
```

the specification requires proving one goal equivalent to the goal for `ch1 <- v; sendHandler()` and another equivalent to the goal for `v := <-ch2; recvHandler(v)`. The same logically atomic specification therefore serves both an ordinary send or receive and a case of a `select`.

Second, the multiplexed channels may have entirely different roles and specifications: for example, a single `select` might wait on a shutdown broadcast channel, a work-item queue, and a timeout signal—three channels with completely different specifications. Because each premise of our rule is only an atomic update, specifications built on top of ChanLib

discharge their own cases, and unrelated channels compose inside a single `select` without ad-hoc reasoning.

Third, a `select` with a `default` branch makes a single *non-blocking* attempt, and a program's control flow can depend on the result. Our most general rule for it hands each case a witness, chosen by the caller, that the case was not ready, and runs the default branch with the witnesses of all of them; a caller that can refute a witness proves the default branch unreachable, which is what verifying a double close guarded by a readiness check requires. From it we derive a simpler rule that asks only for each case to be safe, the default branch included, with no channel operation attached (§ 3.7).

Grounding the specification. Channels are part of the Go language rather than a library, so there is no user-level implementation to prove a specification against. Go's runtime implementation uses features that userspace Go programs do not have, such as parking a goroutine and writing to another's stack, so Perennial has no need to reason about them and Goose does not translate them. Another option is to give channels an axiomatic semantics. Such axioms are validated by human inspection alone, and they may become out of date as Go evolves.

We used a unique approach for this work: an executable operational model of channels (§ 2) implemented as a struct in Go itself with around 300 lines of code, translated by Perennial's existing Goose infrastructure into GooseLang—Perennial's ML-like intermediate representation translated from Go that has a formal operational semantics—with no framework extensions, and validated empirically against the Go runtime's own channel tests. This is the first precise account of Go's channel behavior that we know of: the Go language specification describes channels only informally and without enough detail to illustrate these subtleties.

Verified clients. On top of ChanLib we verified a library of resource-level specifications for common usage patterns, such as channels that transfer ownership of values, channels used to broadcast a completion signal with `close`, and pipelines. We derive them from the `select` rule, and specify their use both in `select` statements and via standalone operations. The proofs freely mix channels with Perennial's specifications for mutexes and atomics. The clients exercise the specification in different directions: `unbounded.go` uses the exclusive right to close and non-blocking sends into a full buffer, while the elimination stack uses both linearization points of the unbuffered rendezvous, since a push and a pop that eliminate each other must each partly run after the other has begun.

Contributions and outline.

- We give an executable operational model of Go channels, written in Go and tested against the Go runtime's own channel tests, and connect it to Perennial (§ 2).
- We specify the full channel API as a logical state machine, covering buffered and unbuffered channels, close, and both blocking and non-blocking `select` (§ 3). To the best of our knowledge, this is the first specification of Go's `select` in concurrent separation logic and the first logically atomic specification for a multi-channel choice construct.
- We verify clients of the specification: `unbounded.go`, an unmodified buffer from gRPC-Go, whose own clients get atomic updates over the wrapper's abstract state (§ 4)
- We verify the Mixtris theory of mixed-choice multiparty message passing, instantiated over ChanLib for the first time on an executable language and reusing its protocol layer unchanged (§ 5).
- We verify an extensive library of derived resource specifications and further verified clients in Appendix B and Appendix D.

All of our specifications and proofs are formalized in the Rocq Prover, within the Perennial framework.

Our work has some limitations. We cannot verify liveness properties, a limitation inherited from Iris. While other work has verified deadlock freedom using Iris [14, 15], integrating such reasoning into Go and combining it with other concurrency primitives is an open problem. Proofs in Perennial are carried out interactively in Rocq, though the Iris Proof Mode [17] enables reasoning at a high level of abstraction.

AI Use Disclosure. The original specifications and much of the formal development were completed without AI assistance. In adapting our work from previous conference submissions to CPP's 12-page format, we simplified the specification presented in Figure 1 and Section 3, requiring substantial updates to the mechanized proofs throughout the codebase. We used AI assistance to implement these changes and develop the newer examples in the paper, primarily to help with the substantial amount of Rocq proof code involved. The specifications and proof strategies were designed by the authors, and the resulting proofs are checked by Rocq. AI assistance was also used in artifact preparation, including writing build scripts, organizing directories, and developing documentation.

2 Operational Model of Channels

A specification is only as trustworthy as the semantics beneath it, so we want to capture Go's channel semantics faithfully. We do so with an operational model: an executable implementation of channels, written in Go itself. Writing it in Go buys two things at once. We can test the model against the Go runtime's own channel tests (§ 2.2), and we can verify reasoning principles on top of it with the same machinery our clients use (§ 2.3). We give the model's representation and its operations below, then the offer protocol that supports `select` (§ 2.1), the empirical validation (§ 2.2), and how user proofs are connected to it (§ 2.3).

The channel model uses the following Go struct:

```
type Channel[T any] struct {
  cap    int          // channel capacity
  mu     *sync.Mutex
  buffer []T          // used for buffered channels
  state  offerState   // used for unbuffered channels
  v      T
}
```

The `mu` mutex protects all mutable state so that channel operations behave atomically in the model. The basic operations—`make`, send (`ch <- v`), receive (`v, ok := <-ch`), and `close`—have straightforward implementations for buffered channels: sending appends to the buffer (blocking at capacity), receiving pops from the buffer (blocking if empty), and closing transitions to a draining state.

When a channel is unbuffered, sending blocks until a receiver is ready to accept the value, and receiving blocks until a sender provides one; the value is transferred directly between the two goroutines without any intermediate storage. The semantics of `close` are identical to the buffered case after the buffer has been drained. The `state` field, an enum of type `offerState`, is the state machine the model runs, implementing the semantics captured by the logical transition system of Figure 1. The logical and physical states serve slightly different purposes: the physical states are what a channel stores and steps through as an operation runs, while the logical states expose only the committed atomic steps a proof client reasons about. We refer the interested reader to our Rocq development for the details.

2.1 Select and the Offer Protocol

The `select` statement waits on multiple channel operations (potentially mixing sends and receives on different channels) and picks one that is ready. A prototypical `select` statement looks like the following:

```
select {
  case v := <-ch1: ...
  case ch2 <- y: ...
  default: ... // no channel was ready
}
```

We say a case is *ready* if one of the following holds: (i) a receive observes a buffered element, (ii) a receive observes a closed channel, (iii) a send finds free buffer capacity, or (iv) an unbuffered operation has a waiting partner. The presence of a `default` branch determines what happens when *no* case is ready: without `default`, the statement blocks until some case becomes ready; with `default`, the statement performs a single non-blocking attempt and, if nothing is ready, executes the default branch.[2] Throughout this section and others, we commonly refer to standalone channel operations (`ch <- v` and `v, ok := <-ch`) and cases in a blocking select as *blocking operations*, and cases in a select with `default` as *non-blocking operations*—a distinction is needed since the two types of operations have different semantics.

Buffered readiness is easy to check, but unbuffered readiness requires *advertising intent* so that a partner can complete the exchange, and a `select` cannot commit to a case before it knows which one is ready. An unbuffered exchange therefore cannot be a single step: it takes at least three, in which (1) one party advertises its intent, (2) the other party accepts it, and (3) the first party observes the acceptance, or rescinds if none came. Between (2) and (3) the advertising party still has intent outstanding elsewhere, and if a second partner could accept it there, one operation would match twice.

We navigate this challenge with an *offer protocol*: an operation advertises non-committal intent that it may rescind if unmatched, and the mutex makes accepting an offer atomic, so at most one partner can take it. Send and receive are retry loops around `TrySend` and `TryReceive`, each walking the state machine of Figure 1. A blocking send moves Idle $\rightarrow$ SndWait(v) and unlocks; on relocking it either rescinds, staying in SndWait, or, seeing RcvDone, completes and returns to Idle. Receive is symmetric. The `state` and `v` fields implement this. An unbuffered channel is a *dual data structure* with a single reservation slot, similar to Java's SynchronousQueue [23]. The steps of this protocol are the transitions our specification exposes to clients (§ 3). Observing the acceptance is a transition of its own, ⑤ for a send and ⑥ for a receive, which is why a blocking operation has two linearization points rather than one.

Non-blocking select requires further care in the model. In Go, non-blocking unbuffered operations do not synchronize with each other, yet non-blocking operations should match with blocking ones, and we want the model to reflect this behavior. We achieve this asymmetry in the model by restricting non-blocking operations to only *accept* existing offers, never create them.

Internally, select uses the primitives `TrySend` and `TryReceive`, each taking a `blocking` boolean parameter:

