A Refinement Methodology for Distributed Programs in Rust

AUREL BÍLÝ, ETH Zurich, Switzerland
JOÃO PEREIRA, ETH Zurich, Switzerland
PETER MÜLLER, ETH Zurich, Switzerland

Refinement relates an abstract system model to 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. Top-down refinement techniques that automatically generate executable code generally produce implementations with sub-optimal performance. Bottom-up refinement allows one to reason about existing, efficient implementations, but imposes 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 bottom-up refinement methodology that removes these limitations. Our methodology uses the familiar notion of guarded transition systems to express abstract models, but combines guards with a novel notion of *locally inductive invariants* to relate the abstract model *locally* to concrete state. This approach is much more flexible than standard coupling invariants; in particular, it supports a wide range of program structures, data representations, and proof structures. We integrate our methodology as a library into Rust, leveraging the Rust type system to reason about ownership of guards. This integration allows one to use our methodology with an off-the-shelf Rust verification tool. It also facilitates practical applications, as we demonstrate on a number of substantial case studies including a concurrent implementation of Memcached.

CCS Concepts: • **Software and its engineering** → Software verification; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: Refinement

ACM Reference Format:
Aurel Bílý, João Pereira, and Peter Müller. 2025. A Refinement Methodology for Distributed Programs in Rust. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 341 (October 2025), 28 pages. https://doi.org/10.1145/3763119

1 Introduction

Distributed systems are notoriously difficult to implement correctly: additional complexity is introduced due to the unpredictability of network failures, communication latency, and the need to interact with low-level system APIs. At the same time, such systems form the backbone of modern communication networks, resulting in stringent performance and correctness requirements. The Rust language [38] is increasingly used in the development of such systems due to its unique combination of modern language features, memory safety guarantees, and access to efficient low-level operations, but its guarantees do not extend to functional safety or system-wide correctness properties.

One effective approach to reducing the complexity of reasoning about distributed systems is *refinement*, wherein the implementation is proven to refine an abstract model, written in a high-level mathematical language and, thus, satisfies all safety properties of the abstract model. The upshot is that reasoning about the protocol, i.e., the abstract behaviour of the nodes of a

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

ACM 2475-1421/2025/10-ART341
https://doi.org/10.1145/3763119

distributed system, is cleanly separated from implementation details. *Top-down* refinement derives or automatically generates implementations from abstract models, often resulting in implementations with sub-optimal performance. In this work, we focus on the alternative *bottom-up refinement*, which allows us to verify efficient Rust implementations including ones not initially written with verification in mind. Recent works have shown this approach to be practically applicable to large-scale codebases [17, 37, 45].

Desiderata. We identify the following requirements for a refinement methodology suitable for verification of large-scale distributed systems:

(**C**1) *Program Modularity and Scalability*: It must be possible to verify each method and program component (including libraries) of a distributed system in isolation, then rely on composition to obtain a proof of the complete system. To this end, component specifications must be expressive enough to expose properties proven by refinement without leaking implementation details.

(**C**2) *Model Modularity*: Distributed systems are often composed of smaller sub-systems and protocols; for example, HTTP communication over TLS is the basis for virtually any browser-based application. It must be possible to define and reason about the abstract models for these sub-systems and protocols in isolation, and, as in (**C**1), rely on composition to obtain a model and proofs for the entire system.

(**C**3) *Proof Modularity*: Bottom-up refinement disentangles abstract models from concrete implementations. A model should not prescribe how to implement it, allowing for straight-forward sequential reference implementations as well as optimised, multi-threaded implementations.

(**C**4) *Program Structure Flexibility*: A requirement that goes hand in hand with the previous one is that a refinement methodology should allow a variety of implementation structures, including different choices for control structures, data structures, and concurrency.

(**C**5) *Automation*: Obtaining a refinement proof should require as little user input as possible while maintaining acceptable verification times. In the setting of SMT-based *deductive verification*, annotations are nonetheless commonly required at all verification boundaries, i.e., methods and loops need to be annotated with contracts and invariants, respectively.

