Refinement Proofs in Rust Using Ghost Locks

Refinement Proofs in Rust Using Ghost Locks

AUREL BÍLÝ, ETH Zurich, Switzerland
JOÃO C. PEREIRA, ETH Zurich, Switzerland
JAN SCHÄR, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Refinement transforms an abstract system model into a concrete, executable program, such that properties established for the abstract model carry over to the concrete implementation. Refinement has been used successfully in the development of substantial verified systems. Nevertheless, existing refinement techniques have limitations that impede their practical usefulness. Some techniques generate executable code automatically, which generally leads to implementations with sub-optimal performance. Others employ bottom-up program verification to reason about efficient implementations, but impose strict requirements on the structure of the code, the structure of the refinement proofs, as well as the employed verification logic and tools.

In this paper, we present a novel refinement technique that removes these limitations. It supports a wide range of program structures, data representations, and proof structures. Our approach supports reasoning about both safety and liveness properties. We implement our approach in a state-of-the-art verifier for the Rust language, which itself offers a strong foundation for memory safety. We demonstrate the practicality of our approach on a number of substantial case studies.

1 INTRODUCTION

Refinement is a technique to connect an abstract model of a system to another, lower-level formulation of the same system. It is especially valuable when specifying and verifying complex systems, such as distributed systems or multithreaded programs, because it allows one to prove invariants of the system as a whole on the level of the abstract model, then show that these invariants are preserved when the system is refined down its implementation-level components. Several recent developments of verified systems make use of refinement [Appel et al. 2017; Hawblitzel et al. 2015a; Klein et al. 2009; Leroy 2006; Lorch et al. 2020].

We focus on *bottom-up refinement*, which connects a concrete implementation, written in a high-level programming language, to an abstract, mathematical model. With such an approach, the implementation can make use of features of modern programming languages, thus allowing for performant, maintainable code. By contrast, other refinement approaches either do not address the connection between the lowest-level abstract model and the implementation; or else use code extraction to generate executable code, which may lead to sub-optimal code.

In this paper, we present a novel methodology to prove that an implementation refines an abstract model given as a transition system. We defer a more thorough discussion of related work to Sec. 6, but to the best of our knowledge, our methodology is the first that enables *flexible* refinement proofs along *all* of the following dimensions:

- *Program structure*. We do not impose any particular implementation structure. Refinement reasoning is localised to code locations which correspond to steps in the abstract model. A refinement proof can thus be added to an existing implementation, without rewriting the implementation itself. We address both multi-threaded processes and distributed systems.
- *Automation and Scalability*. Our approach uses off-the-shelf, SMT-based deductive verifiers to discharge proof obligations. Verification of safety properties requires only modest annotation overhead. To ensure scalability, our approach is method- and thread-modular.

Authors' addresses: Aurel Bílý, ETH Zurich, Zurich, Switzerland, aurel.bily@inf.ethz.ch; João C. Pereira, ETH Zurich, Zurich, Switzerland, joao.pereira@inf.ethz.ch; Jan Schär, ETH Zurich, Zurich, Switzerland; Peter Müller, ETH Zurich, Zurich, Switzerland, peter.mueller@inf.ethz.ch.

- *Liveness properties.* Our approach addresses verification of liveness properties in a deductive verification setting. Proofs of liveness properties are constructed in ghost code integrated with the implementation and the safety part of the proof.
- *Integration and Reusability.* Our approach is implemented in a state-of-the-art general-purpose deductive verifier. The specifications we define for standard library methods are reusable and integrate with the existing functional safety specifications.

Approach. Our methodology embeds the abstract model to be refined into the implementation as ghost state, in the form of a transition system. The implementation then interacts with this model using a *ghost lock*, which allows the program to *acquire* a view of the abstract state and to later *release* it, indicating which transition took place if the state was modified. Every release must then correspond to a legal transition. In other words, the ghost lock provides access to a two-state system invariant.

Each thread in a multi-threaded process, resp. each thread of every node of a distributed system, is given its own handle to the ghost lock. The ghost lock is erasable at compile time, thus it provides no synchronisation. As a result, operations within ghost lock critical sections must be linearisable.

To modularly reason about individual nodes of a system, we use *guards*, which describe how each component may affect the global system state. As in existing guarded transition system reasoning [Dinsdale-Young et al. 2017, 2010], this allows nodes to preserve local knowledge of the system across environment steps.

To prove liveness properties, we deeply embed LTL [Pnueli 1977] formulas in the assertion language. These are then connected to obligations [Boström and Müller 2015; Leino et al. 2010], abstract linear resources which must eventually be discharged by the holder, to prevent non-termination or lack of progress.

To integrate our methodology with real-world codebases, we instantiated it in Rust [Matsakis and Klock 2014], a programming language which provides strong guarantees about memory safety with its ownership type system. It is a systems programming language and allows interaction with low-level data types and operations to maximise performance. These factors make it well suited to implement distributed systems, which often have high correctness, performance, and scalability requirements. The ownership systems allow direct extraction of separation-logic-style assertions from well-typed programs [Astrauskas et al. 2019]. This reduces annotation overhead for proofs, facilitating further reasoning about functional correctness and making Rust an ideal target for deductive verification. The ownership system also facilitates refinement reasoning: for example, mutable references are known to not alias with other variables currently accessible by other threads, which means local data updates can be freely performed within ghost lock critical sections.

We evaluated our approach by implementing and verifying a simplified version of the Memcached caching system [Fitzpatrick 2004], as well as several smaller case studies. Our case studies and tool implementation will be submitted as an artefact. The verification of Memcached has been posed as a challenge in the 2nd VerifyThis long-term challenge presented at ETAPS 2023 [Ernst and Weigl 2023]; we will submit our verified implementation there.

Contributions. We make the following technical contributions:

- We present a verification methodology for refinement proofs that offers more flexibility than prior work in terms of program structure, automation, and real-world integration ($\rightarrow$ Sec. 2).
- We present a novel reasoning technique for local, control-flow sensitive guarded invariants ($\rightarrow$ Sec. 2.4).

```rust
// assume connection to node B in socket
fn node_a(socket: &mut TcpStream) {
  let mut ctr = 0;
  loop {
    // send
    socket.send(ctr);
    // wait for response, with timeout
    match socket.recv_timeout() {
      Some((n, resp)) if ctr == n =>
        { ctr += 1; }
      None => {}
    }
  }
}
```

```rust
// assume connection to node A in socket
fn node_b(socket: &mut TcpStream) {
  loop {
    // receive (block)
    let num = socket.recv();
    // compute
    let resp = do_something(num);
    // respond
    socket.send((num, resp));
  }
}
```

Fig. 1. Implementation of node A (left) and node B (right). The highlighted expressions are I/O operations discussed later in the text. For simplicity, we omit the (de)serialisation of transmitted data.

- We present a novel connection of LTL formulas to obligations, enabling the proof of both safety and liveness properties to be integrated into the executable code (⟶ Sec. 3).
- We implement our methodology in a state-of-the-art Rust verifier, resulting in strong automation, and the ability to reuse general-purpose verification techniques for parts of the proof unrelated to refinement (⟶ Sec. 4).
- We evaluate our methodology on a number of case studies, demonstrating its expressiveness and ability to adapt to an evolving codebase (⟶ Sec. 5).

2 METHODOLOGY: SAFETY

In this section, we show how to prove safety properties using our approach on a running example. The example consists of two nodes interacting over a network. Node A maintains a counter and sends consecutive numbers to node B. In turn, node B performs some (presumably expensive) computation on the received number, then sends a message back, containing both the original request and the computed response. A simplified implementation of both nodes in Rust is shown in Fig. 1. Here, we assume that the communication channel has already been set up, and that the nodes will keep communicating indefinitely. Due to the unpredictable nature of communication over a network, our implementation may exhibit different behaviours, including the following:

- The messages from node A to B may be successfully delivered, and the responses may be computed and successfully delivered to A before it times out.
- The communication channels may lose messages. Node A will repeatedly send the same number until node B responds.
- Node A may time out and repeat its request before node B computes the response. Node B does not keep track of which requests it has already replied to, and may process the same message multiple times. In turn, node A may receive the same message multiple times. In that case, node A ignores stale messages.

Regardless of which communication behaviours are observed during the execution of the system, the messages received (and responded to) by node B will always have a value less than or equal to node A's counter. This is an invariant of the system and we show how to prove it in Sec. 2.5.

2.1 Model definition

In our methodology, we specify programs with transition systems directly in Rust. Because of this, we are able to prove refinement in a general-purpose Rust verifier, without additional formalisms or tooling[1]. In general, a transition system consists of a set of states, a set of action labels, a set of initial states, and a relation that describes the valid transitions from any state, given an action label. In this section, we demonstrate how to define these components for our runnning example in Rust.