```
func (c *Channel[T])
TrySend(v T, blocking bool) (success bool)
func (c *Channel[T])
TryReceive(blocking bool) (success bool, v T, ok bool)
```

`success = false` indicates that the operation should be retried, while `ok = false` means the channel is closed and its buffer is empty. Offers are made only when `blocking` is `true`, and only on unbuffered channels, since buffered readiness is determined by the buffer alone. A blocking `select` calls these on each case in an arbitrary order until one succeeds; a `select` with `default` tries each case once with `blocking = false` and otherwise runs the default handler.

2.2 Empirical Validation

We want our specifications to be faithful to Go, which means the model has to agree with the runtime. We establish that by testing. We reuse the correctness-based unit tests of the

[2] Go randomizes uniformly among the set of ready cases. For our purposes, we are not concerned with probabilities—only that all observable behaviors are possible in our model.

Go runtime's own channel implementation, which exercise combinations of send, receive, close and `select` under varied buffer capacities and scheduling conditions. We regard them as a pragmatic check for divergence in semantics. The Go language specification [28] does not fully describe channels, but we tested the properties it covers, such as that sending on a closed channel panics and receiving on a `nil` channel blocks forever. We added tests of our own for non-blocking semantics, including that two non-blocking operations cannot match. One might ask why we did not verify Go's runtime implementation directly. Go's runtime channels are deeply tied to the runtime rather than being a library: the implementation can park and unpark goroutines, it references goroutines themselves in wait lists, and reads and writes directly to goroutine stacks, none of which is available to user programs and hence not supported by Goose or Perennial. Verifying the runtime implementation would require modeling all of these low-level features, and while it is an interesting direction for future work, it is orthogonal to our goal of providing usable specifications for client code.

2.3 Connecting User Proofs to the Model

Tests relate the model to Go, but a client's proof is about neither: it is about the specification of § 3, which we prove against the model. We do so in Perennial itself, the same system our clients use to verify their own Go programs. Perennial bridges the gap between Go code and formal reasoning in Rocq using a translator called Goose, which translates Go into GooseLang, a small ML-like language with references and concurrency and a formal operational semantics embedded in Rocq. We apply Goose to the channel model exactly as a user would to their own code, verifying the general specification on top of the translated operational model. The ChanLib user then sees this as the specification of the built-in channel primitive. The process is not circular: the channel model does not itself use channels.

Most of the operational model is written in Go and translated with Goose, but `select` requires special handling. We were not able to find a way to implement a heterogeneous `select`, over any number of cases on channels of different element types, in Go and verify it with Perennial. We therefore define `select` directly in GooseLang, which is untyped, as an n-ary operation over a list of cases, and this definition is what underlies all user proofs: the blocking and non-blocking specifications of § 3 are verified against it. The operational model additionally contains fixed-case `select` functions, which we built to develop the specifications and to run the Go runtime's tests against the model.

3 General, Atomic Update Specification

We want our foundational specification to be as expressive as possible, covering behavior unique to different use patterns, while remaining ergonomic for proof clients. A client enqueuing on a buffered channel should not have to pay for the rendezvous. We give ChanLib a general specification in terms of *atomic updates* [8, 26]: a logical state machine that abstracts the model, and, for each operation, the transitions it may make to that state. We introduce the reasoning principles we use (§ 3.1), specify buffered channels, unbuffered channels and closing (§ 3.2–§ 3.4), assemble the operation rules (§ 3.5), and then give the `select` rules (§ 3.6–§ 3.7).

ChanLib builds on Perennial, a concurrent separation logic mechanized in Rocq that extends Iris with support for reasoning about crashes [3], distributed systems [24], and Go code [4]. Perennial supports a standard suite of concurrent separation logic features: the points-to assertion $l \mapsto v$, Hoare triples $\{P\}\ e\ \{Q\}$ and weakest preconditions $\mathrm{wp}\ e\ \{\Phi\}$, the separating conjunction $P * Q$ and separating implication $P -\!\!* Q$, *ghost state* built from resource algebras, and *invariants* for sharing resources between threads. Since our specifications use only standard Perennial reasoning principles, they compose seamlessly with Perennial's specifications for Go's other synchronization primitives (`sync.Mutex`, atomics, `sync.WaitGroup`), which real Go code freely mixes with channels [29].

View shifts. A *view shift* $P \Rrightarrow Q$ consumes P to produce Q, similar to separating implication, but the update is not confined to resources its holder already has. It may allocate ghost state, update ghost state its holder owns, and open an invariant, use what is inside, and close it again.[3]

In a proof of $P \Rrightarrow Q$, P is provided as a hypothesis with which to derive Q, drawing on those extra steps along the way. This is how a client meets every obligation our specifications place on it. Resources shared between threads live in *invariants*: $\boxed{I}$ asserts that I holds at all times, and any thread that knows of it may open the invariant and use I, provided it restores I before it is done. Opening an invariant is one of the steps a view shift permits: a client that has to produce a resource it does not hold opens an invariant containing that resource, takes it, and closes the invariant before the shift ends. What it may not do is take a step of the program in between: Iris allows an invariant to be opened across a view shift but not across a symbolic program step, so no other thread can observe I broken.

As a hypothesis, $P \Rrightarrow Q$ may be used to exchange P for Q, at any point in a program proof. It is not duplicable, so the exchange can happen only once. The two-phase specifications below leave a caller holding exactly such a one-shot right between an operation's two linearization points. We consistently use blue for what is consumed and green for what is provided.

[3] Iris controls that invariants are only opened safely via technical machinery known as masks, laters, and later credits. For brevity sake, we omit further details on these. They are low-level details of the Iris encoding and are not needed to understand the specifications. We refer the interested reader to our Rocq development, which has them in full.

Fractional ghost state. A channel's logical state is a *fractional* ghost variable. We write $\mathsf{OwnChan}^{\gamma}_{q}(\mathsf{ch}, s)$ for a share $q \in (0, 1]$ of the assertion that channel ch is in state s, and shares split and recombine:

$$\mathsf{OwnChan}^{\gamma}_{q_1+q_2}(\mathsf{ch}, s) \dashv\vdash \mathsf{OwnChan}^{\gamma}_{q_1}(\mathsf{ch}, s) * \mathsf{OwnChan}^{\gamma}_{q_2}(\mathsf{ch}, s)$$

Two rules govern them. *Agreement*: any two shares must carry the same state, and shares totalling more than 1 are contradictory,

$$\mathsf{OwnChan}^{\gamma}_{q_1}(\mathsf{ch}, s_1) * \mathsf{OwnChan}^{\gamma}_{q_2}(\mathsf{ch}, s_2) \vdash s_1 = s_2 \ \wedge \ q_1 + q_2 \le 1$$

Update: only the whole variable may change state, and doing so is a *basic* update $\dot{\Rightarrow}$, which allocates and changes ghost state but opens no invariants,

$$\mathsf{OwnChan}^{\gamma}_{1}(\mathsf{ch}, s) \vdash \dot{\Rightarrow} \mathsf{OwnChan}^{\gamma}_{1}(\mathsf{ch}, s')$$

A basic update may be performed inside a view shift, so this rule is available wherever a client is justifying a transition. Exclusive ownership of the state is thus $q = 1$, and we split it between the specification's own invariant and the client, half each. Other splittings are possible, but $1/2$ is the one we use throughout. A half is a right to *observe* the state that no other thread can invalidate, and the two halves together are the right to *change* it, which their holders can only exercise at the same instant.

Specifying state transitions. We specify each channel operation by the transitions it can make to the channel's *logical state*: an abstract state that the specification maintains alongside the physical channel, and that a client reasons about instead of the model. This is a common convention [16, 26]. What is specific to channels is which transitions we have in the logical state.

The logical state is a fractional ghost variable held in halves: the specification's own invariant keeps one, and the client keeps the other, a view of the state that is always accurate and that the client cannot act on alone. To take a logical step, the operation hands its half over at the pre-state and asks for the post-state back, which we write as the view shift

$$\mathsf{OwnChan}^{\gamma}_{1/2}(\mathsf{ch}, s) \Rrightarrow \mathsf{OwnChan}^{\gamma}_{1/2}(\mathsf{ch}, s') * \Phi(r)$$

Holding both halves, the client has exclusive ownership at the transition, so the update rule applies and the state may change. It keeps one half of the new state, returns the other, and supplies a continuation Φ applied to the operation's return value r. Both s' and r may depend on data the shift reads out of the state it finds.

An operation can make different transitions depending on the state it finds, so its specification is the *overlapping conjunction* (also called the intuitionistic conjunction) $\wedge$ of one shift per possible transition, which we call the operation's *atomic update* (AU). The conjunction expresses *external choice*: $A \wedge B$ gives its owner the right to use A or B, but not both, since both are backed by the *same* resources, unlike $A * B$. The channel's state, not the caller, determines which transition fires, so the caller must be prepared to prove every conjunct from the same resources. Dually, this is easier than proving a separating conjunction: $P \vdash A$ and $P \vdash B$ give $P \vdash A \wedge B$, typically by opening one invariant that can justify any of the transitions.

An operation's specification hands it the AU for a caller-chosen Φ and promises Φ on return. Since Φ is universally quantified, the proof of the operation has no way to produce it other than by exercising one of the caller's view shifts, and since a view shift is not duplicable it can do so only once. Exactly one transition fires, at the linearization point. Φ is also how the proof composes. An Iris proof proceeds by symbolic execution: it carries the resources the thread owns, applies a specification to the next statement, and is left with what that statement produced and the remainder of the program to verify. So for Φ a caller picks the obligation to verify everything after the operation, $\lambda y.\ \mathsf{wp}\ \mathit{rest}(y)\ \{\Psi\}$. It then makes the exchange the AU dictates, giving back the post-state and moving whatever resources the transition calls for between itself and the channel, and goes on to verify *rest* with no channel reasoning left in it.

3.1 Channel Specifications

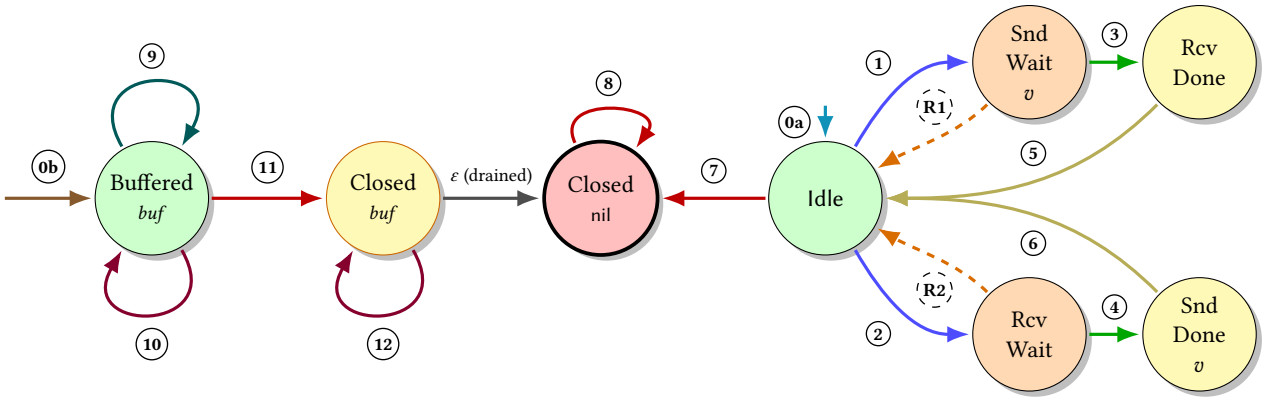
The channel specification describes each operation in terms of transitions of a logical state machine. Transitions have preconditions and postconditions. For example, close is illegal on a closed channel, and the specification guarantees that certain states are never reached.

The logical state of a channel is the following inductive type, indexed by the type V of values sent over the channel:

$$\begin{aligned}\mathsf{Chan}\ V \ ::= \ & \mathsf{Buffered}(\mathit{buf} : \mathsf{list}\, V) \mid \mathsf{Idle} \mid \mathsf{SndWait}(v : V) \\ & \mid \mathsf{RcvWait} \mid \mathsf{SndDone}(v : V) \mid \mathsf{RcvDone} \\ & \mid \mathsf{Closed}(\mathit{drain} : \mathsf{list}\, V)\end{aligned}$$

These constructors are the states of Figure 1, and we introduce each as we meet it. A send that offers and rescinds repeatedly leaves the logical state at Idle throughout, reaching $\mathsf{SndWait}(v)$ only at the instant its offer is guaranteed to be matched, so a client never reasons about retries or transient failed offers. $\mathsf{IsChan}_{\gamma}(\mathsf{ch})$ is a *persistent* (freely duplicable) proposition tying a ghost-state bundle γ (consisting of the channel's ghost name and its fixed capacity $\gamma.\mathsf{cap}$) to the channel location ch; it holds the specification's internal invariants, which clients never see. $\mathsf{OwnChan}_{\gamma}(\mathsf{ch}, s)$ is the client's half of the logical state, currently s; the other half lives inside $\mathsf{IsChan}_{\gamma}(\mathsf{ch})$. We write the fraction explicitly, and for space reasons use the following sugar throughout:

$$[\![s]\!]^{\gamma}_{1/2} \triangleq \mathsf{OwnChan}^{\gamma}_{1/2}(\mathsf{ch}, s)$$



$\mathrm{SendAU}_\gamma(\mathsf{ch}, v, \Phi) \triangleq \mathrm{SendFastPathAU} \wedge \mathrm{SendSlowPathAU} \wedge \mathrm{SendEnqAU} \wedge \mathrm{SendClosedAU}$

$\mathrm{SendFastPathAU} \triangleq \underbrace{[\![\mathrm{RcvWait}]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{SndDone}(v)]\!]^\gamma_{1/2}}_{④} * \Phi$ $\quad$ $\mathrm{SendClosedAU} \triangleq [\![\mathrm{Closed}(_)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* \mathrm{False}$

$\mathrm{SendEnqAU} \triangleq \forall buf.\ \underbrace{|buf| < \gamma.\mathrm{cap} * [\![\mathrm{Buffered}(buf)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Buffered}(buf +\!\!+ [v])]\!]^\gamma_{1/2}}_{⑨} * \Phi$

$\mathrm{SendSlowPathAU} \triangleq \underbrace{[\![\mathrm{Idle}]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{SndWait}(v)]\!]^\gamma_{1/2}}_{①} * (\underbrace{[\![\mathrm{RcvDone}]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Idle}]\!]^\gamma_{1/2}}_{⑤} * \Phi)$

$\mathrm{RecvAU}_\gamma(\mathsf{ch}, \Phi) \triangleq \mathrm{RecvFastPathAU} \wedge \mathrm{RecvSlowPathAU} \wedge \mathrm{RecvDeqAU} \wedge \mathrm{RecvDrainAU} \wedge \mathrm{RecvClosedAU}$

$\mathrm{RecvFastPathAU} \triangleq \forall v.$
$\underbrace{[\![\mathrm{SndWait}(v)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{RcvDone}]\!]^\gamma_{1/2}}_{③} * \Phi(v, \mathtt{true})$

$\mathrm{RecvClosedAU} \triangleq$
$\underbrace{[\![\mathrm{Closed}(\mathrm{nil})]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Closed}(\mathrm{nil})]\!]^\gamma_{1/2}}_{⑧} * \Phi(\mathrm{zero}\ V, \mathtt{false})$

$\mathrm{RecvDeqAU} \triangleq \forall v, vs.$
$\underbrace{[\![\mathrm{Buffered}(v :: vs)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Buffered}(vs)]\!]^\gamma_{1/2}}_{⑩} * \Phi(v, \mathtt{true})$

$\mathrm{RecvDrainAU} \triangleq \forall v, vs.$
$\underbrace{[\![\mathrm{Closed}(v :: vs)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Closed}(vs)]\!]^\gamma_{1/2}}_{⑫} * \Phi(v, \mathtt{true})$

$\mathrm{RecvSlowPathAU} \triangleq \underbrace{[\![\mathrm{Idle}]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{RcvWait}]\!]^\gamma_{1/2}}_{②} * (\forall v.\ \underbrace{[\![\mathrm{SndDone}(v)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Idle}]\!]^\gamma_{1/2}}_{⑥} * \Phi(v, \mathtt{true}))$

$\mathrm{CloseAU}_\gamma(\mathsf{ch}, \Phi) \triangleq \mathrm{CloseIdleAU} \wedge \mathrm{CloseBufAU} \wedge \mathrm{CloseClosedAU}$

$\mathrm{CloseIdleAU} \triangleq \underbrace{[\![\mathrm{Idle}]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Closed}(\mathrm{nil})]\!]^\gamma_{1/2}}_{⑦} * \Phi$ $\quad$ $\mathrm{CloseClosedAU} \triangleq [\![\mathrm{Closed}(_)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* \mathrm{False}$

$\mathrm{CloseBufAU} \triangleq \forall buf.\ \underbrace{[\![\mathrm{Buffered}(buf)]\!]^\gamma_{1/2} \Rrightarrow\!\!\!* [\![\mathrm{Closed}(buf)]\!]^\gamma_{1/2}}_{⑪} * \Phi$

Figure 1. The channel state machine, and the atomic updates realizing its transitions. R1 and R2 are internal to the implementation (§ 2). For readability, the individual conjuncts (such as SendFastPathAU) elide their parameters; each is implicitly parameterized by the γ, ch, v, and Φ of its enclosing operation AU. The Φ's are applied to what the operation returns, so a bare Φ is not an elision: send and close return nothing, while receive returns the value and whether it received one.

OwnChan also carries the pure fact that s is consistent with the capacity, which is what makes the bound below available to clients from state ownership alone. Figure 1 shows the transitions between these states. Each transition is *realized* by a named atomic update that encodes it as a view shift on $[\![s]\!]^\gamma_{1/2}$. When an operation can fire different transitions depending on the channel's state, we define the full

operation's AU as their conjunction, as Figure 1 does for SendAU, RecvAU and CloseAU. The buffered transitions are enabled only when $\gamma.\mathrm{cap} > 0$ and the unbuffered ones only when $\gamma.\mathrm{cap} = 0$, so a channel makes one set of transitions or the other, never both. Concretely, OwnChan enforces that Buffered(_) exists only when $\gamma.\mathrm{cap} > 0$, whereas Idle, SndWait(_), RcvWait, SndDone(_), and RcvDone exist only when $\gamma.\mathrm{cap} = 0$ (determined at creation in Figure 2), with Closed(nil) reachable from both and Closed($v :: vs$) only when $\gamma.\mathrm{cap} > 0$. The two sets of transitions are therefore separated by their state constructors—no explicit $\gamma.\mathrm{cap} = 0$ guard is needed on the unbuffered rules, and the $|buf| < \gamma.\mathrm{cap}$ premise in SendEnqAU solely guards against sending into a full buffer.

Conjuncts and their labels. Each named AU below is one conjunct of an operation's specification, and we write it with the transition it realizes named under the brace:

$$\underbrace{[\![s]\!]^{\gamma}_{1/2} \Rrightarrow\!\!\ast\ [\![s']\!]^{\gamma}_{1/2}}_{\text{(i)}} * \Phi(r)$$

The number is the edge (i) of Figure 1, so a conjunct can be checked against the transition it stands for and an operation against the portion of the diagram its conjuncts cover. We introduce each conjunct alongside its edge below.

The half the conjunct receives is the channel's. The client supplies its own, normally out of an invariant it opens for the occasion, and agreement between the two either confirms that the state matches s or hands the client a contradiction. So the conjuncts name a state outright rather than guarding on one, and an invariant that rules out s discharges the conjunct without the client proving anything. A conjunct of this shape is *single-phase*: one linearization point, one transition.

Two-phase transitions. Blocking operations on unbuffered channels can execute with multiple linearization points, which we represent with a *2-phase* AU: the client must prove two view shifts, and the second is the one that provides Φ. Such a conjunct covers transitions (i) and (j):

$$\underbrace{[\![s]\!]^{\gamma}_{1/2} \Rrightarrow\!\!\ast\ [\![s']\!]^{\gamma}_{1/2}}_{\text{(i)}} * \Big(\underbrace{[\![s'']\!]^{\gamma}_{1/2} \Rrightarrow\!\!\ast\ [\![s''']\!]^{\gamma}_{1/2}}_{\text{(j)}} * \Phi(r)\Big)$$

The return value r and next state s''' may depend on s'': for instance, when a receiver completes a slow-path receive, $s'' = \mathsf{SndDone}(v)$ and the return value is (v, true), with v extracted from the state itself. The continuation Φ is withheld until the second phase fires: the caller holds the residual shift as a resource between the two linearization points and can only obtain Φ once the channel has reached s'''.

Figure 1 depicts the state machine along with labelled atomic update definitions for the transitions of interest. We walk it by kind of channel rather than by operation, since the conjuncts of one operation live in different parts of the diagram, and the conjuncts that belong together across operations do not.

3.2 Buffered Channels

A buffered channel is a bounded queue, and its two states are Buffered(buf) and, once closed, Closed($drain$). A send with room in the buffer fires SendEnqAU, appending and returning without waiting for anyone. A send on a full buffer blocks instead, so the transition is guarded: it is enabled only when $|buf| < \gamma.\mathrm{cap}$, and the caller needs that guard to produce the extended state. The specification exposes the bound in the other direction as well, so a client can always recover it from state ownership alone:

$$[\![\mathsf{Buffered}(buf)]\!]^{\gamma}_{1/2} \vdash |buf| \le \gamma.\mathrm{cap}$$

A receive from a non-empty buffer fires RecvDeqAU and pops the head.

Both are single-phase: one linearization point, one transition, and the Buffered self-loops (9) and (10) are the only edges either one takes. A buffered channel is therefore specified much as a bounded queue would be, one linearization point per operation, with closing as the one addition.

3.3 Unbuffered Channels

An unbuffered channel has no capacity to store a value, so a send and a receive must meet. Neither neither has a single linearization point: each completes only after the other has begun. We navigate this by offering the client a choice between two sequences of transitions, only one of which it takes, with the channel's state deciding which. The five remaining states record how far an exchange has got: Idle when nobody is waiting, SndWait(v) and RcvWait when one party has announced itself, and SndDone(v) and RcvDone when the other has taken it.

When a partner is already waiting, the fast paths take over. SendFastPathAU finds the channel in RcvWait, hands its value directly to the waiting receiver and completes in one step, leaving SndDone(v) behind for that receiver to collect with (6). RecvFastPathAU is the mirror image: it finds SndWait(v), takes the value and leaves RcvDone. The value is not chosen by the receiver but extracted from the state it finds, which is why that conjunct quantifies over v. Both are single-phase.

The slow paths apply when no partner has arrived. SendSlowPathAU steps from Idle to SndWait(v), announcing the send, and yields in place of Φ a residual view shift enabled only once a receiver has taken the value and moved the channel to RcvDone. RecvSlowPathAU mirrors it, with the value delivered in the second phase. The caller holds the residual shift as a resource between its two linearization points and cannot obtain Φ until the channel reaches the state that phase names. These are the only 2-phase conjuncts in the specification.

Those two phases are what let a matched pair overlap in time. Consider a send and a receive that match, with the send arriving first. The sender parks, leaving its first phase with the implementation and keeping only the residual shift ⑤. The receiver runs both: ① moves the channel to SndWait(v) and RecvFastPathAU's ③ takes it to RcvDone, back to back in one atomic step, which is what *helping* means here. The sender then fires ⑤, returning the channel to Idle and releasing Φ_{send}, while the receiver already holds $\Phi_{recv}(v, \texttt{true})$. The logical offer states are never held across a program step: they exist only inside one atomic step, which is why the logical machine needs no edge for a rescinded offer.

The sequence is ①③⑤; the mirror-image sequence when the receiver arrives first is ②④⑥. On either sequence, each operation's transitions straddle one of the other's: receive fires after send begins, and send finishes after receive begins.

An unbuffered channel is not linearizable. It is instead an *exchanger*; a synchronization point where two threads pair up and swap, studied by Castañeda et al. [2] and Dang et al. [5]. The classic client of an exchanger is an elimination stack, and we verify one over an unbuffered Go channel (§ D.1).

3.4 Closing

Closing imposes two opposing requirements. Closing twice panics, so the right to close must be exclusive; a closed channel must remain receivable by any number of threads, so what close produces must be shareable without limit. We meet both with conjuncts of two shapes, one that can never fire and one that never moves the state. Close itself succeeds in two ways, one per kind of channel: CloseIdleAU takes an idle unbuffered channel to Closed(nil), and CloseBufAU closes a buffered one, keeping whatever is still in its buffer.

A receive from a closed channel behaves like an open one until the buffer is empty: RecvDrainAU is a dequeue and still returns ok = `true`. Once the buffer is empty receives become idempotent, each taking RecvClosedAU's self-loop ⑧ and returning the zero value with ok = `false`. Any number of receivers can fire that self-loop, making close a broadcast.

The two illegal operations are the conjuncts ending in False. Sending on a closed channel panics, and closing twice panics, so SendClosedAU and CloseClosedAU are obligations the caller must show unreachable. This is what makes the right to close exclusive in every specification built on top: a client must own enough of the channel's state to know that nobody else has closed it and that no send can follow. Because a caller must *prove* every conjunct of an AU rather than assume it, discharging CloseClosedAU (or SendClosedAU) requires deriving False from the hypothesis $[\![\mathsf{Closed}(_)]\!]^{\gamma}_{1/2}$ by showing it contradicts the caller's own half of the state. In `close(ch); close(ch)`, the first `close` leaves the caller holding a Closed half (e.g., $[\![\mathsf{Closed}(\mathsf{nil})]\!]^{\gamma}_{1/2}$), which agrees with Closed(_) and yields no contradiction, so the second `close` cannot be verified.

3.5 The Operation Specifications

Each operation's atomic update is the conjunction of its transitions, shown at the head of each group in Figure 1. As described in § 3.1, the conjunction is the intuitionistic ∧: the channel's state, not the caller, picks which conjunct fires, so the caller must be prepared to prove all of them from the same resources.

Figure 2 gives the specification of each operation as a predicate transformer: given the persistent $\mathsf{IsChan}_{\gamma}(\mathsf{ch})$ and the operation's atomic update for a caller-chosen Φ, the operation runs safely and establishes Φ.

Creation is the one operation with no atomic update, since it produces the channel's state rather than transforming it. Its two rules are where $\mathsf{IsChan}_{\gamma}(\mathsf{ch})$ and the initial $[\![\,]\!]^{\gamma}_{1/2}$ come from, and the capacity argument decides which of the two initial states the channel starts in: a capacity of zero gives an unbuffered channel, in Idle; any other capacity gives a buffered one, with an empty buffer. These are the two remaining transitions of Figure 1, and γ.cap is fixed here for the lifetime of the channel.

Chan-Send, Chan-Receive, and Chan-Close are the whole interface for standalone channel operations, and the degenerate case of the `select` rule: each is equivalent to our `select` specification with a single case and no case expression, which has identical semantics and which we describe next.

3.6 Blocking Select

Go's `select` waits on several channel operations and runs exactly one that is ready, chosen nondeterministically. Figure 3 gives its syntax, writing each case's body as a call to a handler h_i so that the specification can talk about the body without committing to what it does.

Every case has to be safe to execute even though only one of them will run, so the same resources have to justify all of them. Requiring the separating conjunction of the cases' preconditions would be sound and unusable: the cases typically need the *same* resources, precisely because only one runs, and a caller would have to own a private copy per case. The specification is instead the same external choice as a single operation's, one level up. Where a single operation's AU conjoins the transitions of one channel, Chan-Select conjoins the AUs of its cases, written $\mathsf{AU}_i(\Phi)$. The continuation of case i's AU is the weakest precondition of that case's handler, so each case fires one transition and then runs one body, and the whole statement has a single postcondition Φ.

For example, a two-case `select`