(**C**6) *Language Integration*: Finally, the methodology and its accompanying proof annotations must be tightly integrated into the implementation language, i.e., expressed in concepts familiar to a user of the language. This facilitates practical applications, and further allows the re-use of existing tooling, such as compilers, IDEs, and even verifiers.

Criteria (**C**1) to (**C**3) correspond to the modularity dimensions identified by Sergey et al. [44]. Collectively, they are also necessary to minimise re-verification effort as models and implementations evolve. (**C**4) is a necessity motivated by bottom-up refinement specifically, otherwise verifying existing implementations would be impractical. Both (**C**5) and (**C**6) are motivated by our goals to support substantial codebases and to increase the productivity of verification engineers.

State of the Art. To the best of our knowledge, no refinement approach satisfies all the aforementioned criteria.

A number of recent refinement approaches [26, 44–46, 48] are mechanised in an interactive theorem prover (in these cases Rocq [7] and Isabelle/HOL [8]) and require significant manual effort to produce refinement proofs or the development of an extensive set of automated tactics as in ReLoC [14]. Code resulting from extraction in these approaches uses a limited set of trusted shims to interact with the host system, thus limiting program flexibility.

Other methodologies [16, 50] impose an event-loop structure on the implementation. This allows for easier reasoning about enabled action handlers, and even some liveness guarantees as in IronFleet [17], but does not reflect the organisation of real-world distributed system implementations, is difficult to evolve, and is impractical to extend to multi-threaded implementations.

Armada [37] supports the verification of concurrent, high-performance code written in a C-like language. Refinement proofs are structured as a sequence of steps which gradually transform the implementation into the specification. Non-trivial refinement steps require complex Dafny [33] proofs showing a connection between two state machines.

The Civl verifier [18, 25] also organises the refinement proof into multiple layers. Each layer is a structured concurrent program, where the concurrent behaviour 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.

F★ [47], a verification-focused language with a high degree of SMT-based automation, supports reasoning about distributed systems through session types. More recently, a monotonic-state monad was used [2] to prove stable invariants over (possibly distributed) states, although the invariants must be picked globally on a per-application basis, which limits model and proof modularity.

Verus [31] is an SMT-automated Rust verifier. Its refinement reasoning is based on IronSync [15], which defines how the model state is *sharded* across distributed instances, using a set of predefined sharding strategies. Users define additional transition systems characterising the behaviour of the model only w.r.t. to the state subsets, then separately prove a connection to the global transition system. These transition systems and proofs incur a high specification overhead.

Approach. Our methodology aims to prove that a concrete implementation, written in Rust, refines an abstract model, defined as an abstract transition system. We identify Rust as an ideal implementation language in a bottom-up refinement methodology, as it enables efficient implementations and at the same time provides a solid foundation for verification [4, 9, 19, 31].

The relevant behaviour of a distributed system is determined by how its nodes communicate with each other, typically through messages. Therefore, the goal of our refinement proofs is to show that any *observable behaviour* exhibited by the implementation is also allowed by the model[1]. In particular, any observable step in the implementation, which is in practice performed by a call to a standard library method such as a socket write, must correspond to an abstract transition in the model. Our methodology does not restrict how internal (non-observable) operations are performed; technically, they either correspond to non-observable changes in the model or to *stutter steps*, where the state of the model remains unchanged.

To integrate our methodology with Rust (**C**6), we embed the model being refined as *ghost code* in the program, that is, code that is present for verification, but erased by the compiler. Similarly to Iris invariants [23], access to the state is encapsulated in an abstract, separation logic resource that can be shared among different methods and threads. During observable steps (such as network access, filesystem operations, shared memory usage), the resource is temporarily *opened*, giving access to the abstract state, which may be updated by the application and standard library method calls. Any change performed must correspond to a valid transition of the abstract model.

To enable distributed and concurrent implementations (**C**4), we express abstract models as *guarded* transition systems. *Guards* are linear resources, whose exclusive ownership is enforced by Rust's type system. They express the ability to perform transitions in the abstract model. During verification, the ownership of a guard restricts the possible environment interference, i.e., it restricts the possible modifications other nodes or threads may make to the model state. In particular, having

[1]This notion extends to multi-threaded systems as well: the usage of shared memory or synchronisation primitives is also considered observable behaviour, even though it is localised to one machine.

observed the model state in a previous step allows one to establish that certain inductive invariants must still hold in a later step, so long as the guard ownership was not relinquished in the meanwhile. In general, such invariants are not invariants of the entire model, but they are inductive invariants during the execution segment in which holding a guard rules out certain environment steps. Hence, we refer to them as *locally inductive invariants*.