First, we define the set of states of the abstract model as a **struct** SystemState containing the current value of node A's counter (field a_ctr), the message currently being processed by node B (b_work), as well as the state of the channels in either direction (a_to_b and b_to_a). Next, we define the action labels as an **enum** Action containing a variant per kind of I/O operation performed by the system. Variant BSend has an integer argument, which represents the response chosen by the implementation of node B. The model does not constrain this value, which means it is existentially quantified in the specification; the value in BSend can be seen as the existential witness. The behaviour of the other actions is fully defined by the system state, as we will see shortly. The ALoss and BLoss actions model the behaviour of the *environment*, accounting for message losses.

```rust
struct SystemState {
  a_ctr: i32,
  b_work: Option<i32>,
  a_to_b: Seq<i32>,
  b_to_a: Seq<(i32, i32)>,
}
```

```rust
enum Action {
  ASend,      ARecv,
  BSend(i32), BRecv,
  ALoss,      BLoss,
}
```

The usual init and next predicates, here as Rust functions, define valid initial states and transitions:

```rust
fn init(state: SystemState) -> bool {
  state.a_ctr == 0 Ⓐ
  && state.b_work == None Ⓑ
  && state.a_to_b == Seq::empty() && state.b_to_a == Seq::empty() Ⓒ
}
```

The init predicate requires Ⓐ that the counter is initialised to zero, Ⓑ that node B is initially not performing any computation, and Ⓒ that both channels are empty.

```rust
fn next(p: SystemState, s: SystemState, a: Action) -> bool {
  match a {
    Action::ASend => s == SystemState {
      a_to_b: p.a_to_b.append(p.a_ctr), Ⓓ
      ..p }, // functional update syntax, uses p for all other fields
    Action::ARecv => p.b_to_a.len() > 0 Ⓔ
      && s == SystemState {
        b_to_a: p.b_to_a.tail(),
        a_ctr: p.a_ctr.max(p.b_to_a.head().0 + 1),
        ..p },
    Action::ALoss => p.b_to_a.len() > 0
      && s == SystemState {
        b_to_a: p.b_to_a.tail(),
        ..p },
    // ...
  }
```

[1]To define the behaviour of a system abstractly, it is common to use a formalism such as TLA$^+$, where the model is defined as a transition system. For reference, we also provide a TLA$^+$ specification for the example from Fig. 1 in Appendix A. A mapping from our representation to TLA$^+$ is discussed in Appendix C.

```
}
```

The next predicate[2] is parameterised by the previous state p, the next state s, and the action a taken. The action ASend updates the network state by Ⓓ adding the current counter value to the outgoing channel. This action is always enabled, thus there are no constraints on p, unlike ARecv, which Ⓔ requires at least one message in the incoming channel.

All definitions provided in this section are embedded in the implementation as *ghost code*, i.e., code that is added for specification purposes and which does not interfere with regular code. Thus, it can be safely erased without observable differences in the program outcome.

2.2 Updating the model state

The model state includes the abstract state of each node and that of the environment. The implementation interacts with the environment by calling methods that perform I/O, and which are provided by (unverified) libraries such as the Rust standard library. In Fig. 1, there are calls to three such methods: send in node A (Line 6) and in node B (Line 23), recv_timeout in node A (Line 8), and recv in node B (Line 19). To describe the effect of these calls on the model state, we attach specifications, in the form of *pre-* and *postconditions* to the corresponding methods[3], as shown below for the methods send and recv:

```
#[ensures(c == old(c).append(v))]
fn send<T>(v: T, c: &mut Seq<T>);
```

```
#[ensures(!old(c).is_empty())]
#[ensures(c == old(c).tail()
  && result == old(c).head())]
fn recv<T>(c: &mut Seq<T>) -> T;
```

In both methods, we expect a sequence denoting the state of the channel to be passed as an additional, ghost argument. This argument is used in the postcondition of both methods to describe the effects of the method on the channel: send appends the message c to the sequence denoting the channel, whereas recv removes the first element of the sequence and returns it. Since it is a mutable reference, the verifier considers the value it points to to be modified to an arbitrary value that satisfies the postcondition. Note that these specifications do not require the correct channel to be passed, e.g., node A may call send and mistakingly pass the state of the buffer from node B to A; in Sec. 2.3, we show an improved specification where the correspondence is checked by the verifier.

The responsibility to update the non-environment fields of the abstract model lies with the implementation. For example, node A must update a_ctr according to the transitions it performs. The second part of the loop in node A therefore performs a ghost update:

```
let state = /* ... */; // mutable reference to the system state
match socket.recv_timeout(&mut state.b_to_a) {
  Some((n, resp)) if ctr == n => {
    ctr += 1;
    state.a_ctr += 1; // ghost update
  }
  None => {}
}
```

At this point, node A could assert that it is indeed performing the expected ARecv transition by referring to a copy of the state before the action and after the action:

```
assert!(next(prev_state, state, Action::ARecv));
```

[2]The full listing is provided in Appendix B.

[3]Following the Prusti syntax, the attributes #[requires(..)] and #[ensures(..)] denote pre- and postconditions, respectively. The operator old(..) refers to the value of the given variable in the initial state of the method call, and result refers to the value returned by the call.

2.3 Ghost lock

The model state abstracts over the global state of the system, including the environment. When we verify the implementation of a component of the system, we must maintain, in ghost code, a view of the global state as an instance of the model state; after all, methods that perform I/O may have their preconditions and postconditions defined in terms of the model state.

To guarantee *soundness*, i.e., that if the verifier succeeds, the implementation does indeed refine the abstract model, we must guarantee that the view of the system maintained by the implementation is always consistent with the state of all other nodes and with the state of the environment. In our methodology, we achieve this by *sharing a single instance of the model state among all nodes*. Akin to locks, which control access to shared resources in multi-threaded programs, *ghost locks* control access to the shared model state. Their interface is similar to regular locks, as shown below:

```rust
impl GhostLock {
  fn acquire(&mut self);
  fn release(&mut self, action: ActionKind);
  fn release_stutter(&mut self);
  fn locked(&self) -> bool;
  fn state(&mut self) -> &mut SystemState;
}
```

Like a regular lock, a ghost lock may be *acquired* to gain exclusive access to the (single instance of the) model state, and later *released*, giving up the exclusive access to the model state. Unlike a regular lock, a ghost lock can only be used in *ghost code*, thus calling its methods must not change the behaviour of the original program. As a consequence, ghost locks cannot be used for synchronisation. Moreover, we treat each critical section of a ghost lock as an *atomic* update of the model state (or a stuttering step); thus, no intermediate state of the system should be observable. To ensure that this reasoning is sound, we require all critical sections to be *linearisable* [Herlihy and Wing 1990]. In our implementation, we reason about linearisability of statements within critical sections using Lipton's reduction [Lipton 1975]. Finally, we impose that, at any moment in time, each node running in the system has at most a single instance of the ghost lock. This ensures that no node can maintain two incompatible views of the system, and use them to derive contradictions.

The `acquire` operation marks the beginning of a critical section. The `locked` method indicates whether the node is currently executing in a critical section, and, if so, the method `state` provides a mutable reference to the system state through which updates can be performed. The end of a critical section may be marked with a call to `release`. All calls to `release` are annotated with the action performed in the critical section. Firstly, this allows the programmer to communicate intent, i.e., that a particular action was intended and none other. Secondly, it makes it easier for automated verifiers to prove that the action took place, without relying on instantiating existential quantifiers, which would be needed for the `BSend` action, for example. Critical sections may also end with a call to `release_stutter`, to indicate that no transition took place. An explicit stutter release is useful as it avoids the need for a dedicated "stutter" action label. Since critical sections consist of paired calls rather than syntactic blocks, it is possible, for example, to conditionally release the ghost lock with different actions, depending on the results of operations within the critical section.

When a ghost lock is released, the node loses access to the shared state. The node may later re-acquire the lock to access the model state again; in the time between releasing and re-acquiring the lock, another node may have acquired it and modified the state of the system. As such, when a node acquires the ghost lock, the model state is *havocked*, i.e., the verifier assumes an arbitrary value for the abstract state. In Sec. 2.4, we show how to use invariants of the abstract model and *guards* to soundly preserve knowledge about the model state between different critical sections.

Having introduced ghost locks, we can now rewrite the specifications of the standard library methods shown in Sec. 2.2 such that the user no longer has to pass (maybe erroneously) the components of the model state in each call. Instead, these methods now receive a mutable reference to the *acquired* ghost lock as a ghost argument. The environment can be accessed through such a reference, allowing specifications on external methods to perform updates on the environment state. The specifications for `send` and `recv` which use the ghost lock could thus look as follows[4]:

```
#[requires(gl.locked())]
#[ensures(gl.locked())]
#[ensures(gl.state().a_to_b
  == old(gl.state().a_to_b)
    .append(v))]
fn send(v: i32, gl: &mut GhostLock);
```

```
#[requires(gl.locked())]
#[ensures(gl.locked())]
#[ensures(gl.state().b_to_a
    == old(gl.state().b_to_a).tail()
  && result
    == old(gl.state().b_to_a).head())]
fn recv(gl: &mut GhostLock) -> (i32, i32);
```

With specifications provided for I/O methods, the verifier checks that any I/O operation is indeed performed within a ghost lock critical section. The I/O operations receive the ghost lock as argument; their specifications define how the environment state, accessible via the acquired ghost lock, was updated. This requires that the I/O operations must occur within ghost lock critical sections, that the critical sections are annotated with the correct action label, and that the implementation also updates the non-environment ghost state accordingly. The entire refinement proof thus naturally develops from the proof obligations resulting from using I/O methods, since any implementation must use I/O methods to interact with any other node or the environment.