```
select {
case ch0 <- v:        h0()
case v, ok := <-ch1:  h1(v, ok)
}
```

unfolds to two obligations,

$$\mathsf{SendAU}_{\gamma_0}(\mathsf{ch}_0, v, \mathsf{wp}\ \mathtt{h0()}\ \{\Phi\})$$
$$\wedge\ \mathsf{RecvAU}_{\gamma_1}(\mathsf{ch}_1, \lambda v, \mathsf{ok}.\ \mathsf{wp}\ \mathtt{h1(v,ok)}\ \{\Phi\})$$

Chan-Make-Unbuffered
$$\forall \Phi.\ \left(\forall \mathsf{ch}, \gamma.\ \mathsf{IsChan}_\gamma(\mathsf{ch}) * \gamma.\mathsf{cap} = 0 * [\![\mathsf{Idle}]\!]^\gamma_{1/2} \mathrel{-\!\!*} \Phi(\mathsf{ch})\right) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{make(chan T, 0)}\ \{\Phi\}$$

Chan-Make-Buffered
$$0 < \mathit{cap} \vdash \forall \Phi.\ \left(\forall \mathsf{ch}, \gamma.\ \mathsf{IsChan}_\gamma(\mathsf{ch}) * \gamma.\mathsf{cap} = \mathit{cap} * [\![\mathsf{Buffered(nil)}]\!]^\gamma_{1/2} \mathrel{-\!\!*} \Phi(\mathsf{ch})\right) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{make(chan T, } \mathit{cap}\texttt{)}\ \{\Phi\}$$

Chan-Send
$$\mathsf{IsChan}_\gamma(\mathsf{ch}) \vdash \forall \Phi.\ \mathsf{SendAU}_\gamma(\mathsf{ch}, v, \Phi) \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{ch}\ \texttt{<-}\ v\ \{\Phi\}$$

Chan-Receive
$$\mathsf{IsChan}_\gamma(\mathsf{ch}) \vdash \forall \Phi.\ \mathsf{RecvAU}_\gamma(\mathsf{ch}, \Phi) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{<-}\ \mathsf{ch}\ \{v, \mathsf{ok}.\ \Phi(v, \mathsf{ok})\}$$

Chan-Close
$$\mathsf{IsChan}_\gamma(\mathsf{ch}) \vdash \forall \Phi.\ \mathsf{CloseAU}_\gamma(\mathsf{ch}, \Phi) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{close(ch)}\ \{\Phi\}$$

Figure 2. Specifications for the standalone operations.

```
select {
  case op_0:  h_0(...)
    ⋮
  case op_n:  h_n(...)
}
```
abbreviated `select...`

```
select {
  case op_0:  h_0(...)
    ⋮
  case op_n:  h_n(...)
  default:    h_dflt()
}
```
abbreviated `select...default...`

where op_i is ch_i `<-` v_i for a send case and v, ok `:= <-` ch_i for a receive case, h_i is the body of case i, taking no argument for a send and the received v and ok for a receive, h_{dflt} is the body of the `default` branch, and γ_i is the ghost name of ch_i.

$$\mathsf{AU}_i(\Phi) \triangleq \begin{cases} \mathsf{SendAU}_{\gamma_i}(\mathsf{ch}_i, v_i, \mathsf{wp}\ h_i()\ \{\Phi\}) & \text{if } \mathsf{op}_i = \mathsf{ch}_i \texttt{ <- } v_i \\ \mathsf{RecvAU}_{\gamma_i}(\mathsf{ch}_i, \lambda v, \mathsf{ok}.\ \mathsf{wp}\ h_i(v, \mathsf{ok})\ \{\Phi\}) & \text{if } \mathsf{op}_i = \texttt{<- } \mathsf{ch}_i \end{cases}$$

$$\mathsf{NBAU}_i(\Phi, \Phi_{\mathsf{notready}}) \triangleq \begin{cases} \mathsf{NonBlockingSendAU}_{\gamma_i}(\mathsf{ch}_i, v_i, \mathsf{wp}\ h_i()\ \{\Phi\}, \Phi_{\mathsf{notready}}) & \text{if } \mathsf{op}_i = \mathsf{ch}_i \texttt{ <- } v_i \\ \mathsf{NonBlockingRecvAU}_{\gamma_i}(\mathsf{ch}_i, \lambda v, \mathsf{ok}.\ \mathsf{wp}\ h_i(v, \mathsf{ok})\ \{\Phi\}, \Phi_{\mathsf{notready}}) & \text{if } \mathsf{op}_i = \texttt{<- } \mathsf{ch}_i \end{cases}$$

NB-Send-AU
$$\mathsf{NonBlockingSendAU}_\gamma(\mathsf{ch}, v, \Phi, \Phi_{\mathsf{notready}}) \triangleq \mathsf{SendFastPathAU} \wedge \mathsf{SendEnqAU} \wedge \mathsf{SendClosedAU} \wedge \Phi_{\mathsf{notready}}$$

NB-Recv-AU
$$\mathsf{NonBlockingRecvAU}_\gamma(\mathsf{ch}, \Phi, \Phi_{\mathsf{notready}}) \triangleq \mathsf{RecvFastPathAU} \wedge \mathsf{RecvDeqAU} \wedge \mathsf{RecvDrainAU} \wedge \mathsf{RecvClosedAU} \wedge \Phi_{\mathsf{notready}}$$

Chan-Select
$$\mathop{\Large *}_{i=1}^{n} \mathsf{IsChan}_{\gamma_i}(\mathsf{ch}_i) \vdash \forall \Phi.\ \bigwedge_{i=1}^{n} \mathsf{AU}_i(\Phi) \mathrel{-\!\!*} \mathsf{wp}\ \texttt{select...}\ \{\Phi\}$$

Chan-Select-Default
$$\mathop{\Large *}_{i=1}^{n} \mathsf{IsChan}_{\gamma_i}(\mathsf{ch}_i) \vdash \forall \Phi.\ \left(\bigwedge_{i=1}^{n} \mathsf{NBAU}_i(\Phi, \mathsf{True})\right) \wedge \mathsf{wp}\ h_{\mathrm{dflt}}()\ \{\Phi\} \mathrel{-\!\!*} \mathsf{wp}\ \texttt{select...default...}\ \{\Phi\}$$

Figure 3. `select` and its rules. Chan-Select-Default is derived from the stronger Chan-Select-NotReady of Appendix A.

proved from the same resources. One is equivalent to the goal for `ch0 <- v; h0()` and the other to the goal for `v, ok := <-ch1; h1(v, ok)`, so proving a `select` requires the same reasoning per case as the corresponding standalone channel operation.

Because each premise is only an atomic update, any specification that can produce such an update can supply a case: § 3.8 and our case studies multiplex channels with entirely different specifications in a single `select`, with no `select`-specific reasoning.

3.7 Non-Blocking Select

With a `default` branch, `select` makes a single non-blocking attempt at its cases and runs the default branch if none of them is ready. Its syntax, in Figure 3, gains one more handler.

Non-blocking operations have deliberately different semantics: a concurrent non-blocking send and receive on an unbuffered channel do *not* synchronize, since neither makes an offer.[4] In the model, non-blocking operations only accept existing offers, never create them, and the specification captures this by removing the conjunct that announces one: NonBlockingSendAU is SendAU *without* SendSlowPathAU, and NonBlockingRecvAU is RecvAU without RecvSlowPathAU (NB-Send-AU and NB-Recv-AU).

The weakenings

$$\mathsf{SendAU}_\gamma(\mathsf{ch}, v, \Phi) \vdash \mathsf{NonBlockingSendAU}_\gamma(\mathsf{ch}, v, \Phi, \mathsf{True})$$
$$\mathsf{RecvAU}_\gamma(\mathsf{ch}, \Phi) \vdash \mathsf{NonBlockingRecvAU}_\gamma(\mathsf{ch}, \Phi, \mathsf{True})$$

therefore hold by dropping a conjunct, so programs that do not rely on non-blocking semantics can use their blocking AUs unchanged.

Both non-blocking AUs carry one further conjunct, Φ_{notready}, which is what the caller obtains when no transition is enabled. How much a client wants from that conjunct varies, so we give two rules for non-blocking `select`, one general and one derived from it.

The primitive rule is Chan-Select-NotReady, stated in Appendix A. It lets the caller choose Φ_{notready} per case, and requires that it be proved rather than handing it over: a case that declines to fire must show that the channel was in a state where none of its conjuncts is enabled, and the default branch runs holding what every case handed back. A client that can refute one of those witnesses proves the default branch unreachable.

Chan-Select-Default is derived from it, and is the rule we use unless we say otherwise. Its premises are the non-blocking AUs with $\Phi_{\mathsf{notready}} = \mathsf{True}$, so nothing has to be earned and nothing is learned: the rule is Chan-Select with those AUs and the default handler as one more conjunct of the same $\wedge$, proved from the same resources as every case and run when no case fires. Appendix A states the general rule and works an example of each: a `select` whose communication case can never fire, proved with the derived rule, and one whose case always fires, whose default branch contains a second `close` and is proved unreachable with the general one.

3.8 Building Resource Specifications on Top

The atomic update specification is powerful but heavyweight, and programs that use a channel in a disciplined way often benefit from reasoning only about the limited pattern of resource transfer their own use of it involves, with the state machine hidden. Such specifications are derived, not axiomatized: define an invariant tying the channel's logical state to the resources in transit, prove the operation's AU from it, and package Hoare triples. Our simplest example is the *bag* channel, derived in full in § B.1; the interested reader will find several more in Appendix B. Each derivation also yields AU lemmas, which is what a `select` case requires, so a bag channel's case and a broadcast channel's case are discharged by their own lemmas and Chan-Select does not distinguish between them.

4 Unbounded: A Channel as a Queue, Its Close as a Signal

`unbounded.go` is an unbounded FIFO buffer from gRPC-Go's internals, used to pass updates between components without ever blocking a producer.[5] It wraps a channel to give it slightly different semantics, using the following struct:

```
type Unbounded struct {
  c       chan any // cap 1
  closed  bool
  closing bool
  mu      sync.Mutex
  backlog []any
}
```

The mutex-guarded `backlog` absorbs values the capacity-1 channel `c` has no room for, so a producer never blocks, and the pair of flags makes `Close` idempotent, so a second close does not panic. The channel is the delivery end: `Put` appends to `backlog`, first attempting a non-blocking send into `c` when the backlog is empty; consumers receive from the channel returned by `Get` and call `Load` after each read to move the next backlog element in; `Close` stops new `Puts` and arranges for `c` to be closed once all data has drained, by `Close` itself if the backlog is already empty and otherwise by the final `Load`.

We give `Unbounded` three atomic updates of its own, for `Put`, for receiving off `Get()`, and for `Close`, over an abstract state that is a list with an open/closed flag, so that its clients reason about the buffer and never about the channel inside it. Appendix C states them. They differ from the channel's own updates exactly where the wrapper's semantics do: `Put` never blocks, and `Close` is idempotent where CloseAU has `False` for a second close. All three are proved from the general specification and Perennial's mutex specification and nothing else. The proof state is split in two. An invariant holds the ghost state: the channel's $[\![s]\!]^\gamma_{1/2}$, the buffer's own half, and mirrors of `backlog`, `closing` and `closed`. The mutex's lock invariant holds the struct fields themselves and the other half of each mirror. The split is forced by `Get`, which hands a client the bare channel: a thread receiving off it holds no mutex and cannot take one inside a view shift, so the channel's logical state has to be reachable without one.

[4] This property is not documented in the Go specification; we verified it empirically with stress tests.

[5] https://github.com/grpc/grpc-go/blob/83bead40c01c/internal/buffer/unbounded.go

Each of the buffer's arms fixes its own post-state, so firing the client's view shift *is* the linearization point, with no ghost update of the buffer's state on either side of it. The False conjunct of CloseAU is discharged at each of the two call sites that close `b.c`, since a site that has just tested `closed` under the mutex knows the channel is not Closed, and both sites send with a `select` and a `default`, verified with the non-blocking specification of § 3.7.

5 Mixtris over Go Channels

Mixtris [13] is a recent separation logic theory of *mixed-choice* multiparty session types: n participants, each of whose protocol may offer sends and receives simultaneously, with only one of the offered exchanges taking place and both parties agreeing on which. Verifying a program means showing its pool of protocols *consistent*: for every exchange the protocols permit, what the receiver expects is what the sender supplies. Its formalization targets HeapLang, which has neither a synchronous channel nor a choice construct, so the development builds a synchronization primitive of its own to supply both. We instantiated the Mixtris protocol theory over ChanLib. Go supplies both constructs in the language: an unbuffered channel is a synchronization cell, and a blocking `select` is a mixed choice performed by the runtime. The protocol layer of the Mixtris development is reused unchanged: protocol definitions, the subprotocol relation $p \sqsubseteq p'$, and consistency proofs carry over verbatim, because a protocol action is annotated with the *participant* it communicates with exactly as in the original. Only the channel layer is re-founded. A participant i owns its protocol, written $i \rightarrowtail p$, where protocols are built from participant-annotated sends $!\,[j]\,\langle v\rangle\{P\}.\,p$ (send v and resources P to participant j, then continue as p), receives $?[i]\,\langle v\rangle\{P\}.\,p$ (receive v and P from i), and mixed choices $p_1 + p_2$ (commit to either branch). The subprotocol relation $p \sqsubseteq p'$ permits weakening ownership via $i \rightarrowtail p \mathrel{-\!\!*} i \rightarrowtail p'$ (e.g., projecting a branch $p_1 + p_2 \sqsubseteq p_k$). A Go channel is assigned to a directed edge of the session by a persistent assertion $\mathsf{IsEdge}(\mathsf{ch}, i \to j)$. The interface to the rest of the development is two lemmas, which produce from protocol ownership exactly the atomic updates that Chan-Select demands:

Mixtris-Send

$$\mathsf{IsEdge}(\mathsf{ch}, i \to j) \mathrel{-\!\!*} i \rightarrowtail !\,[j]\,\langle v\rangle\{P\}.\,p \mathrel{-\!\!*} P \mathrel{-\!\!*} \big(i \rightarrowtail p \mathrel{-\!\!*} \Phi\big) \mathrel{-\!\!*} \mathsf{SendAU}_\gamma(\mathsf{ch}, v, \Phi)$$

Mixtris-Recv

$$\mathsf{IsEdge}(\mathsf{ch}, i \to j) \mathrel{-\!\!*} j \rightarrowtail ?[i]\,\langle v\rangle\{P\}.\,p \mathrel{-\!\!*} \big(\forall v.\ P \mathrel{-\!\!*} j \rightarrowtail p \mathrel{-\!\!*} \Phi(v, \mathtt{true})\big) \mathrel{-\!\!*} \mathsf{RecvAU}_\gamma(\mathsf{ch}, \Phi)$$

A send from i toward j consumes the payload P and yields the continuation protocol at the linearization point; the matching receive at j acquires P along with the value. A blocking `select` whose cases are discharged by these lemmas *is* a mixed choice, performed by the Go runtime in one step. Given a mixed-choice protocol $p_1 + p_2$, the caller proves the send case from $i \rightarrowtail p_1 + p_2 \mathrel{-\!\!*} i \rightarrowtail p_1$ and the receive case from $\ldots \mathrel{-\!\!*} i \rightarrowtail p_2$: both cases from the *same* protocol ownership, narrowed to each branch by subprotocol weakening, conjoined by $\wedge$, with the winning case consuming it. Go's `select` implements the mixed choice of Mixtris's protocols precisely.

Both lemmas are proved from a per-channel invariant whose offer states hold the parked party's protocol ownership, so the party arriving second takes the protocol step for both of them while it has the invariant open. This is the same helping as in § D.1, and it is available for the same reason: the offer and the commit are separate transitions of Figure 1.

Channels and edges. The instantiation departs from Mixtris's implementation in one way that Go's semantics invites. Rather than allocating a channel for every ordered pair of participants, the client allocates the channels the protocol actually needs and assigns each to a directed edge; a ring of n participants needs n channels. Channels and edges must correspond exactly, which we get by keying the client's channel map on the edge. Setting the session up trades that whole map at once: the client gives up the $[\![\mathsf{Idle}]\!]^{\gamma}_{1/2}$ of every channel it has just allocated and gets back a persistent $\mathsf{IsEdge}(\mathsf{ch}, i \to j)$ for each, which each operation then looks up for the single edge it uses. A channel serving two senders would let a receiver be handed a value from a participant its protocol offers no receive branch for, and an edge with two channels would let a send on one never meet a receive on the other.

We verified the Mixtris paper's running example, three-way leader election, over Go channels, and give the proof in § D.2. The leader's privilege is a heap location that the leader frees in the original development. Over Go channels it becomes the exclusive permission to `close` the channel, which can be exercised only once (§ 3.4).

6 Mechanization

The development of ChanLib took about one person year, and not at full time effort. The model is about 300 lines of Go for the channel itself, with another 240 for the fixed-case `select` functions we test against, along with 60 lines of GooseLang for the n-ary `select`. Table 1 breaks the Rocq down by specification and by example. The case studies have a fairly low proof-to-code ratio for Perennial proofs, dominated by the effort for the non-channel aspects. Counts are raw `wc -l` totals, including whitespace and comments, over hand-written code only; the Rocq that goose generates from each Go program is excluded.

7 Related Work

Verification frameworks for real-world languages seldom cover message passing: VST [1] supports concurrency for C but verifies no message-passing framework, and Verus [19]

Table 1. Lines of code for the ChanLib Rocq development.

Specification	Rocq
Atomic update spec (§ 3)	2678
Bag (§ B.1)	184
Broadcast (§ B.2)	233
DSP (§ B.3)	2591
SPSC (§ B.4)	470
MPMC (§ B.5)	798
Future (§ B.6)	411
Handshake (§ B.7)	169
Mixtris (§ 5)	2837
Lock (§ D.3)	225
Total	**10596**

Example	Go	Rocq
`select` examples (Appendix A)	50	287
Session monitor (§ B.2)	64	205
DSP server (§ B.3)	112	388
Fibonacci (§ B.4)	22	196
Parallel search (§ B.6)	37	152
Unbounded buffer (§ 4)	117	970
Elimination stack (§ D.1)	76	519
Leader election (§ D.2)	35	282
Lock with timeout (§ D.3)	51	122
Work stealing (§ D.4)	101	713
Gobra search/replace (§ 7)	46	240
Other examples	197	408
Total	**908**	**4482**

does not support Rust's channel-like primitives. The closest work is Gobra [30], an automated verifier for Go that encodes programs into Viper and discharges proof obligations with SMT. Gobra encodes channels into Viper directly rather than modelling them, and its channel support does not extend to `select`. Its channels carry two predicates, one for resources transferred on send and one from receiver to sender, which together cover our bag and handshake resource specifications; our other examples need more. We verified the channel example from the Gobra paper, a parallel search-and-replace, using only ChanLib's bag specification: 95 lines of Gobra annotation against 217 lines of interactive Rocq, much of the difference coming from Gobra's use of SMT on the non-channel parts of the proof.

Deadlock freedom, which our specifications do not establish, is the target of a separate line of work [7, 14, 15, 18, 22]; the property is orthogonal to ours and the tools are complementary.

Channels have been modeled in Iris before with simpler semantics [16]. Somers and Krebbers [25] verify a lock-free implementation of bidirectional session channels, layering Actris's dependent separation protocols [11] over logically atomic queue specifications. We share that structure but differ in scope: they build the implementation from scratch and offer protocols as the sole high-level interface, while we model the Go runtime's own semantics and derive a range of specifications beyond protocols.

8 Conclusion

We have presented ChanLib, a model and specification of Go channels suitable for program verification. It has three components: an operational model of channels tested against the Go runtime's own tests, a general atomic update specification verified against that model and covering unbuffered channels, `select` and non-blocking operations, and a suite of resource specifications derived from it for common programming idioms. Everything is formalized in Rocq within Perennial and Goose, so real Go programs that use channels can be verified against it.

Data-Availability Statement

The Rocq development and Go code that accompany this paper are archived on Zenodo [21].

A The Not-Ready `select` Rule

Figure 4 shows two small programs, each safe only because a branch containing a panic is unreachable, and unreachable for opposite reasons. The first is proved with Chan-Select-Default; the second needs the rule this appendix states.

Neither case can fire. `select_nb_not_ready` runs a non-blocking send in one goroutine and a non-blocking receive on the same unbuffered channel in another. Both panics are unreachable, because two non-blocking operations never match: neither makes an offer, so neither can accept one. The proof makes this a statement about the channel's logical state. It maintains the invariant

$$\boxed{\exists s.\ [\![s]\!]^{\gamma}_{1/2} * s = \mathsf{Idle}}$$

The channel is therefore always Idle, and both cases are proved from this invariant. Since NonBlockingSendAU and NonBlockingRecvAU are the operation AUs with the offer conjuncts SendSlowPathAU and RecvSlowPathAU deleted, every conjunct they retain demands a state other than Idle: RcvWait for the send, SndWait(v) or Closed for the receive. Opening the invariant supplies $[\![\mathsf{Idle}]\!]^{\gamma}_{1/2}$, at which none of them is enabled, so both communication cases are discharged vacuously and only the default branches run.

This proof is impossible with the blocking AUs. Their offer conjuncts ① and ② start from Idle precisely, so the invariant would not contradict them. That is as it should be: with the `default` removed from *either* `select`, the two operations can rendezvous and the panic can run, so no proof may rule it out.

`select_nb_guaranteed_ready` is the opposite case. The channel is closed before the `select`, so the receive case is always ready: transition ⑧ fires from Closed(nil) and returns the zero value with ok = `false`. The default branch is therefore unreachable, which matters because it contains a second `close`, which panics.

This is not verifiable with Chan-Select-Default, which says what holds if a case fires but gives the default branch only a bare wp $h_{\mathrm{dflt}}()\ \{\Phi\}$, with no evidence about *why* the default was reached. Recovering that evidence costs one change to NonBlockingSendAU: replace its last conjunct, which hands over Φ_{notready} outright, by one that makes the caller prove the channel was not ready. Figure 5 gives the result. SendNotReadyAU lends out $[\![s]\!]^{\gamma}_{1/2}$ together with the fact that s admits no transition of the send, and takes it back unchanged. The logical state is the same before and after, and what the caller pays for the conjunct is a witness that the channel really was not ready.

$\mathsf{notready}_{\mathsf{snd}}$ and $\mathsf{notready}_{\mathsf{rcv}}$ hold exactly at the states with no enabled transition. Closed(_) is not among them, for either operation: a send on a closed channel panics rather than blocking, and a receive from one always succeeds. That is what settles `select_nb_guaranteed_ready`. Its receive case is never not-ready, so the caller picks $\Phi_{\mathsf{notready}} = \mathsf{False}$, is never called upon to prove it, and the default handler, second `close` and all, follows vacuously.

P in the rule is ownership the caller keeps across the attempt. Each case is given P to build its update and each case's not-ready payload returns it, so a case that declines to fire gives back everything it borrowed. The default branch receives P and every $\Phi^i_{\mathsf{notready}}$. The cases are combined with $*$ rather than $\wedge$ because every case's not-ready check really does run, and each needs its own resources.

The two atomic updates are ordered: NonBlockingSendAU implies NonBlockingSendAU′, by supplying Φ_{notready} to discharge SendNotReadyAU without opening anything. The converse fails. The order on the rules is therefore the reverse. NonBlockingSendAU′ is the weaker premise, so Chan-Select-NotReady is the stronger rule and Chan-Select-Default follows from it, by taking P to be that rule's own precondition and every $\Phi^i_{\mathsf{notready}}$ to be True: because P comes back on a declined case, each case can re-derive any other case or the default branch, which is what the $\wedge$ in Chan-Select-Default promises.

Giving the not-ready check its own conjunct is what makes this work. As one branch of the same case analysis it shared the single $\exists s.\ [\![s]\!]^{\gamma}_{1/2}$ opening with every other conjunct, so a proof would have needed the client's resources twice and $\wedge$ supplies them once. Given a conjunct of its own it opens nothing. Both programs in Figure 4 follow from the one rule.

B Resource Specifications

Each specification below is derived from the atomic update specification the same way: an invariant ties the channel's logical state to the resources in transit, the operations' atomic updates are proved from it, and the result is packaged as Hoare triples. They differ in how much of the state machine they hide and in what they let a client transfer. We also give the examples verified against each. Each specification is created by a view shift over a channel that has just been made and is still in its initial state; composing that view shift with Chan-Make-Unbuffered or Chan-Make-Buffered gives the familiar `make` triple. We start with the simplest, the bag.

B.1 Bag Channels

A bag channel carries values satisfying a predicate P: senders give up $P(v)$, receivers acquire it, close is disallowed. Figure 6 gives the specification.

The derivation defines the invariant

$$\mathsf{inv}_{\mathsf{bag}}(\mathsf{ch}, P) \triangleq \exists s.\ [\![s]\!]^{\gamma}_{1/2} * \begin{cases} P(v) & s \in \{\mathsf{SndWait}(v), \mathsf{SndDone}(v)\} \\ *_{v \in vs}\, P(v) & s = \mathsf{Buffered}(vs) \\ \mathsf{False} & s = \mathsf{Closed}(_) \\ \mathsf{True} & \text{otherwise} \end{cases}$$