While guarded transition systems are a standard technique for rely-guarantee reasoning in concurrent and distributed systems, our locally inductive invariants are novel and offer another major benefit: since they must hold only *locally* within a method or thread, they can refer to local state, such as local variables. This allows them to express relations between the abstract model and arbitrary representations in the concrete state. This is in contrast to traditional global *coupling invariants* seen in other methodologies [32]. Because the choice of representation is left to the implementation, the model itself can be defined without knowledge of any particular implementation (**C**3).

To be able to reason about the components of a system in isolation (**C**2), we define a composition system for models. Logical components of a system, such as TCP/IP networks, define their own abstract models with their own notion of a state, transitions, and guards. This model state can be modified only by methods belonging to that component. An application can make use of these independently defined models by providing a *projection*: a way to map from the application-specific model to the domain-specific one and back. We leverage the same mechanism, in combination with Rust trait-bounded generics, to be able to provide specifications of standard library methods *once and for all* (**C**1), i.e., such specifications can be exported and maintained independently of individual applications.

We integrate our methodology into Rust as a reusable library, which offers support for expressing models as guarded transition systems, connecting concrete implementations to these models via locally inductive invariants, equipping standard library functions with specifications, and composing models. The specifications in our library are usable with Prusti [4], an SMT-based verifier for Rust, but our methodology is compatible with other Rust verifiers. As is standard in all deductive verifiers, specifications must be provided by the user in the form of preconditions, postconditions, and loop invariants.

To evaluate our methodology, we have verified several major cases studies, including a version of Paxos [29], and a multi-threaded implementation of the Memcached caching system [13]. For the latter, we demonstrate that our methodology is applicable to an existing implementation, and also that there is relatively low overhead to adapting the proof as the codebase evolves.

Contributions. This work makes the following contributions:

(1) A novel bottom-up refinement methodology for distributed and concurrent systems, allowing for independent definitions of models and their implementations, for procedure-modular verification, and for model composition. The methodology includes novel *locally inductive invariants* (LIIs) as well as a notion of model composition and projection.
(2) An integration of our methodology in Rust, suitable for off-the-shelf SMT-based verifiers.
(3) An evaluation of our methodology on several substantial case studies, including implementations of Paxos and Memcached, demonstrating its effectiveness.

Outline. We explain our methodology in Sec. 2 for a simple Rust-like language with first-class support for abstract model declarations and model transitions. In Sec. 3 we discuss the soundness of our approach and in particular LIIs, using a toy language formalisation. Sec. 4 presents the integration into Rust. We present our evaluation in Sec. 5, discuss related work in Sec. 6, and conclude in Sec. 7.

2 Methodology

As in standard refinement work [1], we model the behaviour of a distributed system as a transition system, consisting of a set of variables making up the *abstract state* (or *model state*); a one-state predicate `Init` identifying valid initial states; and a two-state predicate `Next` identifying valid state transitions. Following common practice in TLA$^+$ [28], we further consider `Next` to be a disjunction of existentially quantified conjunctions, with each disjunct corresponding to a *labelled* transition or *action*. A subset of the variables is identified as *observable*: for example, the emission of a packet is observable by machines on the network. Ensuring that the implementation exhibits a subset of the observable behaviour of the model is the goal of our methodology.

We annotate the implementation with *transition blocks* which declare that changes to the model state may be performed in the contained code fragment. The refinement proof must establish three properties: (1) that the `Init` predicate is established by the initialisation code (→ Sec. 2.7); (2) that each transition block modifies the model state according to the `Next` predicate (→ Sec. 2.4); and (3) that the model state is not modified outside of transition blocks (→ Sec. 2.4). The use of explicit transition blocks helps to enforce the third property, and additionally localises the proof of the second property to the delineated pieces of code.