A first attempt to update the running example with ghost lock annotations may be as follows (showing only the loop bodies, and assuming `gl` is a mutable reference to the ghost lock):

```
// send
gl.acquire();
socket.send(ctr, gl);
gl.release(Action::ASend); Ⓒ

// wait for response, with timeout
gl.acquire();
match socket.recv_timeout() {
  Some((n, resp)) => {
    if ctr == n {
      ctr += 1;
      gl.state().a_ctr += 1;
    }
    gl.release(Action::ARecv); Ⓓ
  }
  None => { gl.release_stutter(); Ⓐ }
}
```

```
// receive
gl.acquire();
let num = socket.recv(gl);
gl.state().b_work = Some(num);
gl.release(Action::BRecv); Ⓔ

// compute
let resp = do_something(num);

// respond
gl.acquire();
socket.send((num, resp), gl);
gl.state().b_work = None;
gl.release(Action::BSend(resp)); Ⓑ
```

The environment state accessible through the acquired ghost lock is updated according to the specifications of I/O methods, by passing the ghost lock as a ghost argument. Non-environment ghost state updates are performed by the implementation; our methodology ensures that all updates to the model state are linearisable, and that, when the lock is released, the system state was correctly updated according to the label passed to the `release` method.

[4]These specifications are only suitable for node A; in our implementation there is a further mapping layer that allows different channels in the system to be treated uniformly without *ad-hoc* specifications for each node or channel. We do not describe the layer in this paper, but the high-level idea is that the abstract model of a program is *projected* to other transition systems related to particular features, e.g., the I/O channels; these are then linked to the main model based on a shared abstraction of the feature, e.g., sequences for I/O channels.

When node A does not receive a response from node B within some pre-determined timeout Ⓐ, the ghost lock is released with a *stutter step* to indicate that no change occurred in the system state. When releasing Ⓑ the ghost lock for the `BSend` action, the sent value is provided as an argument.

2.4 Preserving local knowledge using guards

In the previous subsection, we show how to use ghost locks to ensure that all nodes have a consistent view of model state. Moreover, we annotate the running example with a ghost lock, from which a node may obtain permission to update the model state. Trying to verify the running example as is fails though, as the verifier is not able to prove that each critical section ending on a call to `release` corresponds to a valid step of the system. A closer look reveals the following problems:

- Ⓒ For the `ASend` action to be valid, the sent value (`ctr`) must be equal to the value of the counter in the abstract state (`a_ctr`). However, the verifier conservatively assumes that there are other nodes which may modify `a_ctr`.
- Ⓓ The same problem occurs for the `ARecv` action, but there is a further complication: the update of `ctr` and `a_ctr` only follows the specification if node B is responding to a number less than or equal to the current `ctr` value.
- Ⓑ Ⓔ Node B cannot safely assume that node A does not update the field `b_work`.

At the root of these problems is the fact that, up to this point, we treated all nodes in the system as equal. This need not be true; in fact, complex systems are often made up of heterogeneous nodes, each of which may play a different role in the system, thus it can be expected that they perform different kinds of actions. Crucially, not every node may perform every transition.

To account for the heterogeneous nature of distributed systems, we use *guards*, a common solution to reason about shared-state concurrency [Dinsdale-Young et al. 2017, 2010]. Guards are affine resources owned by nodes, i.e., every guard has exactly one owner (an executing node or the environment), or it has no owner (the guard cannot be used anymore). Guards represent the permission to perform certain actions in the system. Ownership of a guard thus places an upper bound on what the environment might do, such that we can preserve information about the system state that is stable under environment interference.

In our example, we define three kinds of guards, for the two nodes and the environment, using a Rust **`enum`**[5] and a function which determines whether a guard is needed for an action:

```
enum GuardKind {
  NodeA,
  NodeB,
  Environment,
}
```

```
fn guard_needed(a: Action, g: GuardKind) -> bool {
  match a {
    Action::ASend | Action::ARecv => g == GuardKind::NodeA,
    Action::BSend(_) | Action::BRecv => g == GuardKind::NodeB,
    Action::ALoss | Action::BLoss => g == GuardKind::Environment,
  }
}
```

This definition states that guard `NodeA` is required for the `ASend` and `ARecv` actions, for example. Although in this system, there are only actions which require a single guard, in our case studies we also found situations where multiple guards are required to perform a single action. In such cases, the `guard_needed` function would contain a disjunction of the needed guard kinds for the action. The advantage over a single guard is that multiple guards can be (temporarily) owned by different nodes, as long as it is not necessary to perform actions which require their combination.

[5]As with actions, the variants of this **`enum`** may contain data. Therefore, there can be unboundedly many guards in a system. This is useful, for example, in server applications, where each client-handling thread owns one guard instance.

All guards of the system are created when the model is initialised and stored in a *guard dispenser*. The verifier checks that each guard is only dispensed once. However, the affine nature of guards is already naturally represented and proven by the Rust type system: each guard is an owned value and using a guard is considered a mutating operation. Since safe Rust guarantees that there is only one mutable reference to any given value, only one node is able to use any guard at a time. The main operations of guards accessible within Rust are as follows:

```rust
impl Guard {
  fn kind(&self) -> GuardKind;
  fn open<F: Fn(SystemState) -> bool>(&mut self, gl: &GhostLock, pred: F);
  fn last_state(&self) -> SystemState;
}
```

`kind` returns the kind of the guard, as defined earlier. Within a ghost lock critical section, the method `open` can be used to *open* the guard for the duration of that critical section, i.e., to show that the executing node indeed mutably owns the guard at this point. A reference to the ghost lock is passed at the same time, allowing the verifier to check that the ghost lock is currently acquired. Any opened guard is closed when the critical section ends. The `open` method allows the node to perform actions protected by this guard.

Often, when opening a guard, it is useful to learn additional facts about the abstract state that can be justified by owning that very guard. In other words, there are invariants that hold in the system as long as transitions protected by the given guard (or set of guards) are *not* taken. As an example, suppose node A owns the guard `NodeA`. Given the definition of `guard_needed`, this means node B cannot perform the `ARecv` action. Consequently, if node A knows that the value of `a_ctr` is, for example, zero at the time it releases the ghost lock, then it must remain zero in the state when it subsequently re-acquires the ghost lock, because only the `ARecv` action modifies the value of `a_ctr`. Such a claim is expressed in ghost code via the second argument of `open`. In the loop of node A, the local variable `ctr` and the ghost variable `a_ctr` should remain in sync. By the same line of reasoning as outlined above, the predicate expressing this equality is accepted by the verifier[6]:

```rust
// send
gl.acquire();
guard.open(gl, |state| state.a_ctr == ctr); // ctr matches a_ctr
socket.send(ctr, gl); // this operation thus sends the correct value (i.e. a_ctr) to node B
gl.release(Action::ASend);
```

The verifier checks, by induction, that the given predicate holds when the guard is opened:

(1) *Base case*: the state in which the guard was last closed must satisfy the predicate. To this end, a copy of the state is recorded for each guard and accessed using the `last_state` function.
(2) *Inductive case*: given two consecutive states related by an action which does *not* require the guard, if the predicate holds in the first state, then it must hold in the second state.

The reasoning principle is similar to VCC *claims* [Cohen et al. 2010], although unlike VCC claims, the guard does not fix a particular invariant – the user may open the guard with arbitrary invariants. In our approach we record the *state* in which the guard was last closed and defer checking the invariant until it is actually needed by the implementation. As a result, the implementation is free to choose a different predicate every time the guard is opened, where the predicate may make use of node-local data, or differ depending on the control flow taken.

[6]In this paper we overload Rust closure syntax for predicates. Such closures cannot cause side-effects (otherwise, it is rejected by the type-checker), but may capture copies of the local state. To map such syntax to logical assertions, the captured variables are seen as the free variables of a formula, instantiated with some values at the point the `open_with_predicate` call takes place.

In our methodology, guards may change owners. For example, a guard may be placed inside a lock. For this to be useful, the lock invariant must restrict the value of `last_state` in some way; otherwise, opening the guard with any non-trivial predicate would fail.

The `Environment` guard encompasses the permissions of the environment to perform implementation-unrelated actions. In this example, this includes losing messages on either of the two communication channels (the `ALoss` and `BLoss` actions). This guard must not be given to any node (we forbid the construction of such a guard). Although defining an environment guard explicitly makes the model clearer, individual node implementations need not make any special assumptions due to its existence: it is sound to simply assume that any guard that the current node does *not* hold may be used by another part of the system.