```go
func select_nb_not_ready() {
  ch := make(chan struct{})
  go func() {
    select {
    case <-ch: panic("bad")
    default:
    }
  }()
  select {
  case ch <- struct{}{}: panic("bad")
  default:
  }
}
```

```go
func select_nb_guaranteed_ready() {
  x := make(chan int)
  close(x)
  select {
  case <-x: // always ready
  default:
    close(x) // must prove unreachable
  }
}
```

Figure 4. Two non-blocking `select`s. In the first, neither communication case can *ever* fire; in the second, the communication case *always* fires. Each panic is unreachable for the opposite reason.

$$\mathsf{NonBlockingSendAU}'_\gamma(\mathsf{ch}, v, \Phi, \Phi_{\mathsf{notready}}) \triangleq \mathsf{SendFastPathAU} \wedge \mathsf{SendEnqAU} \wedge \mathsf{SendClosedAU} \wedge \mathsf{SendNotReadyAU}$$

$$\mathsf{NonBlockingRecvAU}'_\gamma(\mathsf{ch}, \Phi, \Phi_{\mathsf{notready}}) \triangleq \mathsf{RecvFastPathAU} \wedge \mathsf{RecvDeqAU} \wedge \mathsf{RecvDrainAU} \wedge \mathsf{RecvClosedAU} \wedge \mathsf{RecvNotReadyAU}$$

$$\mathsf{SendNotReadyAU} \triangleq \forall s.\ \llbracket s \rrbracket^\gamma_{1/2} * \mathsf{notready}_{\mathsf{snd}}(s) \Rrightarrow \llbracket s \rrbracket^\gamma_{1/2} * \Phi_{\mathsf{notready}}$$
(RecvNotReadyAU likewise, with $\mathsf{notready}_{\mathsf{rcv}}$)

$$\mathsf{notready}_{\mathsf{snd}}(s) \triangleq \begin{cases} \mathsf{False} & s = \mathsf{RcvWait} \text{ or } s = \mathsf{Closed}(_) \\ |buf| \geq \gamma.\mathsf{cap} & s = \mathsf{Buffered}(buf) \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathsf{notready}_{\mathsf{rcv}}(s) \triangleq \begin{cases} \mathsf{False} & s = \mathsf{SndWait}(_),\ s = \mathsf{Closed}(_) \text{ or } s = \mathsf{Buffered}(v :: vs) \\ \mathsf{True} & \text{otherwise} \end{cases}$$

CHAN-SELECT-NOTREADY

$$\textstyle\bigstar_{i=1}^{n} \mathsf{IsChan}_{\gamma_i}(\mathsf{ch}_i) \vdash \forall \Phi, P, \Phi^1_{\mathsf{notready}} \ldots \Phi^n_{\mathsf{notready}}.$$

$$\textstyle\bigstar_{i=1}^{n} \left(P \mathrel{-\!\!*} \mathsf{NBAU}'_i(\Phi,\ P * \Phi^i_{\mathsf{notready}})\right) * P * \left(P \mathrel{-\!\!*} \bigstar_{i=1}^{n} \Phi^i_{\mathsf{notready}} \mathrel{-\!\!*} \mathsf{wp}\ h_{\mathsf{dflt}}()\ \{\Phi\}\right)$$

$$\mathrel{-\!\!*} \mathsf{wp}\ \texttt{select}\ldots\texttt{default}\ldots\ \{\Phi\}$$

Figure 5. The non-blocking `select` rule that reports why the default branch was reached.

$\mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch},\ P)$ Persistent fact that ch is a bag channel carrying values that satisfy P.

BAG-CREATE

$$s \in \{\mathsf{Idle}, \mathsf{Buffered}(\mathsf{nil})\} \vdash \mathsf{IsChan}_\gamma(\mathsf{ch}) * \llbracket s \rrbracket^\gamma_{1/2} \Rrightarrow \mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch}, P)$$

BAG-SEND

$$\{\mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch}, P) * P(v)\}\ \mathsf{ch} <\!\!- v\ \{\mathsf{True}\}$$

BAG-RECEIVE

$$\{\mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch}, P)\}\ \mathsf{v} := <\!\!-\mathsf{ch}\ \{P(\mathsf{v})\}$$

Figure 6. The bag specification.

with $\mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch}, P) \triangleq \mathsf{IsChan}_\gamma(\mathsf{ch}) * \boxed{\mathsf{inv}_{\mathsf{bag}}(\mathsf{ch}, P)}$.

Every conjunct is discharged the same way, and the fractions make the shape of the argument plain. The invariant holds the client's half of the state alongside the resources that half entitles it to; the operation lends in the other half at the state it found. The client opens the invariant, puts the two halves together, makes the transition, and splits them apart again, one half back into the invariant next to the

resources the *new* state calls for and one half back to the operation. Any resources the new state's case does not ask for are released to the client, and those are what establish Φ.

For a bag send this leaves nothing over: transition ① moves the state to $\mathsf{SndWait}(v)$, whose case demands $P(v)$, so the $P(v)$ the sender came in with goes into the invariant and Φ is True. A bag receive runs the same step backwards. Transition ③ moves $\mathsf{SndWait}(v)$ to RcvDone, whose case demands nothing, so the $P(v)$ sitting in the invariant is released, and it establishes the receive's continuation. The Closed case carries False, which both discharges the panic obligations and forbids closing a bag channel.

The derivation also proves the AU lemmas

$$\mathsf{isChan}_{\mathsf{bag}}(\mathsf{ch}, P) \mathrel{-\!\!*} (\forall v.\ P(v) \mathrel{-\!\!*} \Phi(v)) \mathrel{-\!\!*} \mathsf{RecvAU}_{\gamma}(\mathsf{ch}, \Phi)$$

(and its send counterpart). These are what a `select` case requires: a bag channel's case is discharged by this lemma, a broadcast channel's by its own, and the `select` rule does not distinguish them.

B.2 Broadcast Channels

A broadcast channel is an unbuffered `chan struct{}` and carries no values. Its only communication is `close`, which releases a resource P to every receiver at once and may happen only once. P must be persistent: one closer supplies it and every receiver takes a copy, so we write it $\square P$ throughout. The specification is stated over a single predicate, $\mathsf{Broadcast}(\mathsf{ch}, P, s)$, where s records what its holder knows about whether the channel has been closed: Figure 7 gives the specification.

$\mathsf{Broadcast}(\mathsf{ch}, P, s)$ is duplicable, at the cost of weakening s to Unknown: any number of goroutines may hold the channel, but the right to close it is not among the things they share.

Broadcast-Close consumes the Pending handle, so a second close is unprovable: the first one took the only copy. That is how the specification discharges the False conjunct of CloseAU on behalf of its clients. Broadcast-Receive is sound for any s because nothing is ever sent: a receive that returns at all has observed the close, so it reports Done, hands out $\square P$, and always returns `ok = false`. The Done and Unknown handles are themselves persistent, which is what lets many receivers share one.

A non-blocking receive is what turns Unknown into knowledge. As a `select` case it fires only if the channel is closed, so the case branch gets $\mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Done}) * \square P$ and the default branch gets $\mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Pending})$. The AU lemma behind this is the one § 3.6 asks of every derived specification, and it is proved from an invariant that pairs the channel's logical state with $\square P$: it sits in the invariant while the state is Idle and is released once the state is $\mathsf{Closed}(\mathsf{nil})$.

Case study: a session monitor. Figure 8 shows the pattern this specification was built for. A monitor goroutine watches a session; waiters block until the session is established. The signal is a channel closed once per session, and a fresh channel is installed for the next one, so a waiter that arrives late still sees a closed channel and returns immediately, while one that arrives early blocks until the close. The mutex protects only the variable `sessionc`, not the channel it names.

The annotations in the figure are the proof. The monitor reacquires the mutex knowing only Unknown, since another session may have closed the channel since it last looked. The non-blocking receive resolves that: if the case fires the channel was closed, and the monitor installs a fresh one, which Broadcast-Create gives at Pending; if the default branch runs, the channel it already has is Pending. Either way the monitor leaves the `select` with the one thing Broadcast-Close needs, and the close that follows is safe. `waitSession` holds only Unknown and only receives, which is all Broadcast-Receive asks, and it composes the session channel with an unrelated cancellation channel in a single `select`.

B.3 Session-Style Interactions via Dependent Separation Protocols

Paired channels in Go can be used to implement structured request–response interactions between different goroutines. These can be difficult to reason about since there is implicit protocol state while the goroutines are communicating, but the existing theory of session types is a good fit for this code.

To verify such patterns, we model the interaction between two goroutines using paired channels ch_{lr} and ch_{rl}. One goroutine (the "left" endpoint) uses ch_{lr} to send and ch_{rl} to receive, while the other (the "right" endpoint) uses ch_{rl} to send and ch_{lr} to receive. This idiom abstracts two Go channels as two endpoints corresponding to the communicating goroutines, where messages passed over either channel may depend on each other.

To specify these interactions, we use the existing theory of *dependent separation protocols* [11, 12], which describe a sequence of interdependent exchanges:

$$p \in \mathsf{iProto} ::= \mathbf{!}\,(\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p \mid \mathbf{?}(\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p \mid \mathbf{end}$$

The protocol $\mathbf{!}\,(\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p$ specifies sending the logical variables $\vec{x}:\vec{\tau}$, the physical value v, and the resources P, then continuing the protocol with p; $\mathbf{?}(\vec{x}:\vec{\tau})\ \langle v \rangle \{P\}.\ p$ is the same for receiving. The logical variables $\vec{x}:\vec{\tau}$ bind into v, P, and p, permitting dependent behavior. The **end** construct signifies the end of the protocol. The protocols are higher-order: the resources P may themselves contain protocol endpoints, enabling higher-order communication patterns where channels can be sent over channels.

To get a taste of what protocols can look like, consider this example, ported to Go from the Actris paper (which borrows it from Tassarotti et al. [27]) [11]:

Broadcast(ch, P, Pending) The holder knows ch is open, and so may close it.

Broadcast(ch, P, Done) The holder knows ch is closed, and holds P.

Broadcast(ch, P, Unknown) The holder knows neither, and must test the channel before closing it.

Broadcast-Create
$\mathsf{IsChan}_\gamma(\mathsf{ch}) * [\![\mathsf{Idle}]\!]^\gamma_{1/2} \Rrightarrow \mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Pending})$

Broadcast-Close
$\{\mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Pending}) * \square P\}$
close(ch)
$\{\mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Done})\}$

Broadcast-Receive
$\{\mathsf{Broadcast}(\mathsf{ch}, P, s)\}$
_, ok := <- ch
$\{\mathsf{Broadcast}(\mathsf{ch}, P, \mathsf{Done}) * \square P * \mathsf{ok} = \mathtt{false}\}$

Figure 7. The broadcast specification.

```
var sessionc chan struct{} = make(chan struct{})
var mu sync.Mutex

func monitorSession() {
  for {
    waitForSessionExpiration()
    mu.Lock()
      { Broadcast(sessionc, True, Unknown) }
    select {
    case <-sessionc:
      sessionc = make(chan struct{})
    default:
    }
      { Broadcast(sessionc, True, Pending) }
    mu.Unlock()
    err := newSession()
    if err != nil { continue }
    mu.Lock()
      { Broadcast(sessionc, True, Pending) }
    close(sessionc)
      { Broadcast(sessionc, True, Closed) }
    mu.Unlock()
  }
}

func waitSession[A any](
    cancel <-chan A) error {
  mu.Lock()
  s := sessionc
  mu.Unlock()
    { Broadcast(s, True, Unknown) }
  select {
  case <-s:      return nil
  case <-cancel:
    return errors.New("cancelled")
  }
}
```

Figure 8. A session monitor. `close` signals every waiter at once, and a fresh channel is installed for the next session.