2.1 Running Example

Throughout this section, we will demonstrate our approach on a running example, consisting 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.

2.2 Model Declaration

We will initially consider a hypothetical programming language RustRef which has first-class support for refinement proofs, i.e., the abstract model is declared and the refinement proof is directly expressed in the language; we will describe the full Rust integration in Sec. 4. The exact syntax of RustRef is not important, but it is an extension of Rust with declarations of `Vars`, `Observable`, `Init`, and `Next`, the latter two of which are defined using standard FOL connectives. The language also contains transition blocks (**`next`**), and method calls which perform I/O operations and also change the observable part of the model state.

Fig. 2 shows the RustRef version of the model for the running example from Fig. 1[2]. It consists of Ⓐ declaring a set of variables, Ⓑ some of which are observable; Ⓒ defining their initial values in the `Init` predicate; Ⓓ defining the `Next` predicate as a disjunction of the six possible

[2]A complete version of the model in TLA$^+$ can be seen in Appendix A.

```rust
fn node_a(ch: Channel<i32, (i32, i32)>) {
  let mut ctr = 0;
  loop {
    // send
    a_send(&mut ch, ctr);
    // wait for response, with timeout
    match a_recv_timeout(&mut ch) {
      Some((n, resp)) if ctr == n =>
        { ctr += 1; }
      _ => {}
    }
  }
}
```

```rust
fn node_b(ch: Channel<(i32, i32), i32>) {
  loop {
    // receive (blocking)
    let num = b_recv(&mut ch);
    // compute
    let resp = do_something(num);
    // respond
    b_send(&mut ch, (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 and assume a connection between the two nodes is pre-established. Here, `Channel<X, Y>` is a two-way channel to which values of type `X` can be sent, and from which values of type `Y` can be received.

RustRef: Model

```
Vars := a_ctr, b_work, a_to_b, b_to_a (A)
Observable := a_to_b, b_to_a (B)
Init := ∧ a_ctr = 0 (C)
        ∧ b_work = None
        ∧ a_to_b = []
        ∧ b_to_a = []
Next := ∨ ASend ∨ ARecv ∨ ALoss ∨ BSend ∨ BRecv ∨ BLoss (D)
ASend := ∧ a_to_b' = a_to_b ++ [a_ctr] (E)
         ∧ UNCHANGED a_ctr, b_work, b_to_a
ARecv := ∧ |b_to_a| > 0
         ∧ b_to_a' = tail(b_to_a)
         ∧ a_ctr' = max(head(b_to_a).0 + 1, a_ctr)
         ∧ UNCHANGED b_work, a_to_b
ALoss := ∧ |b_to_a| > 0
         ∧ b_to_a' = tail(b_to_a)
         ∧ UNCHANGED a_ctr, b_work, a_to_b
... (node B's actions omitted)
```

Fig. 2. Model definition for the system from Fig. 1, consisting of a definition of the abstract transition system. The analogous definitions for node B's actions are omitted. Borrowing from TLA$^+$, `x'` represents the value of `x` in the *next* state, i.e., after a transition has taken place; `UNCHANGED x` is a shorthand for `x' = x`.

actions; (E) definitions of each action. The action `ASend`, for example, updates the value of `a_to_b` while keeping the other variables unchanged. The two variables `a_to_b` and `b_to_a` are queues of packets transmitted over the network from A to B and from B to A, respectively. The actions `ASend` and `BSend` enqueue packets into these variables, while the other actions dequeue them.

2.3 I/O Operations

Modifications to the observable part of the model state are performed exclusively by I/O method calls. In real-world programming languages, such methods are typically provided by the standard library. Note that model actions generally consist of a combination of changes to the observable and

RustRef: I/O