2.5 Model invariants

In this section, we explain how invariants of the model may be made available to the implementation. As stated in Sec. 2, the messages received (and responded to) by node B will always have a value less than or equal to node A's counter. In the absence of integer overflow, this property seems intuitive: node B cannot respond to a request it has not received yet. Although this property is not explicitly stated in the abstract model, it follows from the specification. If our specification was obtained from a TLA$^+$ module, such a property could in principle instance be proved (for instance, using the TLA$^+$ *Proof System* [Chaudhuri et al. 2010]) and simply assumed by our implementation. However, we instead justify the property within the verifier itself, which has the benefit that no additional tool or formalism is required. The proof can be decomposed into the following steps, tracing the flow of values from node A to node B and then back again:

(1) `a_ctr` is monotonically increasing.
(2) Numbers in `a_to_b` are at most `a_ctr`.
(3) If `b_work` is not `None`, the value it wraps is at most `a_ctr`.
(4) Numbers in the first component of `b_to_a` are at most `a_ctr`.
(5) The first component of all entries in `b_to_a` is at most `a_ctr`.

Step (1) follows from the definition of `next`. Steps (2)–(4) are justified inductively: the property holds in the initial state and is preserved by any valid transition. Step (5) is a concrete application of step (4) for the callsite in node A. Within the verifier, each step is expressed as a lemma method [Jacobs et al. 2010] whose specification expresses the property of interest, and whose body provides the proof for that lemma. Using the theorem then corresponds to calling the lemma method.

3 METHODOLOGY: LIVENESS

Up to this point, we have focused on the safety properties of the system: properties which can be proven to be preserved by each individual step of the system, i.e., when releasing the ghost lock. Verifying *liveness properties*, on the other hand, poses multiple challenges. First, liveness properties concern infinite traces, so it is generally not possible to show that the property holds when performing a single ghost lock step. Second, in the case of reactivity properties[7], there may never be a point at which the property is definitely satisfied; instead, it may be possible only to show that progress towards the property was made. Finally, it may not be possible to check the property locally in one node.

In our example, some liveness properties of interest include:

(1) Node A keeps sending requests to node B indefinitely.
(2) Node B will eventually respond to any request from node A.

[7]Using terminology from Manna and Pnueli [1991], reactivity properties are conjunctions of formulas of the form $\Box\Diamond p \lor \Diamond\Box q$, when expressed in LTL.

(3) Every number will eventually be requested and responded to.

In the following sections, we describe how LTL formulas are used to represent liveness properties in our methodology (→ Sec. 3.1); we introduce a mechanism to verify that the properties are *eventually* satisfied (→ Sec. 3.2); we discuss how the verifier is guided to prove liveness properties of a system (→ Sec. 3.3); and we discuss how fairness assumptions, such as ones about network behaviour, can be embedded into the proof (→ Sec. 3.4).

3.1 LTL formulas

A common formalism to express liveness properties is linear temporal logic (LTL) [Pnueli 1977]. It is a logic where temporal properties are expressed using first-order logic combined with temporal operators, such as *always* (□) or *eventually* (◊). In the model of abstract transition systems, *always* is a universal quantification over all the subsequent states reached in the current computation. *Eventually* is the existential counterpart. In this paper, we use the following fragment of LTL:

$$
\begin{array}{lll}
\phi ::= \phi \land \phi \mid \phi \lor \phi \mid \phi \rightarrow \phi & & \text{standard logical connectives} \\
\quad \mid \Box\phi \mid \Diamond\phi \mid \phi \Rightarrow \phi & & \text{temporal operators} \\
\quad \mid \bot \mid \top \mid S \mid A & & \text{atoms} \\
\quad \mid \forall v.\, \phi \mid \exists v.\, \phi & & \text{value quantifiers}
\end{array}
$$

In the above, S represents an arbitrary *state formula*, a predicate over a single state of the abstract model. A represents an arbitrary *action formula*, a two-state predicate over consecutive states of the abstract model. We will use action labels, such as `ARecv`, as action formulas. The *temporal entailment*, $\phi_1 \Rightarrow \phi_2$ represents $\Box(\phi_1 \rightarrow \phi_2)$. In addition to the temporal operators (which quantify over states) we also allow quantification over *values* (not connected to any particular state), which can bind free variables appearing in state formulas and action formulas.

The three liveness properties of our system can thus be represented as:

(1) $\Box\Diamond$`ASend`
(2) $\Box\Diamond(\exists r.$ `BSend(r)`$)$
(3) $\forall i.\, \Diamond(\exists r.$ `b_to_a' == b_to_a.append((i, r))` $\land$ `BSend(r)`$)$

`ASend` and `BSend(r)` are action formulas corresponding to actions of the model. `b_to_a' == b_to_a .append((i, r))` is also an action formula, where `b_to_a'` refers to state of `b_to_a` *after* the step. The formula has the free variables `i` and `r`, each of which is bound by one of the quantifiers.

In our implementation, we use a deep embedding of LTL to specify liveness properties. Although we defer a more detailed discussion of this embedding until Sec. 4.1, we will now describe how LTL reasoning relates to the rest of our approach. Throughout the rest of this section, we will use a blue background to denote LTL formulas, to avoid the syntactic overhead of the deep embedding.

3.2 Obligations

Obligations [Bizjak et al. 2019; Boström and Müller 2015; Hamin and Jacobs 2019; Leino et al. 2010], are resources that must be explicitly discharged. Failure to discharge them, either by terminating the execution without calling an appropriate method to discharge the obligation or by never terminating are rejected by the proof system. In the previous subsection, we showed how we represent the liveness properties to be verified as LTL formulas. In this subsection, we discuss how such formulas can be associated with obligations to guarantee that these properties are satisfied.

Every obligation must eventually be discharged by the node that holds it, i.e., there are finitely many steps before the obligation is fulfilled. To this end, each obligation is associated with a termination measure, a standard technique to turn liveness (termination in particular [Floyd 1993])

$$\text{L.DISCHARGE}\ \frac{\phi(\texttt{i})}{\{\texttt{show\_at}(\phi,\ \texttt{i})\}\ \texttt{discharge()}\ \{\}} \qquad \text{L.STR}\ \frac{\phi_1(\texttt{i}) \Rightarrow \phi_2(\texttt{j})}{\{\texttt{show\_at}(\phi_2,\ \texttt{j})\}\ \texttt{strengthen()}\ \{\texttt{show\_at}(\phi_1,\ \texttt{i})\}}$$

$$\text{L.SPLIT}\ \frac{}{\{\texttt{show\_at}(\phi_1 \wedge \phi_2,\ \texttt{i})\}\ \texttt{split()}\ \{\texttt{show\_at}(\phi_1,\ \texttt{i}) * \texttt{show\_at}(\phi_2,\ \texttt{i})\}}$$

$$\text{L.QSPLIT}\ \frac{P(v)}{\{\texttt{show\_at}(\forall i.\, P(i) \Rightarrow A(i),\ \texttt{i})\ \}\ \texttt{qsplit()}\ \{\texttt{show\_at}(A(v),\ \texttt{i}) * \texttt{show\_at}(\forall i.\, i \neq v \wedge P(i) \Rightarrow A(i),\ \texttt{i})\}}$$

$$\text{L.QEMPTY}\ \frac{\forall i.\, \neg P(i)}{\{\texttt{show\_at}(\forall i.\, P(i) \Rightarrow A(i),\ \texttt{i})\ \}\ \texttt{qempty()}\ \{\}}$$

Fig. 2. Rules of the LTL proof system related to obligations and universal quantifier reasoning. L.DISCHARGE discharges an LTL obligation if the formula holds. L.STR strengthens an LTL formula to be shown. L.SPLIT splits an obligation to show a conjunction of two formulas into two obligations to show the conjuncts. L.QSPLIT splits off a conjunct off a quantified formula. Note that we impose an additional restriction on the order of conjuncts in the quantifier, not shown here. L.QEMPTY discharges a quantifier which holds vacuously.

into safety properties, amenable to automated verification. The measure is a function mapping obligations to a value in a well-founded set. Any method call or loop iteration entered with an obligation held must decrease the termination measure value for that obligation. Because the set is well-founded, there is no way to indefinitely delay discharging the obligation.

Specifications, such as method preconditions or loop invariants, may refer to obligations. Such an assertion is a resource assertion in separation logic. For example, an obligation in a method precondition indicates that a call to the method consumes the obligation. A call to the method from a scope where the obligation is not held violates the precondition.

An example of obligations in our methodology is a ghost lock release obligation. This obligation appears in the postcondition of the ghost lock acquire method, thus indicating that the node acquiring the ghost lock must release it. Symmetrically, the obligation also appears in the precondition of the ghost lock release method, but *not* in its postcondition, indicating that the obligation is discharged by a call.

```rust
impl GhostLock {
  #[ensures(gl_release())]
  fn acquire(&mut self);
}
```

```rust
impl GhostLock {
  #[requires(gl_release())]
  fn release(&mut self, action: ActionKind);
}
```

3.3 LTL proof rules

To prove that the implementation satisfies a given liveness property, we associate the LTL formula representing that property with an obligation given to the implementation. The obligation is discharged when it can be shown that the associated LTL formula holds. The truth value of an LTL formula depends on the current state and the subsequent states in the trace. As an example, the action formula ASend is true for consecutive states of the trace, related by a ASend transition.