```
func DSPExample() int {
  c, signal := make(chan any), make(chan any)

  go func() {
    ptr := (<-c).(*int)
    *ptr = *ptr + 2
    signal <- struct{}{}
  }()

  ptr := new(40)
  c <- ptr
  <-signal
  return *ptr // dereference to get 42
}
```

From the perspective of the main goroutine (as opposed to the one created with go), the protocol for the (c, signal) channel pair is the following:

$$!(ptr : \mathsf{loc})(x : \mathsf{int})\,\langle \mathtt{ptr}\rangle\{\mathtt{ptr} \mapsto x\}.$$
$$?\langle \mathtt{struct\{\}\{\}}\rangle\{\mathtt{ptr} \mapsto (x+2)\}.$$

This protocol allows the main goroutine to send (!) ownership of an arbitrary pointer ptr along with a value x (an integer) , and afterward receive (?) a "unit" value (struct{}{}, an empty struct value in Go) along with ownership of ptr with the value incremented by 2. Notice how the protocol sends not only a value but ownership over memory, and receives the same.

Dependent separation protocols have a notion of duality $\overline{p}$, that flips all sends and receives. Two endpoints with dual protocols are guaranteed to behave correctly; whenever one endpoint sends the other receives, and vice versa. In the example above, the spawned goroutine must be verified to follow the dual of the protocol above.

To prove our specifications using dependent separation protocols, we used the Actris ghost theory [12], which can

be instantiated for any pair of reliable channels in Iris, such as ours.

Our specifications additionally permit recursive protocols and protocol subtyping, in accordance with prior work on DSPs [12]. The specifications are sound for both unbuffered and buffered channels of any size. The separation-logic specifications that connect a protocol to Go channels and operations is shown in Figure 9.

Unlike prior work on DSP where close is a protocol action, our specifications treat closing as a separate operation that is only permitted on terminated protocols. This design reflects Go's semantics where `close` is a side-effecting operation rather than a synchronization point, and where closed channels remain readable (returning zero values with `ok=false`). The $\lightning$ resource captures this post-close state, allowing continued receives that observe the closed status.

Case study using DSP. To illustrate a simple session-style interaction, consider this example with a simple request-response stream. The `Client` function creates a stream `hw` which satisfies a dependent separation protocol that allows it to send a request string `s` on the request channel `hw.req` and receive responses on `hw.res`.

The dependent separation protocol for this example is defined recursively as:

$$\begin{aligned}&\mathsf{FProt}(\Phi_{\text{pre}}, \Phi_{\text{post}}) \triangleq \\ &\quad !\,(req : \text{string})\ \langle req\rangle\{\Phi_{\text{pre}}(req)\}. \\ &\quad ?(res : \text{string})\ \langle res\rangle\{\Phi_{\text{post}}(req, res)\}. \\ &\quad \mathsf{FProt}(\Phi_{\text{pre}}, \Phi_{\text{post}})\end{aligned}$$

SERVE-SPEC

$$\begin{aligned}&\{\forall s.\ \{\Phi_{\text{pre}}(s)\}\ f(s)\ \{s'.\ \Phi_{\text{post}}(s, s')\}\} \\ &\quad \mathtt{s} := \mathtt{Serve}(f) \\ &\{(s.req, s.res) \rightarrowtail \mathsf{FProt}(\Phi_{\text{pre}}, \Phi_{\text{post}})\}\end{aligned}$$

CLIENT-SPEC

$$\begin{aligned}&\{\mathsf{True}\} \\ &\quad \mathtt{hw} := \mathtt{Client}() \\ &\{\mathtt{hw} = \texttt{"Hello, World!"}\}\end{aligned}$$

To verify `Client`, we instantiate the protocol with

$$\begin{aligned}&\Phi_{\text{pre}}(req) \triangleq \mathsf{True} \\ &\Phi_{\text{post}}(req, res) \triangleq \\ &\quad res = req + \texttt{", World!"}\end{aligned}$$

From SERVE-SPEC the client obtains the endpoint

$$(hw.req, hw.res) \rightarrowtail \mathsf{FProt}(\Phi_{\text{pre}}, \Phi_{\text{post}}).$$

The first protocol step is a send; instantiating it with req = `"Hello"` allows the client to send `"Hello"`. Advancing the protocol yields a receive step, which provides the fact res = `"Hello, World!"`, which gives the post-condition of the CLIENT-SPEC spec.

B.4 Pipelines: SPSC and MPMC

An SPSC channel models a one-way pipeline between a single producer and a single consumer. The producer streams a sequence of values using `ch <- v` and signals the end of the stream with `close`; the consumer processes them in order, often using a `for x := range ch` to receive all the sent values.

The SPSC specification is shown in Figure 11. When using this specification, the producer maintains an assertion Producer(ch, *sent*) that records which values have been sent, and the consumer maintains Consumer(ch, *rcvd*) that records which have been received. Sends must prove a predicate $P(i, v)$ over the index in the stream and the value. On `close`, the producer shows $R(sent)$, which is the postcondition for receive on the consumer side when the channel is closed and drained (`ok` = `false`).

We verified a pipeline that produces Fibonacci numbers using the SPSC specification (Figure 12), with code adapted from the Go Tour.[6]

We define the Fibonacci pipeline sequence as $\mathsf{fib}(0) = 0$, $\mathsf{fib}(1) = 1$, and $\mathsf{fib}(n) = \mathsf{fib}(n{-}1) + \mathsf{fib}(n{-}2)$ for $n \geq 2$. We write $\mathcal{F}_n = [\,\mathsf{fib}(0), \mathsf{fib}(1), \ldots, \mathsf{fib}(n{-}1)\,]$ for the prefix of length n.

The `fibonacci` function assumes the channel satisfies $\mathsf{isChan}_{\text{spsc}}(\mathsf{c}, \mathsf{int}, P, R)$ with $P(i, v) \triangleq v = \mathsf{fib}(i)$ and $R(rcvd) \triangleq rcvd = \mathcal{F}_n$. Each iteration of `fibonacci` sends the current value x with $P(i, \mathsf{fib}(i))$, advancing the producer history to $sent{+\!\!+}[\mathsf{fib}(i)]$. The loop maintains the invariant $x = \mathsf{fib}(i)$ and $y = \mathsf{fib}(i{+}1)$. After n iterations, SPSC-CLOSE consumes Producer(c, $\mathcal{F}_n$) and $R(\mathcal{F}_n)$, which will eventually be used to prove that the correct number of elements were generated.

The consumer's `range` loop maintains the assertion Consumer(c, *results*). Each successful receive provides $P(|rcvd|, \mathtt{v})$, ensuring $\mathtt{v} = \mathsf{fib}(|rcvd|)$. It maintains an invariant that the `results` slice contains the received values. When the range loop terminates, the receive post-condition grants $R(rcvd)$, obtaining $rcvd = \mathcal{F}_{10}$ (the consumer knows n is 10 due to how it called `fibonacci`), which allows to prove that the returned slice contains exactly the elements $\mathcal{F}_{10}$.

SPSC is limited to a single producer and single consumer. We also developed a specification MPMC for the multi-producer, multi-consumer case, which tracks multisets of values but does not provide precise ordering guarantees.

B.5 MPMC

MPMC is used for multi-producer, multi-consumer channels, where the set of values are tracked precisely but there is no ordering for either producers or consumers.

[6] From https://go.dev/tour/concurrency/4, adapted only to collect results in a consumer function instead of printing.

$(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail p$ Permission to use ch_{lr} for sending and ch_{rl} for receiving, according to p.

$\lightning (\mathsf{ch}_{lr}, \mathsf{ch}_{rl})$ Resource asserting that ch_{lr} is closed.

DSP-Create

$s_1, s_2 \in \{\mathsf{Idle}, \mathsf{Buffered}(\mathsf{nil})\} \vdash$
$\mathsf{IsChan}_{\gamma_1}(\mathsf{ch}_{lr}) * \mathsf{IsChan}_{\gamma_2}(\mathsf{ch}_{rl}) * [\![s_1]\!]^{\gamma}_{1/2} * [\![s_2]\!]^{\gamma}_{1/2} \Rrightarrow$
$(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail p * (\mathsf{ch}_{rl}, \mathsf{ch}_{lr}) \rightarrowtail \overline{p}$

DSP-Send

$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail (! \vec{x} : \vec{\tau} \langle v \rangle \{P\}.\, p) * P[\vec{y}/\vec{x}]\}$
$\mathsf{ch}_{lr} \texttt{<-} v[\vec{y}/\vec{x}]$
$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail p[\vec{y}/\vec{x}]\}$

DSP-Receive

$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail (? \vec{x} : \vec{\tau} \langle v \rangle \{P\}.\, p)\}$
$w := \texttt{<-}\, \mathsf{ch}_{rl}$
$\{\exists (\vec{y} : \vec{\tau}).\, w = v[\vec{y}/\vec{x}] * (\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail p[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$

DSP-Close

$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail \mathbf{end}\}$
$\mathtt{close}(\mathsf{ch}_{lr})$
$\{\lightning (\mathsf{ch}_{lr}, \mathsf{ch}_{rl})\}$

DSP-ReceiveEnd

$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail \mathbf{end}\}$
$_, b := \texttt{<-}\, \mathsf{ch}_{rl}$
$\{(\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) \rightarrowtail \mathbf{end} * b = \mathtt{false}\}$

DSP-ReceiveClosed

$\{\lightning (\mathsf{ch}_{lr}, \mathsf{ch}_{rl})\}$
$_, b := \texttt{<-}\, \mathsf{ch}_{rl}$
$\{\lightning (\mathsf{ch}_{lr}, \mathsf{ch}_{rl}) * b = \mathtt{false}\}$

Figure 9. Session-style interaction over a pair of channels, using dependent separation protocols.

```go
type stream struct {
  req chan string
  res chan string
}

func appWrld(s string) string {
  return s + ", World!"
}

func Client() string {
  hw := Serve(appWrld)
  hw.req <- "Hello"
  return <-hw.res
}
```

```go
func Serve(f func(string) string)
    stream {
  s := stream{
    req: make(chan string),
    res: make(chan string),
  }
  go func() {
    for {
      s.res <- f(<-s.req)
    }
  }()
  return s
}
```

Figure 10. A request-response stream verified with the DSP specification.

The stream ends in three steps. Producers finish and one of them closes the channel, depositing $R(\biguplus \vec{X})$; consumers keep receiving, each receive with ok = `true` yielding $P(v)$. The receive that finds the channel closed and empty returns ok = `false` and yields isDrained(ch), which is how a consumer learns that the stream is finished and the only way to learn it. With isDrained(ch) in hand, MPMC-Collect trades the n_c consumer tokens for $R(\biguplus \vec{Y})$. Closing therefore establishes nothing on its own: a buffered channel closed with values still in it is not drained until the last of them is taken.

B.6 Future

The *future* idiom models the pattern where multiple goroutines independently compute results and send them to a shared channel, and a single consumer collects all results without caring about the order in which they arrive. A canonical example is the replicated search pattern presented in a Go concurrency talk by Go author Rob Pike [20]:

```go
func Google(query string) (results []Result) {
    c := make(chan Result, 3)
    go func() { c <- Web(query) }()
    go func() { c <- Image(query) }()
    go func() { c <- Video(query) }()
    for i := 0; i < 3; i++ {
        result := <-c
        results = append(results, result)
    }
    return
}
```

This pattern is natural to write but subtle to specify: While the waiting consumer knows what each goroutine will produce (a web result, an image result, a video result), it is not possible to know *in what order* the results will arrive. The bag specification can be used for waiting on a single result or a group of indistinguishable results, but cannot differentiate distinct work items.

The future specification, shown in Figure 14, addresses this by maintaining a pending list of contracts for each launched goroutine and specifies that each receive resolves exactly one contract from the list at an unspecified position.

The specification introduces three resources: Fulfill(ch, P) is a one-shot token held by a worker goroutine that

$\mathsf{isChan}_{\mathsf{spsc}}(\mathsf{ch}, T, P, R)$ Persistent fact that ch is an SPSC pipeline over values of type T.

$\mathsf{Producer}(\mathsf{ch}, \mathit{sent})$ Permission to send, and tracks history *sent*.

$\mathsf{Consumer}(\mathsf{ch}, \mathit{rcvd})$ Right to receive the next value, and tracks history *rcvd*.

SPSC-Create

$s \in \{\mathsf{Idle}, \mathsf{Buffered}(\mathsf{nil})\} \vdash$
$\mathsf{IsChan}_\gamma(\mathsf{ch}) * [\![s]\!]^\gamma_{1/2} \Rrightarrow$
$\mathsf{isChan}_{\mathsf{spsc}}(\mathsf{ch}, T, P, R) * \mathsf{Producer}(\mathsf{ch}, \mathsf{nil}) * \mathsf{Consumer}(\mathsf{ch}, \mathsf{nil})$

SPSC-Send

$\left\{ \begin{array}{l} \mathsf{isChan}_{\mathsf{spsc}}(\mathsf{ch}, T, P, R) * \\ \mathsf{Producer}(\mathsf{ch}, \mathit{sent}) * \\ P(|\mathit{sent}|, \mathsf{v}) \end{array} \right\}$
$\mathsf{ch} <\!\!- \mathsf{v}$
$\{\, \mathsf{Producer}(\mathsf{ch}, \mathit{sent} +\!\!+ [\mathsf{v}]) \,\}$

SPSC-Close

$\left\{ \begin{array}{l} \mathsf{isChan}_{\mathsf{spsc}}(\mathsf{ch}, T, P, R) * \\ \mathsf{Producer}(\mathsf{ch}, \mathit{sent}) * \\ R(\mathit{sent}) \end{array} \right\}$
$\mathsf{close}(\mathsf{ch})$
$\{\, \mathsf{True} \,\}$

SPSC-Receive

$\{\mathsf{isChan}_{\mathsf{spsc}}(\mathsf{ch}, T, P, R) * \mathsf{Consumer}(\mathsf{ch}, \mathit{rcvd})\}$
$\mathsf{v}, \mathsf{ok} := <\!\!- \mathsf{ch}$
$\{\, \text{if ok then } P(|\mathit{rcvd}|, \mathsf{v}) * \mathsf{Consumer}(\mathsf{ch}, \mathit{rcvd} +\!\!+ [\mathsf{v}]) \text{ else } R(\mathit{rcvd}) * \mathsf{v} = \mathsf{zero}\, T \,\}$

Figure 11. The SPSC specification.