```
ensures a_to_b' = a_to_b ++ [v]
ensures b_to_a' = b_to_a
fn a_send(ch: &mut Channel<i32, (i32, i32)>, v: i32);

ensures |b_to_a| > 0 ⇒ b_to_a' = tail(b_to_a)
                     ∧ result = Some(head(b_to_a))
ensures |b_to_a| = 0 ⇒ b_to_a' = b_to_a
                     ∧ result = None
ensures a_to_b' = a_to_b
fn a_recv_timeout(ch: &mut Channel<i32, (i32, i32)>) -> Option<(i32, i32)>;
```

Fig. 3. I/O operations used in the system implemented in Fig. 1. The analogous definitions for node B's I/O operations are omitted. Note that the abstract model defined in Fig. 2 has no notion of time, thus we define the receive-with-timeout operation as immediately returning None if there are no messages in the queue.

non-observable model variables. For example, ARecv modifies the non-observable variable a_ctr (representing the state of node A) as well as the queue b_to_a (representing part of the state of the network).

In RustRef, we declare the I/O operations as methods that can read and modify *only* the observable part of the state, and that can be called from within the implementation like any other method. Since such calls modify the model state, they may appear only within transition blocks to guarantee that each node maintains a consistent view of the model state.

As is standard in modular verification, we assume that all library methods, including those performing I/O, are equipped with (trusted) specifications. The I/O operations used in Fig. 1 are shown in Fig. 3. Here, the operations and their specifications are specific to the model and even each node; we will discuss how to define them generically in Sec. 4.6.

2.4 Transition Blocks

To prove refinement, we need to show that every step of a concrete implementation corresponds either to a *stutter step* in the model, i.e., a step where the abstract state is not modified; or to a valid transition according to the Next predicate. To this end, we annotate the implementation with *transition blocks* which indicate that changes to the model state may be performed in the contained code fragment (conversely, code outside a transition block must not affect the model state). In addition to performing operations on the concrete state, the code may directly update the non-observable part of the model state (in RustRef accessible through the variable model) and perform I/O operations to indirectly modify the observable part. The combined effect must correspond to a valid transition. The loop body in node A's implementation from Fig. 1, annotated with transition blocks and direct model updates, looks as follows:

```
// send
next { Ⓐ
  a_send(&mut ch, ctr);
}
next { Ⓑ
  // wait for response, with timeout
  match a_recv_timeout(&mut ch) {
    Some((n, resp)) if ctr == n => {
      model.a_ctr += 1;
      ctr += 1;
    }
    None => (), Ⓒ
  }
}
```

This implementation repeatedly performs two observable actions: sending a message and receiving a response, thus we annotate the code with two transition blocks, denoted by the **`next`** keyword. When control flow reaches the end of a transition block, the code must have either left the model state unchanged, performing a stutter step; or else it must have performed exactly one valid transition. The block Ⓐ always performs the `ASend` action, but we can prove this only if the `ctr` variable remains equal to the *model* variable `a_ctr`; we will discuss in Sec. 2.6 how to maintain such relations between model and concrete state. The block Ⓑ may perform a stutter step if the receive operation times out or if the received response is not the one A last requested (reaching Ⓒ); otherwise it performs the `ARecv` action, but again only if `ctr` was equal to `a_ctr` to begin with.

In general, the proof obligations for a transition block are as follows.

- Within a transition block, the variable `model` is declared to provide access to the model state. Its value at the beginning of the block is an arbitrary abstract state[3]. More precise assumptions about the model state are discussed in Sec. 2.6.
- Verify the body of the transition block using a standard verification technique. As usual, this includes showing that ghost code does not affect the non-ghost execution, such that it can be safely erased. In addition, we need to prove that the block executes atomically and terminates (so that it corresponds to an abstract transition, which executes atomically).
- Check that at the end of the block either the model state was not changed, i.e., a stutter step was performed, or the original model state and the updated model state are related by the `Next` predicate, i.e., a valid action was performed.

Recall that we define `Next` to be a disjunction of existentially quantified conjunctions. When verifying that an action was performed, the verifier must therefore be able to prove at least one existential quantifier. For automation purposes, we ask the programmer to provide the intended action as additional program annotation (→ Sec. 4.4).

Code *outside* of transition blocks must not modify the model state in any way, such that its execution corresponds to stutter steps in the refinement proof. To enforce this property, our methodology verifies (1) that model variables are not modifiable outside of transition blocks; and (2) that I/O operations (→ Sec. 2.3) are not called outside of transition blocks. In RustRef, the former property is simply a type check: the special `model` variable is declared only within the transition blocks; the latter property is a syntactic check.

2.5 Guards

Guarded transition systems, where guards represent a node's capability to perform certain actions in the system, are a common solution when reasoning about shared-state concurrency [10, 11]. They enable rely-guarantee reasoning about environment interference. Our methodology makes use of guards in a novel fashion; for the sake of clarity, this subsection explains how *standard* guard reasoning fits into our approach, our novel extensions are then explained in Sec. 2.6.

A model definition which makes use of guards in RustRef must declare the set of all guards in the system, along with a mapping from actions to the guards required to perform those actions: an action may require zero, one, or more guards. In our example, we declare the guard `GA`, required for the actions `ASend`, `ARecv`, and `ALoss`; and the guard `GB`, required for `BSend`, `BRecv`, and `BLoss`.

In the refinement proof, guards are affine resources exclusively owned by nodes, i.e., every guard has exactly one owner at a time (an executing node or the environment), or it has no owner (the guard cannot be used any more). To ensure that the implementation respects guards, we impose the following proof obligation on transition blocks (in addition to those from Sec. 2.4):

[3]The model state is typed, i.e., only the analogue of `TypeInv` in TLA$^+$ specifications is assumed.

- Check at the end of the transition block that every guard required for the action performed by the transition block, if any, is owned by the current node.

The implementation from Fig. 1 is valid if and only if there is only one instance of node A running: two concurrent instances might not produce a monotonically increasing sequence of requests[4]. Assuming the initialisation code passes `GA` to node A, this property is enforced: since there is only one instance of `GA` in the entire system, there can only be one instance of node A performing the protected actions `ARecv`, `ASend`, and `ALoss`.

2.6 Locally Inductive Invariants

Refinement proofs need to maintain a relation between the abstract state of the model and the concrete state of the implementation. For instance, in Sec. 2.4 we have seen that the proof for node A requires the abstract counter `a_ctr` to remain in sync with the concrete counter `ctr`.

The standard solution is to express such relations using a *coupling invariant* [32], relating the concrete implementation state to the model state before and after each transition. However, expressing and maintaining a coupling invariant is difficult when the concrete state is split across multiple threads or nodes, and when the concrete state is represented using local variables or the callstack shape rather than variables (e.g., which phase of a protocol we are in might be reflected by the method that is currently running). Moreover, coupling invariants are inherently non-modular, as they rely on implementation details of each component.

Our solution is based on the insight that we can reason modularly about the relation between abstract and concrete state using two separate steps: (1) we use guards to show that a property is *stable*, that is, cannot be invalidated through interference from other nodes while the current node owns certain guards; (2) we introduce *locally inductive invariants* (LIIs) to specify stable properties relating the abstract state and the (local) concrete state.

Since LIIs must be stable w.r.t. the held guards, we associate each LII with a guard. In their full generality, they are reflexive, inductive two-state properties, but we will first explain a simpler, one-state version, where an LII is a predicate over the model state and concrete state, such as `model.a_ctr == ctr` in our example.

A transition block may declare an LII for any of the guards currently held. This LII needs to hold for the current abstract and concrete state. The current abstract state is generally not known precisely, but must have evolved from the abstract state observed at the end of the previous transition block by environment transitions that are not precluded by the guards locally owned.

For example, the first transition block of node A declares the following LII:

```
1 next {
2   invariant(GA) model.a_ctr == ctr
3   a_send(&mut ch, ctr);
4 }
```

Assuming the LII holds when node A starts executing (which is justified in Sec. 2.7), then it will also hold here because (1) the environment can neither perform actions that modify `a_ctr` (since node A holds `GA`) nor modify `ctr` because it is a local variable of node A; (2) node A also does not violate the property, which we check locally during verification. Consequently, the LII provides the relation between abstract and concrete state to prove that this transition block performs the `ASend` action. The same invariant is also used to verify the second transition block of node A.

While *using* an LII to verify a transition block is straightforward, *proving* that the LII holds at the beginning of the block is more involved. Proving properties of an abstract state that evolved

[4]For example, both A-instances may request the number 0; B responds and one A-instance receives the response, updating `a_ctr` to 1; finally, the other A-instance will obtain a timeout and send a 0 again, which violates the specification of `ASend`.