With a global view of the trace, states can be indexed with natural numbers. $\phi(n)$ then refers to the truth value of the LTL formula ϕ in the n-th state of the trace. Although individual nodes do not have a global view of the trace (i.e., the intermediate states visited during an environment step are not known), successive ghost lock steps within the same node always correspond to states further in the trace. Consequently, we omit the *next* operator in our LTL fragment, as it is not useful when reasoning about a single node in a distributed system: any environment step performs zero or more transitions in the general case, invalidating any local knowledge about precise step counts. We will refer to the obligation to show ϕ(i), i.e., that ϕ holds at trace index i as show_at(ϕ, i).

The atoms of LTL formulas over abstract states specify the state changes performed in critical sections of the ghost lock, which are the only changes of the abstract state. The method id(), accessible on an acquired ghost lock, returns a natural number corresponding to the current index in the trace. We define state formulas as predicates over the abstract state at the point of release, and action formulas as two-state predicates over the states at the points of acquire and release.

We provide the rules of an LTL proof system as a library of ghost methods. A selection of these rules is shown in Fig. 2. The implementation can then use calls to ghost methods to manipulate LTL obligations, thus proving the LTL formulas. As an example, we consider the first property, □◊ASend, and show it holds in the implementation of node A:

```
let mut show_from = 0; Ⓐ
loop {
  invariant!(show_at(□◊ASend, show_from));
  gl.acquire();
  let id = gl.id(); // id >= show_from Ⓑ
  // obligations: show_at(□◊ASend, show_from)
  strengthen(); Ⓒ
  // obligations: show_at(◊ASend, show_from) * show_at(□◊ASend, show_from + 1)
  strengthen(); Ⓓ
  // obligations: show_at(ASend, id) * show_at(□◊ASend, show_from + 1)
  strengthen(); Ⓔ
  // obligations: show_at(ASend, id) * show_at(□◊ASend, id + 1)
  guard.open(gl, |state| state.a_ctr == ctr);
  socket.send(ctr, gl);
  gl.release(Action::ASend); // thus ASend(id) is true
  discharge(); Ⓕ
  // obligations: show_at(□◊ASend, id + 1)
  show_from = id; Ⓖ
  // obligations: show_at(□◊ASend, show_from + 1)
  // (remainder of the loop implementation)
}
```

In the above, we Ⓐ introduce the ghost variable show_from to keep track of progress in the LTL property, and then use show_from in the obligation in the loop invariant. Ⓑ This variable is a lower bound on the index of any ghost lock critical section appearing in this loop, i.e., id is greater or equal to show_from.

We use three calls to the strengthen ghost method to rewrite the obligations into more concrete, stronger goals[8]. First, Ⓒ we split the *always* operator into its first state (at index show_from) and the rest of the states. Second, Ⓓ , we concretise the *eventually* operator, since we will perform the ASend action in the current ghost lock step. Finally, Ⓔ , we push the second obligation to a later state to fit the shape of the loop invariant at the end of the loop.

After the ghost lock step, Ⓕ the action formula is known to hold at this index, thus the first obligation can be discharged. Ⓖ After updating the show_from variable, the remaining obligation is consumed by the loop invariant, and the proof is completed.

Quantifiers with an infinite domain are useful to express properties of non-terminating programs. The third property of our system becomes the loop invariant of node B:

```
let mut sent = 0; Ⓗ
loop {
```

[8]Once again, to avoid the syntactic complexity of the deep embedding, we omit the arguments to these calls and only present the obligations in comments.

```
    invariant!(show_at(∀i.i >= sent ⇒ ◊(∃r.b_to_a' == b_to_a.append((i, r)) ∧ BSend(r)), 0));
    // receive
    gl.acquire();
    guard.open(gl, |state| state.b_work == None);
    let num = socket.recv(gl);
    gl.state.b_work = Some(num);
    gl.release(Action::BRecv);
    // compute
    let resp = do_something(num);
    // respond
    gl.acquire();
    let id = gl.id();
    guard.open(gl, |state| state.b_work == Some(num));
    socket.send((num, resp), gl);
    gl.state.b_work = None;
    gl.release(Action::BSend(resp));
    // obligations: show_at(∀i.i >= sent ⇒ ◊(∃r.b_to_a' == b_to_a.append((i, r)) ∧ BSend(r)), 0)
    if num > sent {
      qsplit();    Ⓘ
      // obligations: show_at(◊(∃r.b_to_a' == b_to_a.append((num, r)) ∧ BSend(r)), 0)
      //   * show_at(∀i.i >= sent + 1 ⇒ ◊(∃r.b_to_a' == b_to_a.append((i, r)) ∧ BSend(r)), 0)
      strengthen(); Ⓙ
      // obligations: show_at(b_to_a' == b_to_a.append((num, resp)) ∧ BSend(resp), id)
      //   * show_at(∀i.i >= sent + 1 ⇒ ◊(∃r.b_to_a' == b_to_a.append((i, r)) ∧ BSend(r)), 0)
      discharge(); Ⓚ
      sent += 1;
      // obligations: show_at(∀i.i >= sent ⇒ ◊(∃r.b_to_a' == b_to_a.append((i, r)) ∧ BSend(r)), 0)
    } else { Ⓛ }
  }
```

In the above, we Ⓗ once again introduce a ghost variable, this time to track the domain of the quantifier in the LTL formula. Once node B has received a number, computed the response, and sent it to node A, we check if the number has not been sent by node B before[9]. If so, Ⓘ we split the first conjunct off the quantifier. Ⓙ We concretise the *eventually* operator and pick a witness for the existential quantifier, allowing us to Ⓚ discharge the conjunct. After updating the ghost variable, the obligation matches the one in the loop invariant again.

In the case that the number has been seen by node B before (branch Ⓛ), we must still show that progress is made towards the LTL property overall. We omit the details in this paper, but our proof system contains rules for discharging progress based on the guaranteed behaviour of other nodes.

The use of ghost methods to describe proof steps allows the full expressiveness of (our fragment of) LTL, at the cost of some automation. As an example, the verifier will automatically infer that $\phi_1 \land \phi_2$ holds when ϕ_1 and ϕ_2 both hold, but it will not find witnesses for existential quantifiers.

3.4 Fairness assumptions

Some of the liveness properties of our system can be justified only if the network is not faulty. If the channel from A to B is faulty and always loses messages, node A may keep sending requests (and thus satisfy the first property), but the system overall does not progress (the third property). Similarly, if the channel from B to A always loses messages, node may B keep responding to requests (the second property), but once again, the system overall does not progress.

[9]The entire condition and obligation manipulation is ghost code, thus this state need not be stored by the actual implementation.

Although the network is part of the model, it is not part of our implementation, so we express the expected behaviour as assumptions. We prove liveness properties under the *strong fairness assumption* that any message repeatedly sent to a channel will eventually be delivered, which we can represent as the formulas `◊□(b_to_a.len() == 0) ∨ □◊ARecv` and `◊□(a_to_b.len() == 0) ∨ □◊BRecv`. These formulas are assumed when calling network I/O methods, to justify later LTL proof steps. In the example of node B above, the second formula, combined with the first property of the system (`□◊ASend`), justifies why the loop in node B is guaranteed to see every number at some point.

There are other, lower-level assumptions, for instance, that the underlying execution environment and thread scheduler work correctly. These assumptions are built into our verification technique for concurrent programs and, thus, not explicit in the specifications.

4 IMPLEMENTATION

We implemented our approach in the state-of-the-art deductive Rust verifier, Prusti [Astrauskas et al. 2019], based on Viper [Müller et al. 2016], a framework for automated separation logic reasoning. To support our methodology, we had to extend Prusti with support for obligations. Our methodology is not inherently tied to Prusti-specific reasoning, thus it should be implementable in other Rust verifiers.

4.1 Deep embedding of LTL

In this section we briefly describe our embedding of LTL into Rust with Prusti annotations. LTL formulas are encoded as types which implement the `Ltl` trait. This trait is parameterised by the abstract model (described in Sec. 4.2), and has one associated function[10]:

```
trait Ltl<M: Model> {
  #[pure] fn holds(&self, gl_id: Nat) -> bool;
}
```

The `holds` function defines whether the current LTL formula holds in the given state, which is identified by the trace index of the ghost lock critical section (discussed in Sec. 3.3).

The obligations are then generic over types of this trait:

```
obligation! { fn show_at<M: Model, L: Ltl<M>>(l: L, id: Nat); }
```

Discharging the obligation is accomplished by calling a method which consumes the obligation, but only if the precondition ϕ`.holds(gl_id)` is satisfied.

Atoms in LTL consist of terminal formulas, such as `True` or `False`, which are implementations of the `Ltl` trait with a trivial `holds` implementation; one-state predicates on the abstract model's state; and two-state predicates on the state, indicating that some transition has happened. The user can invoke a function-like macro to declare a type which represents a state formula, which internally implements the `Ltl` trait with a suitable specification on the `holds` function.

Composite LTL formulas are types which combine other LTL formulas. Such combinators are generic over the sub-formulas, as can be seen in the definition of conjunction:

```
struct Conj<M: Model, L1: Ltl<M>, L2: Ltl<M>>(L1, L2);
```

As a result, the conjunction $\phi_1 \land \phi_2$ is represented by a different type than the conjunction $\phi_2 \land \phi_1$.