```
1 func fibonacci(n int, c chan int) {
2   x, y := 0, 1
3   for i := 0; i < n; i++ {
4     c <- x
5     x, y = y, x+y
6   }
7   close(c)
8 }
```

```
1 func fib_consumer() []int {
2   c := make(chan int, 10)
3   go fibonacci(cap(c), c)
4   results := []int{}
5   for i := range c {
6     results = append(results, i)
7   }
8   return results
9 }
```

Figure 12. A Fibonacci pipeline verified with the SPSC specification, adapted from the Go Tour.

must satisfy $P(v)$ to send value v after completing work. $\mathsf{Await}(\mathsf{ch}, \mathit{pending})$ is held by the consumer and tracks the list of contracts not yet matched to a received value. $\mathsf{isChan}_{\mathsf{future}}(\mathsf{ch})$ is a persistent channel invariant, as usual. Verifying channel sends is straightforward: Future-Fulfill consumes $\mathsf{Fulfill}(\mathsf{ch}, P)$ and contract $P(v)$ and permits sending v on the channel.

The consumer side has to handle the nondeterminism. Future-Await specifies that each receive removes *some* P from *pending*, at some split point $\mathit{pre} +\!\!+ P :: \mathit{post}$. A future channel is never closed, so the receive always returns ok = `true`. The caller learns: (a) what split was chosen, (b) the value v returned by the receive, and (c) that $P(v)$ holds. The remaining $\mathsf{Await}(\mathsf{ch}, \mathit{pre} +\!\!+ \mathit{post})$ is returned for the next receive. The caller cannot predict *which* P is chosen, since arrival order depends on scheduling.

Verification of the* `Google` *replicated search. We verify `Google` with implementations of the `Web`, `Image` and `Video` functions that return a unique file name for the given query. After creating the channel, we use Future-Register three times to register contracts P_{web}, P_{img}, P_{vid} that are defined as the postconditions of their respective functions, building up $\mathsf{Await}(c, [P_{\text{web}}, P_{\text{img}}, P_{\text{vid}}])$. Each goroutine holds the corresponding Fulfill token and provides the postcondition resource gained from running their search function before sending.

The `for` loop iterates three times. Each iteration applies Future-Await, which peels off one contract from *pending* at an existentially chosen position. After three receives, *pending* is exhausted. The postcondition of `Google` states that the result slice is a permutation of the three expected results. This is established by collecting the three existential witnesses (P_k, v_k) across the loop iterations: each witness gives $v_k = \mathit{expected}_k$ for one $k \in \{\text{web}, \text{img}, \text{vid}\}$, and the three witnesses together cover all three contracts—since the contracts are distinct and each can be matched at most once. This last step uses injectivity of the contracts (distinct extensions `.html`, `.png`, `.mp4` are distinguishable) to conclude that the collected values are exactly the expected set, up to permutation.

$\mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch},\ T,\ P,\ R,\ n_p,\ n_c)$ Persistent fact that ch is an MPMC pipeline for values of type T, with n_p producers and n_c consumers.

$P : T \to \mathsf{iProp}$ Per-value ownership predicate.

$R : \mathcal{M}(T) \to \mathsf{iProp}$ Predicate on the multiset of values sent at close.

$\mathsf{Producer}(\mathsf{ch},\ X)$ Right to send on ch, tracking sent values in multiset $X : \mathcal{M}(T)$.

$\mathsf{Consumer}(\mathsf{ch},\ Y)$ Right to receive on ch, tracking received values in $Y : \mathcal{M}(T)$.

$\mathsf{isDrained}(\mathsf{ch})$ Persistent fact that ch is closed *and* every sent value has been received.

MPMC-Create

$$s \in \{\mathsf{Idle}, \mathsf{Buffered}(\mathsf{nil})\},\ n_p > 0,\ n_c > 0 \vdash \mathsf{IsChan}_\gamma(\mathsf{ch}) * [\![s]\!]^\gamma_{1/2} \Rrightarrow \mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch}, T, P, R, n_p, n_c) * \mathop{\Large *}_{i=1}^{n_p} \mathsf{Producer}(\mathsf{ch}, \emptyset) * \mathop{\Large *}_{j=1}^{n_c} \mathsf{Consumer}(\mathsf{ch}, \emptyset)$$

MPMC-Send

$$\left\{ \begin{array}{l} \mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch}, T, P, R, n_p, n_c) * \\ \mathsf{Producer}(\mathsf{ch}, X) * P(v) \end{array} \right\}$$
$$\mathsf{ch} \mathrel{<\!\!-} v$$
$$\{\ \mathsf{Producer}(\mathsf{ch},\ X \uplus \{v\})\ \}$$

MPMC-Receive

$$\left\{ \mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch}, T, P, R, n_p, n_c) * \mathsf{Consumer}(\mathsf{ch}, Y) \right\}$$
$$v, ok := \mathrel{<\!\!-} \mathsf{ch}$$
$$\left\{\ \text{if } ok \text{ then } P(v) * \mathsf{Consumer}(\mathsf{ch},\ Y \uplus \{v\}) \text{ else } \mathsf{isDrained}(\mathsf{ch}) * \mathsf{Consumer}(\mathsf{ch}, Y) * v = \mathsf{zero}\, T\ \right\}$$

$$\text{Let } \vec{X} = [X_1, \ldots, X_{n_p}] \text{ and } \vec{Y} = [Y_1, \ldots, Y_{n_c}].$$

MPMC-Close

$$\left\{ \begin{array}{l} \mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch}, T, P, R, n_p, n_c) * \\ \mathop{\Large *}_{X \in \vec{X}} \mathsf{Producer}(\mathsf{ch}, X) * R(\biguplus \vec{X}) \end{array} \right\}$$
$$\mathtt{close(ch)}$$
$$\{\ \mathsf{True}\ \}$$

MPMC-Collect

$$\mathsf{isChan}_{\mathsf{mpmc}}(\mathsf{ch}, T, P, R, n_p, n_c) * \mathsf{isDrained}(\mathsf{ch}) \vdash \mathop{\Large *}_{Y \in \vec{Y}} \mathsf{Consumer}(\mathsf{ch}, Y) \Rrightarrow R(\biguplus \vec{Y})$$

Figure 13. The MPMC specification.

B.7 Handshake

A *handshake* is a synchronous exchange on an *unbuffered* channel. Each send/receive pair transfers two pieces of ownership: the sender provides $P(v)$ and receives Q, while the receiver obtains $P(v)$ and must provide Q in return. This captures bidirectional ownership transfer across the rendezvous. The specification is in Figure 15. A handshake channel is never closed, which the invariant records, so Handshake-Receive always returns ok = true.

C Unbounded: The Buffer's Atomic Updates

§ 4 verifies unbounded.go, a buffer from gRPC-Go's internals, and gives its clients three atomic updates of their own. They are stated here, over an abstract state that is the channel's own buffered and closed states with the capacity bound dropped:

$$\begin{array}{rcl} \mathsf{Ub}\ V & ::= & \mathsf{Buffered}(\mathit{buff} : \mathsf{list}\, V) \\ & | & \mathsf{Closed}(\mathit{drain} : \mathsf{list}\, V) \end{array}$$

$\mathsf{OwnUb}(s)$ is the client's half of the buffer's state, as $[\![s]\!]^\gamma_{1/2}$ is for the channel's. UbSendAU is what Put takes, UbRecvAU what a receive off Get() takes, and UbCloseAU what Close takes; each is named for the channel operation it corresponds to rather than the Go method. Figure 16 states them.

NewUnbounded has an ordinary Hoare triple, since it is a function in the verified package rather than a language primitive: it returns $\mathsf{isUnbounded}(b)$ together with $\mathsf{OwnUb}(\mathsf{Buffered}(\mathsf{nil}))$. Put and Close take the updates above. Get returns the bare channel with its IsChan, so a client receives from it with Chan-Receive: this is the one place the wrapper is deliberately not opaque, and a bridge lemma turns UbRecvAU into the channel's RecvAU, discharging the two rendezvous conjuncts from the fact that a capacity-1 channel is always Buffered or Closed. Load needs no update at all, and its specification is {True} b.Load() {True}: whichever of its three outcomes fires, moving the head of the backlog into c, doing nothing because c is full, or closing c once closing is set and the backlog has drained, the abstract queue is unchanged.

$\mathsf{isChan}_{\mathsf{future}}(\mathsf{ch})$ Persistent fact that ch is used as a future collection channel.

$\mathsf{Fulfill}(\mathsf{ch}, P)$ Producer token: a commitment to send some v with $P(v)$.

$\mathsf{Await}(\mathsf{ch}, \mathit{pending})$ Consumer token: tracks the list of contracts not yet received.

FUTURE-CREATE

$$s \in \{\mathsf{Idle}, \mathsf{Buffered}(\mathsf{nil})\} \vdash \mathsf{IsChan}_\gamma(\mathsf{ch}) * [\![s]\!]^\gamma_{1/2} \Rrightarrow \mathsf{isChan}_{\mathsf{future}}(\mathsf{ch}) * \mathsf{Await}(\mathsf{ch}, \mathsf{nil})$$

FUTURE-REGISTER

$$\mathsf{isChan}_{\mathsf{future}}(\mathsf{ch}) \vdash \forall P.\ \mathsf{Await}(\mathsf{ch}, \mathit{pending}) \Rrightarrow \mathsf{Fulfill}(\mathsf{ch}, P) * \mathsf{Await}(\mathsf{ch}, \mathit{pending} +\!\!+ [P])$$

FUTURE-FULFILL

$\{\mathsf{isChan}_{\mathsf{future}}(\mathsf{ch}) * \mathsf{Fulfill}(\mathsf{ch}, P) * P(v)\}$

ch <- v

$\{\mathsf{True}\}$

FUTURE-AWAIT

$\{\mathsf{isChan}_{\mathsf{future}}(\mathsf{ch}) * \mathsf{Await}(\mathsf{ch}, \mathit{pending})\}$

v := <- ch

$\left\{ \begin{array}{l} \exists P,\ \mathit{pre},\ \mathit{post}.\ \mathit{pending} = \mathit{pre} +\!\!+ P :: \mathit{post} \\ \quad * \ P(v) \ * \ \mathsf{Await}(\mathsf{ch}, \mathit{pre} +\!\!+ \mathit{post}) \end{array} \right\}$

Figure 14. The future specification.

$\mathsf{isChan}_{\mathsf{hs}}(\mathsf{ch}, T, P, Q)$ Persistent fact that ch is an *unbuffered* handshake channel of type T.

$P : T \to \mathsf{iProp}$ Ownership transferred from sender to receiver.

$Q : \mathsf{iProp}$ Ownership transferred from receiver back to sender.

HANDSHAKE-CREATE

$$\mathsf{IsChan}_\gamma(\mathsf{ch}) * [\![\mathsf{Idle}]\!]^\gamma_{1/2} \Rrightarrow \mathsf{isChan}_{\mathsf{hs}}(\mathsf{ch}, T, P, Q)$$

HANDSHAKE-SEND

$\{\mathsf{isChan}_{\mathsf{hs}}(\mathsf{ch}, T, P, Q) * P(v)\}$

ch <- v

$\{Q\}$

HANDSHAKE-RECEIVE

$\{\mathsf{isChan}_{\mathsf{hs}}(\mathsf{ch}, T, P, Q) * Q\}$

w := <- ch

$\{\exists v{:}T.\ \mathtt{w} = v * P(v)\}$

Figure 15. The handshake specification.

D Case Studies

The case studies below exercise the specification from different directions. Each is a Go program we verified against ChanLib; Table 1 gives their sizes.

D.1 Elimination Stack: A Channel as an Exchanger

§ 3.3 states that an unbuffered channel is not linearizable, and points here for the object built on one that is. The elimination stack's exchange point is a Go channel, and its clients see an ordinary linearizable stack.