```
requires guard GA
requires last_model(GA).a_ctr = 0
fn node_a(ch: Channel<i32, (i32, i32)>) {
  let mut ctr = 0;
  loop
    invariant guard GA
    invariant last_model(GA).a_ctr = ctr
  {
    // send
    next {
      invariant(GA) model.a_ctr = ctr
      a_send(&mut ch, ctr);
    }
    // wait for response, with timeout
    next {
      invariant(GA) model.a_ctr = ctr
      match a_recv_timeout(&mut ch) {
        Some((n, resp)) if ctr == n => {
          model.a_ctr += 1;
          ctr += 1;
        }
        _ => {}
      }
    }
  }
}
```

```
requires guard GB
requires last_model(GB).b_work = None
fn node_b(ch: Channel<(i32, i32), i32>) {
  loop
    invariant guard GB
    invariant last_model(GB).b_work = None
  {
    // receive (blocking)
    next {
      invariant(GB) model.b_work = None
      let num = b_recv(&mut ch);
      model.b_work = Some(num);
    }
    // compute
    let resp = do_something(num);
    // respond
    next {
      invariant(GB) model.b_work = Some(num)
      b_send(&mut ch, (num, resp));
      model.b_work = None;
    }
  }
}
```

```
fn init_model() {
  init({ (I)
    a_ctr: 0,
    b_work: None,
    a_to_b: [],
    b_to_a: [],
  });
  let (ach, bch) = Channel::create_pair();
  thread::spawn(|| node_a(ach)); (A)
  thread::spawn(|| node_b(bch)); (B)
}
```

Fig. 4. Implementation of nodes A (left) and B (top right), annotated with transition blocks, loop invariants, and method contracts. The initialisation model (bottom right) will be discussed in Sec. 2.7.

by a sequence of environment actions generally requires inductive reasoning, which SMT-based verifiers cannot automate. Therefore, we establish the LII indirectly by proving (1) that it holds in the *current* concrete state, but using the abstract state at the end of the *previous* transition block, and (2) that the LII is stable under environment interference (taking into account the owned guards). Both checks can be performed locally and, together, imply that the LII holds at the beginning of the current transition block.

To enable this verification strategy, we define a variable `last_model` which maps a guard in a given state to the last-seen abstract state, that is, the state at the end of the previous transition block or the initial state. We can then perform check (1) locally, using the current concrete state and the stored abstract state. For check (2), stability, we set up an inductive proof. The base case is the model state recorded in the guard: the LII must hold in that last state. The inductive case performs an arbitrary transition allowed by the system *without* using the guard. If the LII held in an earlier model state and such a transition was taken, then the LII must still hold in the later model state.

Since the last-seen state might come from a different method or loop iteration, we must be able to specify its properties in pre- and postconditions and loop invariants for modular verification to

succeed. Therefore, we allow `last_model` to occur in specifications. For example, the loop invariant preserved by node A is the same as the invariant we used in the transition blocks above, but since we are not inside a transition block, we refer to the model state using `last_model(GA)`. The fully specified node A implementation is shown in Fig. 4 (left), with the loop invariant in Line 7, and a method precondition in Line 2.

In summary, declaring a locally inductive invariant `Inv` for a guard `G` at the beginning of a transition block generates the following proof obligations:

- Check that `G` is owned by the current node.
- (Base case) Check that the invariant holds in the state `last_model(G)`.
- (Inductive case) Check that the invariant is stable:
  $\forall p, n, a. a(p, n) \wedge \neg(a \texttt{ requires G}) \wedge \texttt{Inv}(p) \Rightarrow \texttt{Inv}(n)$
  That is, for every pair of model states p and n, and action a, if a updates p to obtain n and does not require guard G then `Inv` is preserved by the action a.
- Assume that `Inv` holds in the state at the start of the transition block.

These proof obligations use the concrete state only when declaring an LII to ensure that the claimed connection to the abstract state is indeed true. Reasoning about how the concrete state evolves is standard and requires no additional machinery.

At the end of a transition block, `last_model` is updated to the current model state for each owned guard. Note that a node may own multiple guards, and their `last_model` may be different, because the guards were used in different transition blocks. This is also why we specify an LII per guard: different guards can justify different invariants.