The library also provides a number of methods to manipulate LTL formulas. In particular, a rewriting system allows exchanging the obligation to show an LTL formula for the obligation to show a stronger LTL formula, which corresponds to the L.Str rule. Because the shape of LTL

[10]The Prusti annotation `#[pure]` indicates that a function is deterministic, side-effect-free, and terminating. Pure functions can be used within specifications of other functions and ghost code.

Sec.	Program		Liveness	$\#_C$	$\#_M$	$\#_P$	VT (s)
5.1	Memcached [Fitzpatrick 2004]	v1	✓	117	225	286	334.7
		v2	✓	118	225	290	354.6
		v3	✓	181	235	377	379.7
		TCB		320	–	57	–
5.2	Prod./cons. queue [Hance 2022]	SeqCst	✓	125	169	247	392.8
		AcqRel		125	169	428	654.5
5.3	Paxos [Lamport 1998]			105	111	205	217.7
5.3	Lock-free hash set [Kuppe 2017]			54	134	361	280.6

Fig. 3. Evaluation results. $\#_C$ is the number of lines of code in the implementation, i.e., non-ghost code that is required for the implementation to work. We exclude empty lines, comments, import statements. $\#_M$ is number of lines in the model definition, including the `Model` trait implementation and any associated types. $\#_P$ is the number of lines of specifications and ghost code operations. For lines with a checkmark, $\#_P$ also includes the annotations required for the liveness proof. The verification time (**VT**) is the 10% Winsorised mean of the wall-clock runtime across 10 verification runs using Prusti commit 79d48686, measured on an Intel Core i9-10885H 2.40GHz CPU with 16 GiB of RAM.

formulas is fully expressed in the type, such rewriting is often based purely on types, though sometimes additional preconditions are added.

4.2 Model trait

The mapping of our methodology to a Rust library makes heavy use of the trait system. The `Model` trait defines an abstract model:

```
trait Model {
  type AbsState: Copy;      Ⓐ type Action: Copy;   Ⓑ type GuardKind: Copy; Ⓒ
  type Liveness: Ltl<Self>; Ⓓ type EnvState: Copy; Ⓔ
  #[pure] fn get_env_state(s: Self::AbsState) -> Self::EnvState; Ⓕ
  #[pure] fn init(s: Self::AbsState) -> bool;
  #[pure] fn next(s: Self::AbsState, n: Self::AbsState, a: Self::Action) -> bool;
  #[pure] fn guard_needed(a: Self::Action, g: Self::GuardKind) -> bool;
  #[pure] fn init_ltl(s: Self::AbsState) -> Self::Liveness;
}
fn new<M: Model>() -> (GhostLock<M>, GuardDispenser<M>, Ltl<M>) { .. } Ⓖ
```

Any implementation of this trait corresponds to an abstract model. The implementation declares Ⓐ the type of full system states, Ⓑ the type of labelled transitions, Ⓒ the type of guard kinds, and Ⓓ the shape of the LTL formula representing the liveness property.

The associated type `EnvState` Ⓔ and the required method `get_env_state` Ⓕ together define a subset of the full system state that represents the state of the environment. This part of the state can only be modified by calling trusted methods, such as methods of the standard library. The remainder of the required methods maps naturally to methods already explained in Sec. 2.

Finally, the provided `new` method Ⓖ initialises an instance of the ghost lock, a *guard dispenser* with all the guards of the system, and the liveness property. The lock can be used as described before, while the dispenser can be used to obtain specific guards.

5 EVALUATION

To evaluate our approach, we implemented a number of concurrent and distributed systems from the literature, then specified and verified them. The results of our evaluation are shown in Fig. 3.

Before we discuss each case study in detail, we summarise the overall findings. The general annotation overhead is in the usual range for SMT-based verifiers. Interestingly, the liveness specification and proof add only around 10% to both annotation and verification time overhead in both the Memcached and queue programs, even though they require manual applications of proof rules. Our models are larger than one might expect; however, around 40% (in the case of SeqCst queue) of the model lines are boilerplate, much of which could be generated by a macro or otherwise reduced by further focusing on a better user-facing interface. Despite their size, models are easier to review because they are declarative and contain fewer details. The verification times are comparable to those of similarly sized codebases in prior Prusti work. As noted in Sec. 4, our approach is not tied to Prusti itself and could be instantiated in other verifiers, as long as they support obligation-based reasoning, which may lead to performance improvements.

5.1 Memcached

Memcached is an in-memory key-value store which can be accessed through a network protocol. At the most basic view, it stores a mapping from keys to values, where both keys and values are byte sequences. The protocol offers Set, Get, and Delete commands.

We developed a simplified version of Memcached in Rust, and proved that it refines an abstract model using our approach. It is executable and interoperable with the original Memcached for the subset of features that we implemented. We developed it in multiple iterations, each adding features that are present in the original implementation. This iterative process allowed us to demonstrate that our approach is suitable for refinement proofs in an evolving codebase. The protocol parser and serialiser are trusted, since such code was not the focus of this work.

First version (v1). The first version uses Rust's built-in `HashMap` data structure to store key-value pairs, and only supports the Set, Get, and Delete commands. This version is not concurrent yet; it handles one connection at a time. The state and action definitions of the model are shown here:

```rust
enum ConState {
  Idle,
  HaveCommand(AbsCmd),
  HaveResponse(AbsRes),
}
struct AbsState {
  con_cmd: Map<ConId, Seq<AbsCmd>>,
  con_res: Map<ConId, Seq<AbsRes>>,
  con_state: Map<ConId, ConState>,
  cache: Map<Seq<u8>, Option<Seq<u8>>>,
}
```

```rust
enum Action {
  SendCommand(ConId, AbsCmd),
  ReceiveCommand(ConId, AbsCmd),
  SendResponse(ConId, AbsRes),
  ReceiveResponse(ConId, AbsRes),
  ProcessCommand(ConId, AbsCmd, AbsRes),
}
enum GuardKind {
  Storage,
  Connection(ConId),
}
```

`AbsCmd` and `AbsRes` are abstract representations of commands (requests) and responses, respectively. Each incoming connection is identified by a `ConId`. As in the running example in Sec. 2, we represent the incoming and outgoing channels as sequences. Upon accepting a client, the implementation enters a loop, in which the actions `ReceiveCommand`, `ProcessCommand`, and `SendResponse` are performed in turn. The current step for the client is tracked using the **`enum`** `ConState`. The `SendCommand` and `ReceiveResponse` actions are performed by clients connecting to the server, thus we model them as environment actions. Our case study crucially relies on guards to couple the current point in the receive-process-send loop of each client to the `ConState` in the model state, using that client's `Connection(ConId)` guard. The `ProcessCommand` action is additionally protected by the `Storage` guard, which maintains the link between the concrete `HashMap` and the abstract `Map` in the `cache` field. The verification of this first version was straightforward, and involved adding the ghost lock, guard annotations, and ghost updates, keeping track of state in pre- and postconditions and loop invariants, and maintaining the connection between abstract and concrete versions of data structures. The predicates used when opening guards are trivial, since the respective parts of the model state are fully determined by local state.

The liveness property which we prove is `∀con.□◊∃res.SendResponse(con, res)`. To be able to show this, we assume that there is always eventually a new client connection, and that each client always eventually sends a command. These assumptions are encoded by assuming that the `accept` and `read` methods terminate; such assumptions can be lifted with a more precise model of the network. We also ignore error handling in the liveness verification.

Concurrent connections (v2). In a later version, we spawn a thread for each incoming connection, which runs the receive-process-send loop. The `HashMap` is protected by a lock, which is only acquired for the `ProcessCommand` action. We also put the `Storage` guard into the lock, such that it stays together with the concrete state for which it maintains the coupling to the model state. This coupling invariant is then part of the lock invariant. The model definition is unchanged compared to the previous version without concurrency, which shows that our technique can treat concurrency as an implementation detail that is introduced during refinement, but not reflected in the model.

Fine-grained locking (v3). In Memcached, the storage is not protected by a single global lock; instead, each hash table bucket is protected by a separate lock. To implement this, we can no longer use `HashMap`, and instead need our own hash table implementation. Our hash table is a vector of buckets, each inside a lock. Each bucket contains a linked list of items, which stores the key, value, and pointer to the next item.

We no longer have a single `Storage` guard, but one `StorageBucket(usize)` guard for each bucket. This guard is needed for a `ProcessCommand` action if the key that the command operates on hashes to this bucket. Consequently, the predicate used when opening a `StorageBucket` guard is not a simple equality anymore, but a quantifier:

```
bucket.guard.open(gl, |state|
  forall(|key: Seq<u8>| hash_abs(key) == hash ==>
    item_valid(bucket.list, key, state.cache.get(key))));
```

Even though the structure of our implementation changed significantly, i.e., we added concurrency and fine-grained locking to the initially sequential implementation, the model remains unchanged, apart from the guard definitions. This demonstrates that our approach allows for great flexibility in program structure.

The difference in verification times for the successive versions is small. Due to the modular nature of the verifier and our methodology, methods which are not changed need not be re-verified, so with the use of caching[11], interactive development and verification in such a codebase is possible.