```
type ElimStack struct {
  base      *LockedStack
  exchanger chan string // unbuffered
}
```

The elimination stack [10] optimizes a mutex-protected stack under contention: a concurrent push and pop can cancel out, handing the pushed value directly to the popper without touching the stack. The Go implementation uses an unbuffered channel as the exchange point: Push selects between sending on the exchanger and a time.After timeout, falling back to the locked stack on timeout; Pop mirrors it with a receive.

No value is ever stored in the exchanger, and the code is correct only because the matched send in Push and receive in Pop take effect together: the pop must take effect while the push is still in flight, so that the stack appears untouched by the eliminated pair.

The specification clients see. Clients see an ordinary atomic stack, with one linearization point per operation. The abstract state is $\mathsf{Stack}(xs)$, half of a ghost variable holding the stack contents, and Push and Pop get logically atomic specifications over it:

$$\mathsf{PushAU}_\gamma(s, v, \Phi) \triangleq \forall xs.\ \mathsf{Stack}(xs) \Rrightarrow \mathsf{Stack}(v :: xs) * \Phi$$

$$\mathsf{PopAU}_\gamma(s, \Phi) \triangleq \forall xs.\ \mathsf{Stack}(xs) \Rrightarrow \begin{cases} \mathsf{Stack}(\mathsf{nil}) * \Phi(\mathsf{zero}\, V, \mathtt{false}) & \text{if } xs = \mathsf{nil} \\ \mathsf{Stack}(xs') * \Phi(v, \mathtt{true}) & \text{if } xs = v :: xs' \end{cases}$$

No channel appears in either update. This is the specification of a concurrent stack, and a client of ElimStack need not know that a channel is used at all.

OwnUb(s) The client's half of the buffer's abstract state.

UbSendAU
$$\begin{aligned}&\mathsf{UbSendAU}_\gamma(v,\Phi) \triangleq \forall \mathit{buff}, \mathit{drain}.\\ &\quad \mathsf{OwnUb}(\mathsf{Buffered}(\mathit{buff})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Buffered}(\mathit{buff} +\!\!+ [v])) * \Phi\\ &\quad \wedge\ \mathsf{OwnUb}(\mathsf{Closed}(\mathit{drain})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Closed}(\mathit{drain})) * \Phi\end{aligned}$$

UbCloseAU
$$\begin{aligned}&\mathsf{UbCloseAU}_\gamma(\Phi) \triangleq \forall \mathit{buff}, \mathit{drain}.\\ &\quad \mathsf{OwnUb}(\mathsf{Buffered}(\mathit{buff})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Closed}(\mathit{buff})) * \Phi\\ &\quad \wedge\ \mathsf{OwnUb}(\mathsf{Closed}(\mathit{drain})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Closed}(\mathit{drain})) * \Phi\end{aligned}$$

UbRecvAU
$$\begin{aligned}&\mathsf{UbRecvAU}_\gamma(\Phi) \triangleq \forall v, \mathit{buff}, \mathit{drain}.\\ &\quad \mathsf{OwnUb}(\mathsf{Buffered}(v :: \mathit{buff})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Buffered}(\mathit{buff})) * \Phi(v, \mathtt{true})\\ &\quad \wedge\ \mathsf{OwnUb}(\mathsf{Closed}(v :: \mathit{drain})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Closed}(\mathit{drain})) * \Phi(v, \mathtt{true})\\ &\quad \wedge\ \mathsf{OwnUb}(\mathsf{Closed}(\mathsf{nil})) \Rrightarrow\\ &\qquad \mathsf{OwnUb}(\mathsf{Closed}(\mathsf{nil})) * \Phi(\mathsf{zero}\, V, \mathtt{false})\end{aligned}$$

Figure 16. The atomic updates of Unbounded.

The linearization points. On the slow path, where Push and Pop both fall back to the locked stack, each linearizes at its own critical section. On an elimination, both linearize at the *same* instant: whichever thread fires the commit transition runs both updates back to back, with the invariant open, and the pair takes effect together. The channel underneath is not linearizable (§ 3.3); the stack built on it is, and the rendezvous's two linearization points permit this.

The proof. The other half of Stack(xs) lives in a single invariant, next to the locked stack's contents, the exchanger's OwnChan, and one further component that cases on the channel's logical state:

Idle	both halves of the exchange tokens
SndWait(v)	the pusher's atomic update
RcvWait	the popper's atomic update
SndDone(v)	a witness that the pop's update has run
RcvDone	a witness that the push's update has run
otherwise	False

The two offer states hold *the waiting thread's atomic update itself*. When a push finds the channel Idle it fires ①, and in the same step deposits its own update into the invariant, keeping only the residual shift ⑤. A pop that later fires ③ finds that update waiting for it and runs it, together with its own, while it holds Stack(xs). The two compose to the identity on the abstract stack: the push pushes v, the pop pops it, and the locked stack is never touched. This is *helping*: the committing thread does the waiting thread's logical work. It is possible only because ① and ③ are separate transitions, with the invariant open in between.

What the waiting thread gets back is a witness rather than its postcondition: RcvDone carries a one-shot token, SndDone(v) a fractional ghost variable naming the value that was exchanged. When the pusher's residual ⑤ fires it recovers its postcondition from the witness through an escrow invariant, and the popper's ⑥ does the same. The escrow allows a continuation Φ, chosen by the caller and unknown to the invariant, to be stored and recovered later.

The remaining obligations are discharged without further work. The exchanger is unbuffered and never closed, so every channel state the invariant does not list is False: the buffered and closed conjuncts of SendAU and RecvAU are refuted from the invariant exactly as in Appendix A. The timeout case of each `select` is discharged by the bag specification of § B.1: an exchanger and a timer, two channels with unrelated specifications, composed in one `select` with no reasoning specific to either.

Exchangers whose matched operations commit together have been studied as concurrency-aware objects [9], and Compass [5] gives a logically atomic specification for a relaxed-memory exchanger, verifying an elimination stack on top.

D.2 Leader Election over Mixtris

§ 5 instantiates the Mixtris protocol theory over ChanLib. Here we verify that paper's running example (its §2.4), three-way leader election, over Go channels (Figure 17). Three participants in a ring each race a mixed-choice `select`: send my identifier clockwise, or receive from counter-clockwise and be elected. Each participant runs the protocol

$$(!\,[\mathit{nxt}]\,\langle \mathit{id}\rangle.\,\mathbf{end}) + (?[\mathit{prv}]\,\langle v\rangle\{\mathit{leader}\}.\,\mathbf{end})$$

Consistency guarantees that at most one exchange occurs, hence at most one leader. It is proved by simulating every

```
func ThreeWayElectionParty(send, recv chan uint64,
                           id uint64, done chan uint64) {
  select {
  case send <- id: // not elected
  case <-recv:     // elected: sole permission to close
    close(done)
  }
}
```

Figure 17. Three-way leader election (one participant). `ThreeWayElection` allocates one channel per ring edge and forks three parties.

possible communication: three exchanges, each leaving a pool in which one participant still has work.

In the original, the leader's privilege is a heap location that the leader frees; a second leader would be a double free. Go has no `free`, but it has an operation with the same single-use character: closing a channel. Our leader's privilege is the exclusive permission to `close(done)`, which in ChanLib terms is $\mathsf{OwnChan}_{\gamma_{done}}(\mathit{done}, \mathsf{Idle})$, carried as the payload P of the election protocol's receive branch. It is as single-use as the original's $l \mapsto -$ for exactly the reason § 3.4 gives: CloseAU demands False in the Closed state. Note that done is an ordinary Go channel and not part of the session; the session's own edge channels are never closed at all.

The top-level specification

$$\{\mathsf{True}\}\ \texttt{ThreeWayElection()}\ \{\mathsf{True}\}$$

then certifies leader uniqueness by adequacy, exactly as in the original: a provable weakest precondition means the program is safe, and the program contains a `close` that panics if reached twice. The proof of each participant is a two-case `select`: one Mixtris-Send case, one Mixtris-Recv case whose payload carries the close permission, and nothing else.

D.3 A Lock with a Timeout

A capacity-one channel is a mutex: acquiring sends a unit value, releasing receives it, and the buffer holds the lock state. The specification is created from a channel whose capacity is exactly one, together with the resource R it will protect. Acquiring with a timeout is a `select` between the send and a timeout signal, so an acquire can fail.

The client sees a single persistent $\mathsf{isLock}(\ell, R)$: acquiring returns R and releasing consumes it, so no separate locked token is needed. It is derived from an invariant that cases on the channel's logical state and holds R exactly while the buffer is empty: a send moves $\mathsf{Buffered}(\mathsf{nil})$ to $\mathsf{Buffered}([()])$, taking R out of the invariant, and the matching receive puts it back. The capacity is what makes this an invariant and not merely a protocol. Nothing here would hold of a channel with room for two values, so a specification that treated Go channels as unbounded asynchronous queues could not establish mutual exclusion at all.

The timeout is the reason the example is a `select` rather than a send. Only the branch in which the send case fires may claim R; the timeout branch is verified without it, so the operation returns a boolean and its postcondition is R exactly when that boolean is true. The timeout channel, a `time.After` signal, is discharged by the bag specification of § B.1, with no reasoning connecting it to the lock. A non-blocking `TryLock` has the same shape, with $\Phi_{\mathsf{notready}} = \mathsf{True}$: failing to acquire tells the caller nothing.

D.4 Work Stealing

Figure 18 is a word-count program whose workers steal tasks from one another. It is the largest of our examples and the one that uses the most of the API at once: buffered channels as task queues, a higher-order channel carrying reply channels, `select` with and without `default`, and `close` as the termination signal.

Each worker owns a buffered queue of documents and a `steal` channel of type `chan chan *string`, over which a thief sends the reply channel it wants the answer on. All documents start on worker 0's queue, so the program is correct only if stealing works. The workers share a done channel, closed by whichever worker decrements `remaining` to zero, and every worker returns once it observes that close.

The `select` in `run` mixes a receive on the shared done channel, a receive on the worker's own queue, and a receive of a steal request, and its cases are discharged independently: done is a broadcast channel and the other two are bags, each supplying its own AU lemma, and the `select` rule composes them without relating the three. Both of its `selects` carry a `default` branch and are verified with the non-blocking rule of § 3.7: the outer one falls through to stealing from a neighbour, and in the inner one the thief learns from `reply <- nil` that the victim's queue was empty. Ownership of a document, $\mathsf{Task}(\mathit{doc})$, moves from queue to thief to worker without ever being duplicated, which is what makes the final total a sum over each document exactly once.

The close is guarded by an atomic counter rather than a flag. done is a broadcast channel (§ B.2), so closing it needs its Pending handle, and that handle sits in the invariant holding the counter: the worker that decrements it to zero is the only one that can take the handle, so it is the only one that closes. The coordinator's own receive on done then returns, and its postcondition is that the value returned is the total word count of the input.

```
type Worker struct {
  queue chan string
  steal chan chan *string
}

type shared struct {
  remaining *atomic.Int64
  total     *atomic.Int64
  done      chan struct{}
}

{ isWorker(w) ∗ isCoordinator(sh) ∗ Task(doc) }
func (w *Worker) process(doc string, sh shared) {
  sh.total.Add(int64(len(strings.Fields(doc))))
  if sh.remaining.Add(-1) == 0 { close(sh.done) }
}

{ isWorker(w) ∗ isWorker(neighbor) ∗ isCoordinator(sh) }
func (w *Worker) run(
    neighbor *Worker,
    sh shared) {
  for {
    select {
    case <-sh.done:
      return
    case doc := <-w.queue:
      w.process(doc, sh)
    case reply := <-w.steal:
      select {
      case doc := <-w.queue:
        { Task(doc) }
        reply <- &doc
      default:
        reply <- nil
      }
    // run() loop, cont'd
    default:
      reply := make(chan *string, 1)
      select {
      case <-sh.done:
        return
      case neighbor.steal <- reply:
        if doc := <-reply; doc != nil {
          w.process(*doc, sh)
        }
      case doc := <-w.queue:
        w.process(doc, sh)
      }
    } // select
  } // for
} // func run

// top-level coordinating function (high-level flow)
{ docs ↦∗ ds }
func wordCount(docs []string) int64 {
  var workers []*Worker = ... // initialize buffered channels
  // (imbalanced) initial distribution of tasks:
  for _, doc := range docs { workers[0].queue <- doc }
  sh := shared{...} // initialize
  { isCoordinator(sh) }
  for i, w := range workers {
    neighbor := ... // get next in array
    go w.run(neighbor, sh)
  }
  <-sh.done
  return sh.total.Load()
}
{ ret = ∑_{d∈ds} wc(d) }
```

Figure 18. Work stealing. The `steal` channel carries reply channels, and `done` is closed once by whichever worker finishes last.

Acknowledgments

We thank Luke Zarko for feedback on this submission, and Ethan Cecchetti, Joseph Tassarotti, and the Wisconsin MadPL group for feedback on earlier drafts. We also thank the anonymous reviewers for their careful reading and their suggestions.

References

[1] Andrew W. Appel. 2011. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbrücken, Germany) *(ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17. ISBN 9783642197178. doi:10.1007/978-3-642-19718-5_1

[2] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2018. Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability. *J. ACM* 65, 6, Article 45 (Nov. 2018), 42 pages. doi:10.1145/3266457

[3] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 243–258. ISBN 9781450368735. doi:10.1145/3341301.3359632

[4] Tej Chajed, Joseph Tassarotti, M Frans Kaashoek, and Nickolai Zeldovich. 2020. Verifying Concurrent Go Code in Coq with Goose. In *Proceedings of the 6th International Workshop on Coq for Programming Languages (CoqPL)*. New Orleans, LA.

[5] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 792–808. ISBN 9781450392655. doi:10.1145/3519939.3523451

[6] Nicolas Dilley and Julien Lange. 2019. An Empirical Study of Messaging Passing Concurrency in Go Projects. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 377–387. doi:10.1109/SANER.2019.8668036

[7] Nicolas Dilley and Julien Lange. 2020. Bounded Verification of Message-Passing Concurrency in Go Using Promela and Spin. *Electronic Proceedings in Theoretical Computer Science* 314 (April 2020), 34–45. arXiv:2004.01323 [cs] doi:10.4204/EPTCS.314.4

[8] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 16 (Nov. 2021), 134 pages. doi:10.1145/3477082

[9] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. 2015. Modular Verification of Concurrency-Aware Linearizability. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363* (Tokyo, Japan) *(DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 371–387. ISBN 9783662486528. doi:10.1007/978-3-662-48653-5_25

[10] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 9780123973375.

[11] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–30. doi:10.1145/3371074

[12] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). doi:10.46298/LMCS-18(2:16)2022

[13] Jonas Kastberg Hinrichsen, Iwan Quémerais, and Lars Birkedal. 2026. Mixtris: Mechanised Higher-Order Separation Logic for Mixed Choice Multiparty Message Passing. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 116 (April 2026), 26 pages. doi:10.1145/3798224

[14] Jules Jacobs and Stephanie Balzer. 2023. Higher-Order Leak and Deadlock Free Locks. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1027–1057. doi:10.1145/3571229

[15] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. 2024. Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 1385–1417. doi:10.1145/3632889

[16] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 637–650. ISBN 9781450333009. doi:10.1145/2676726.2676980

[17] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 205–217. ISBN 978-1-4503-4660-3. doi:10.1145/3009837.3009855

[18] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go Using Behavioural Types. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 1137–1148. ISBN 978-1-4503-5638-1. doi:10.1145/3180155.3180157

[19] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. ISBN 9798400712517. doi:10.1145/3694715.3695952

[20] Rob Pike. 2012. Go Concurrency Patterns. Google I/O, https://go.dev/talks/2012/concurrency.slide. Video available at https://www.youtube.com/watch?v=f6kdp27TYZs.

[21] Levi Redlin, Upamanyu Sharma, Jonas Kastberg Hinrichsen, and Tej Chajed. 2026. *ChanLib: Reconciling Choice, Rendezvous, Broadcast, and Buffering in a Mechanized Specification of Go Channels (artifact)*. Zenodo. doi:10.5281/zenodo.24455475 Version 1.0.

[22] Georgian-Vlad Saioc, Julien Lange, and Anders Møller. 2024. Automated Verification of Parametric Channel-Based Process Communication. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (Oct. 2024), 2070–2096. doi:10.1145/3689784

[23] William N. Scherer, Doug Lea, and Michael L. Scott. 2009. Scalable synchronous queues. *Commun. ACM* 52, 5 (May 2009), 100–111. doi:10.1145/1506409.1506431

[24] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, Koblenz Germany, 113–129. doi:10.1145/3600006.3613172

[25] Thomas Somers and Robbert Krebbers. 2024. Verified Lock-Free Session Channels with Linking. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 292 (Oct. 2024), 30 pages. doi:10.1145/3689732

[26] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag, Berlin, Heidelberg, 149–168. ISBN 9783642548321. doi:10.1007/978-3-

642-54833-8_9

[27] Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Vol. 10201. Springer Berlin Heidelberg, Berlin, Heidelberg, 909–936. ISBN 978-3-662-54433-4 978-3-662-54434-1. doi:10.1007/978-3-662-54434-1_34

[28] The Go Authors. 2025. The Go Programming Language Specification. https://go.dev/ref/spec. Language version go1.25.

[29] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. ISBN 9781450362405. doi:10.1145/3297858.3304069

[30] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. 2021. Gobra: Modular Specification and Verification of Go Programs. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 367–379. ISBN 978-3-030-81684-1. doi:10.1007/978-3-030-81685-8_17