5.2 Producer/consumer queue

Our next case study is a single-producer, single-consumer queue, taken from the documentation for Verus Transition Systems [Hance 2022]. This case study demonstrates that our refinement approach can support various concurrency primitives, including atomics with weak ordering guarantees, and that data structures built with these primitives can be verified. We also demonstrate (with the `VerifiedCell` component below) that our refinement methodology is suitable for ensuring safety in the presence of Rust **`unsafe`** code.

[11]Note that Fig. 3 shows the full verification times, without enabling the cache.

Unlike MEMCACHED, this case study is centred around a single data structure. The queue consists of a vector storing the content of the queue, and head and tail pointers. It is accessed through the `Producer` and `Consumer` handles, which are created when the queue is constructed. To allow the `Producer` and `Consumer` to be used from different threads, the head and tail pointers are stored in atomic variables, which allows them to be accessed concurrently. In RUST, a data structure which is shared between threads allows only read access by default. The queue elements, however, are written by one thread and later read by another. Typically, this would require the use of a wrapper data structure which allows *interior mutability*, such as a `Mutex<T>`, which would ensure synchronisation and mutual exclusion, and then allow mutable access to the contained `T`. However, in the queue example, the atomic accesses to the head and tail pointers already provide sufficient synchronisation, so the additional synchronisation overhead of the mutex is unnecessary. Instead, the unverified code uses `UnsafeCell<T>`, which provides raw interior mutability. Any access to the contents of the `UnsafeCell` must be wrapped in an `unsafe { ... }` block, which indicates that safety must be checked manually be the user.

To allow verification of code which uses `UnsafeCell`, we introduce `VerifiedCell<T>`, a wrapper around `UnsafeCell` which relies on verification for checking the safety requirements, and is thus safe to use. Concretely, access to an `UnsafeCell` is only safe if there are no data races. A program has a *data race* if there are two accesses to the same location, where at least one is a write, at least one is non-atomic, and there exists no happens-before relation between the accesses [ISO 2021]. *Happens-before* is a partial order of all memory accesses.

For each atomic operation (read or write call), and each non-atomic access to a `VerifiedCell`, we define an abstract operation identifier. We then encode the happens-before relation as a binary predicate between these identifiers. The `VerifiedCell` accessor method ensures memory safety by requiring that happens-before holds between subsequent accesses.

We verified two versions of the queue, one with sequentially consistent and one with acquire-release atomics. Acquire-release atomics offer better performance, but are more difficult to reason about, as evidenced by the increased amount of annotations needed to verify this version of the queue. The key difference compared to sequentially consistent atomics in terms of the abstract model is that we store a *set* of operation identifiers for each atomic variable, representing all write operations performed so far, as opposed to only the single identifier of the last write operation.

5.3 Other case studies

Finally, we also specified and verified a version of the distributed consensus algorithm PAXOS [Lamport 1998], as well as a lock-free *hash-set* [Kuppe 2017], used in the implementation of the TLC model checker. In both cases we were able to verify an executable implementation with relatively low specification overhead and acceptable verification times.

6 RELATED WORK

Various approaches [Lesani et al. 2016; Rahli et al. 2018; Sergey et al. 2018; Woos et al. 2016] develop implementations that are correct by construction by refining abstract models within COQ and then extracting executable OCaml programs. Similarly, Liu et al. [2020] model distributed systems in Maude's rewriting logic and compile them into implementations running in distributed Maude sessions. The code extracted by these approaches is typically sub-optimal (for instance, does not use mutable data structures) and cannot interface with existing libraries, which is often necessary in practice. In contrast, our methodology uses bottom-up verification and can handle efficient implementations using concurrency, distribution, and node-local mutable state.

TRILLIUM [Timany et al. 2021] is a refinement technique based on separation logic. Like our methodology, it does not impose strict requirements on the structure of the implementations, and the function of invariants in Trillium is similar to our ghost lock. Furthermore, because Trillium is based on Iris and formalised in Coq, it provides an expressive specification language and enables foundational correctness proofs. On the other hand, proofs in this framework are not easily automatable and require extensive manual work. Instead, our methodology represents abstract models in first-order logic and automates verification using an SMT solver.

Armada [Lorch et al. 2020] supports the verification of concurrent, high-performance code written in a C-like language. To achieve refinement against an abstract model, the user specifies a sequence of steps to gradually transform the implementation into the specification. Non-trivial refinement steps require complex Dafny [Leino 2010] proofs showing a connection between two state machines. Unlike Armada, our methodology does not convert programs to state machines and the coupling between the abstract model and the implementation can be much looser. The CIVL verifier [Hawblitzel et al. 2015b; Kragl et al. 2020] also organises the refinement proof into multiple layers. Each layer is a structured concurrent program, where the concurrent behavior is reflected in the program structure. This structure simplifies the proof obligations and allows automation, but also reduces program flexibility. Refinement steps are based on a set of trusted tactics. By contrast, our methodology imposes no restrictions on the program or proof structure. Igloo [Sprenger et al. 2020] connects abstract models to concrete implementations via dedicated I/O specifications [Penninckx et al. 2015]. Similarly to our work, they support a variety of separation logics to reason about concrete implementations. However, their technique has not been shown to allow for threads performing I/O operations concurrently, whereas we have shown that our methodology has no such limitation.

Similar to our methodology, IronFleet [Hawblitzel et al. 2015a] embeds abstract models as ghost state into executable programs and automates verification using an SMT-based verifier, in their case Dafny. However, their refinement technique imposes severe restrictions on how programs are structured. In particular, the programs must be sequential and their structure must mirror the structure of the abstract model. Like IronFleet, our approach supports verifying liveness properties. Unlike our temporal library, their API to prove temporal properties is proven correct against a model of the system's behaviour. On the other hand, IronFleet makes use of *always-enabled* actions to guarantee certain liveness properties by construction. It is unclear how this reasoning can be extended to unbounded transitions or guard-based reasoning.

ironsync [Hance et al. 2023] also embeds the abstract model as ghost state, and, like us, uses ownership to reason about accesses to the model state. In their approach, the abstract model is decomposed into *shards*, i.e., resources that provide access to a partial view of the global state. Importantly, shards can only be *owned* by a single thread. Actions in the system are specified in terms of the parts of the state that they access. Thus, a thread may only perform an action if it owns all the shards that the action depends on – this is similar to how we use guards to reason about allowed actions. Guards allow for a more precise accounting of the allowed actions: two actions may modify the same part of the abstract state; in ironsync, it is not clear how to express that only one action may have taken place when a thread does not hold the shard. In our approach, we express this by keeping the ownership of one of the guards, but not the other. Finally, their work focuses only on safety properties, whereas we support both safety and liveness properties. We believe that our novel technique of tying LTL formulas to obligations could be applied to their setting to support liveness properties.

The refinement technique [Koh et al. 2019] used in DeepSpec [Appel et al. 2017] is based on the Verified Software Toolchain (VST) [Cao et al. 2018], a framework for verifying C programs via a separation logic embedded in Coq. Instead of transition systems, they specify the intended system behavior using interaction trees [Xia et al. 2020], which are embedded into VST's separation logic. In contrast, our methodology allows us to apply standard separation logics and existing program verifiers. Oortwijn and Huisman [2019] embed process calculus models into a concurrent SL, which is automated using Viper. Their refinement approach preserves state assertions, but it is unclear whether arbitrary trace properties are preserved.

7 CONCLUSION

In this paper, we have introduced a novel methodology for refinement proofs of programs written in a high-level language that refine an abstract transition system model. The methodology centres around the use of ghost locks to allow flexible program structure and concurrency, showing both safety and liveness proofs. We have implemented our approach in the Prusti verifier, and evaluated our approach on several case studies, including an implementation of Memcached, demonstrating that the approach is expressive, amenable to automation, and performant. As future work, we plan to more thoroughly address initialisation in distributed systems; to provide better automation for the LTL fragment; and to formalise our approach.

REFERENCES

Andrew W Appel, Lennart Beringer, Adam Chlipala, Benjamin C Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20160331.

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: Managing obligations in higher-order concurrent separation logic. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

Pontus Boström and Peter Müller. 2015. Modular Verification of Finite Blocking in Non-terminating Programs. In *29th European Conference on Object-Oriented Programming*. 639.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. 2018. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning* 61 (2018), 367–422.

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. Verifying safety properties with the TLA+ proof system. In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*. Springer, 142–148.

Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. 2010. Local verification of global invariants in concurrent programs. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 480–494.

Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. C aper: automatic verification for fine-grained concurrency. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings 26*. Springer, 420–447.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP 2010–Object-Oriented Programming: 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings 24*. Springer, 504–528.

Gidon Ernst and Alexander Weigl. 2023. 2nd VerifyThis Long-term Challenge: Specifying and Verifying a Real-life Remote Key-Value Cache (memcached). (2023).

Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81.

Jafar Hamin and Bart Jacobs. 2019. Transferring obligations through synchronizations. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, Vol. 134. Dagstuhl LIPIcs; Dagstuhl, Germany, 19–1.

Travis Hance. 2022. Verus Transition Systems. https://verus-lang.github.io/verus/state_machines/. Accessed: 2023-11-15.

Travis Hance, Yi Zhou, Andrea Lattuada, Reto Achermann, Alex Conway, Ryan Stutsman, Gerd Zellweger, Chris Hawblitzel, Jon Howell, and Bryan Parno. 2023. Sharding the State Machine: Automated Modular Reasoning for Complex Concurrent Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 911–929. https://www.usenix.org/conference/osdi23/presentation/hance

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015a. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating*

Systems Principles. 1–17.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015b. Automated and modular refinement reasoning for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 449–465.

Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

ISO. 2021. *International Standard ISO/IEC 14882:2020(E) – Programming Language C++*. International Organization for Standardization (ISO), Geneva, Switzerland.

Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems: 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28-December 1, 2010. Proceedings 8*. Springer, 304–311.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 207–220.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 234–248.

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 275–298.

Markus A Kuppe. 2017. *A Verified and Scalable Hash Table for the TLC Model Checker*. Master's thesis. University of Hamburg.

Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370.

K Rustan M Leino, Peter Müller, and Jan Smans. 2010. Deadlock-free channels and locks. In *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 407–426.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 42–54.

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Principles of Programming Languages (POPL)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 357–370.

Richard J Lipton. 1975. Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18, 12 (1975), 717–721.

Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs. In *NASA Formal Methods (Lecture Notes in Computer Science, Vol. 12229)*, Ritchie Lee, Susmit Jha, and Anastasia Mavridou (Eds.). Springer, 22–40.

Jacob R Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 197–210.

Zohar Manna and Amir Pnueli. 1991. Completing the temporal picture. *Theoretical Computer Science* 83, 1 (1991), 97–130.

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. *ACM SIGAda Ada Letters* 34, 3 (Nov 2014), 103–104. https://doi.org/10.1145/2692956.2663188

Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model checking, and Abstract interpretation (VMCAI)*. Springer, 41–62.

Wytse Oortwijn and Marieke Huisman. 2019. Practical Abstractions for Automated Verification of Message Passing Concurrency. In *Integrated Formal Methods (iFM) (Lecture Notes in Computer Science, Vol. 11918)*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer, 399–417.

Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 158–182.

Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 619–650.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018), 28:1–28:30.

Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly linking compositional refinement and separation logic for distributed system verification. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2021. Trillium: Unifying refinement and higher-order distributed separation logic. *arXiv preprint arXiv:2109.07863* (2021).

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Certified Programs and Proofs (CPP)*, Jeremy Avigad and Adam Chlipala (Eds.). 154–165.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32.

A TLA$^+$ SPECIFICATION OF RUNNING EXAMPLE

MODULE *Example*

EXTENDS $Naturals, Sequences$
VARIABLES $ACtr, BWork, BWorkNum, AToB, BToA$

$$Vars \triangleq \langle ACtr, BWork, BWorkNum, AToB, BToA \rangle$$

$$\begin{aligned} Init \triangleq\ & \land ACtr = 0 \\ & \land BWork = \text{FALSE} \\ & \land BWorkNum = 0 \\ & \land AToB = \langle\rangle \\ & \land BToA = \langle\rangle \end{aligned}$$

$$\begin{aligned} TypeInv \triangleq\ & \land ACtr \in Nat \\ & \land BWork \in \text{BOOLEAN} \\ & \land BWorkNum \in Nat \\ & \land AToB \in Seq(Nat) \\ & \land BToA \in Seq(Nat \times Nat) \end{aligned}$$

$$\begin{aligned} ASend \triangleq\ & \land AToB' \\ & \quad = Append(AToB, ACtr) \\ & \land \text{UNCHANGED } \langle ACtr, BWork, \\ & \qquad BWorkNum, BToA \rangle \end{aligned}$$

$$\begin{aligned} ARecv \triangleq\ & \land BToA \neq \langle\rangle \\ & \land ACtr' = Max(Head(BToA)[1] + 1, ACtr) \\ & \land BToA' = Tail(BToA) \\ & \land \text{UNCHANGED } \langle BWork, BWorkNum, AToB \rangle \end{aligned}$$

$$\begin{aligned} BSend \triangleq\ & \land BWork = \text{TRUE} \\ & \land BWork' = \text{FALSE} \\ & \land \exists\, Resp \in Nat : BToA' \\ & \quad = Append(BToA, \langle BWorkNum, Resp \rangle) \\ & \land \text{UNCHANGED } \langle ACtr, BWorkNum, AToB \rangle \end{aligned}$$

$$\begin{aligned} BRecv \triangleq\ & \land BWork = \text{FALSE} \\ & \land AToB \neq \langle\rangle \\ & \land BWork' = \text{TRUE} \\ & \land BWorkNum' = Head(AToB) \\ & \land AToB' = Tail(AToB) \\ & \land \text{UNCHANGED } \langle ACtr, BToA \rangle \end{aligned}$$

$$\begin{aligned} ALoss \triangleq\ & \land BToA \neq \langle\rangle \\ & \land BToA' = Tail(BToA) \\ & \land \text{UNCHANGED } \langle ACtr, BWork, \\ & \qquad BWorkNum, AToB \rangle \end{aligned}$$

$$\begin{aligned} BLoss \triangleq\ & \land AToB \neq \langle\rangle \\ & \land AToB' = Tail(AToB) \\ & \land \text{UNCHANGED } \langle ACtr, BWork, \\ & \qquad BWorkNum, BToA \rangle \end{aligned}$$

$$\begin{aligned} Next \triangleq\ & \lor ASend \\ & \lor ARecv \\ & \lor BSend \\ & \lor BRecv \\ & \lor ALoss \\ & \lor BLoss \end{aligned}$$

$$\begin{aligned} Live \triangleq\ & \land \text{SF}_{Vars}(ASend) \\ & \land \text{WF}_{Vars}(BSend) \end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{Vars} \land Live$$

THEOREM $Spec \implies \Box TypeInv$

B FULL CODE LISTING OF THE NEXT PREDICATE

```
fn next(p: SystemState, s: SystemState, a: Action) -> bool {
  match a {
    Action::ASend => s == SystemState {
      a_to_b: p.a_to_b.append(p.a_ctr),
      ..p
    },
    Action::ARecv => p.b_to_a.len() > 0
      && s == SystemState {
        b_to_a: p.b_to_a.tail(),
        a_ctr: p.a_ctr.max(p.b_to_a.head().0 + 1),
        ..p
      },
    Action::ALoss => p.b_to_a.len() > 0
      && s == SystemState {
        b_to_a: p.b_to_a.tail(),
        ..p
      },
    Action::BSend(resp) => match p.b_work {
      Some(req) => s == SystemState {
        b_to_a: p.b_to_a.append((req, resp)),
        b_work: None,
        ..p
      },
      None => false,
    },
    Action::BRecv => p.a_to_b.len() > 0
      && p.b_work == None
      && s == SystemState {
        a_to_b: p.a_to_b.tail(),
        b_work: Some(p.a_to_b.head()),
        ..p
      },
    Action::BLoss => p.a_to_b.len() > 0
      && s == SystemState {
        a_to_b: p.b_to_a.tail(),
        ..p
      },
  }
}
```

C RELATION TO TLA$^+$ AND OTHER FORMALISMS

We give a brief overview of the correspondence between our transition systems and ones defined in formalisms like TLA$^+$.

State, predicates

The `AbsState` type of the model trait (`SystemState` in the running example from Sec. 2) combines all the variables of the abstract system with ones related to the environment. In TLA$^+$, both categories map to VARIABLES declarations.

The `init` predicate of our system also naturally maps to the *Init* predicate in TLA$^+$.

Within the `next` predicate, our approach prescribes a more specific form than TLA$^+$. In TLA$^+$, *Next* can be an arbitrary two-state predicate, although it is common that it is a formula that is a disjunction of existentially quantified conjunctions (actions). In our approach, the outer disjunction is represented by the **`enum`** type of action labels. Furthermore, the existential quantifiers are replaced by data stored in the **`enum`** variants. As noted in Sec. 2.3, this form of actions was chosen for better automation, and for the programmer to be able to better communicate intent when transitions take place.

Types

Our model definitions are embedded directly in Rust, a strongly typed language. TLA$^+$ is not a typed language, though it is common for TLA$^+$ modules to contain a definition of a type invariant which must be preserved by any transition.

The types used in our case studies, such as natural numbers and abstract sequences, sets, or maps, all have a counterpart in TLA$^+$. It would thus be possible to obtain a definition of the type invariant from the types in our model.

Guards

Unlike in our approach, guard-based reasoning is not a first-class feature in TLA$^+$. It is, of course, possible to encode guards as ghost values within a TLA$^+$ module. However, the concrete semantics of such guard algebras can vary from module to module.

Liveness

The LTL formulas we use to express liveness properties can directly map to TLA$^+$, since it also uses LTL for temporal reasoning.

Summary

In summary, mapping from our approach to TLA$^+$ (or similar transition system-based formalism) is possible and automatable. A mapping in the other direction would require the source model to have additional annotations, or else depend on heuristics, e.g. to infer types for variables based on the definition of actions.