Locally inductive invariants are a core contribution of our methodology. As we explained above, they offer a more modular and flexible way to express relations between concrete and abstract states. They also go beyond coupling invariants by supporting reflexive two-state properties to express how abstract states may evolve. For instance, the transition blocks of *node B* could declare the LII `prev_model.a_ctr <= model.a_ctr`, where `prev_model` refers to *any* previous model state in the trace, stating that A's counter must be monotonically increasing, regardless of what node B does. Such properties allow us to prove properties that would otherwise require induction over the abstract transition system. The proof obligations for this generalisation of LIIs change the inductive case to check $\texttt{Inv}(p, n)$ instead of the implication, and also include a check $\forall s.\texttt{Inv}(s, s)$ to ensure that the LII is indeed reflexive.

2.7 Initialisation

To close off our refinement proof, we need to establish that the model state is valid according to the `Init` predicate upon initialisation. In our methodology, we require the user to provide a concrete instance of the model state that satisfies the predicate. This instance is used as the initial state stored (in `last_model`) for every guard of the system. The initialisation takes place before any transition blocks are entered and is denoted with the keyword **`init`**.

In the case of a distributed system, an implementation often consists of multiple nodes that are not necessarily started at the same time. In such cases, there is no single entrypoint (`main` method) where the model would be initialised; instead we can use an *initialisation model*, which serves as a specification of how the model is *conceptually* initialised, and how the guards are distributed across the system. It is analogous to the entrypoint of a multi-threaded program. The initialisation model for the running example is shown in Fig. 4 (bottom right). In applications where the entire system is initialised within one environment, such as multi-threaded programs, the program entrypoint already serves this function, thus an initialisation model is not needed.

This model first ① provides concrete values for the model state (in this case exactly mirroring the equalities in the `Init` predicate). At this point, the model is considered to be initialised and transition blocks may be used beyond this point. The guards `GA` and `GB` are also available. Ⓐ An instance of node A is spawned, which takes ownership of the `GA` guard due to its precondition; Ⓑ analogously for node B, which takes ownership of `GB`.

In general, when the model is initialised with the state `I`, it is checked that `Init(I)` holds. During initialisation, all guards declared in the model are created and their `last_model` is set to `I`. Note that, unlike in approaches with a coupling invariant, there is no requirement for the initial state to be related in any way to the concrete state at the start of the program. This is by design and as discussed previously, allows for greater freedom in how the model state is represented by the implementation. In practice, specifications reflect properties of the initial state in their specifications by referring to guards and `last_model` in method contracts, in particular, to establish LIIs in subsequent transition blocks. We can see this in node A's precondition (Line 2 in Fig. 4), which uses the newly created guard `GA` and its `last_model` to constrain the initial state, and similarly for node B (Line 28 in Fig. 4). If the initial state were wrong, then later LIIs would not be provable.

At this point, we have seen the main ingredients of our refinement methodology. We decomposed the refinement proof into three sub-properties, making use of guards and the novel notion of locally inductive invariants. Our treatment of guards as separation-logic-style [42] resources naturally transfers to Rust's ownership system (→ Sec. 4.3), while LIIs serve as a modular, expressive replacement for coupling invariants. In Sec. 4, we describe the Rust integration of our approach, which addresses various simplifications made in RustRef, and discuss additional features relevant to real-world languages, such as specifying standard library methods.

3 Soundness of LII Reasoning

A full formalisation of our methodology, as integrated in Rust would require a semantics for the Rust language as well as a soundness proof for the underlying program logic (in our case Prusti's). In absence of these prerequisites, we formalise the core part of our methodology (LII reasoning) in the setting of a toy language ToyLang and a standard Hoare logic [20]. Our semantics assumes atomic transition blocks, which we will justify in Sec. 3.1.

The syntax of ToyLang is shown in Fig. B1[5]. This language is less complex than Rust but inherits some of the same invariants, such as the uniqueness of resources (guards), or that thread-local state changes are not observable by the other threads. Much of ToyLang's syntax and operational semantics is standard. Of interest are transition blocks an LIIs: transition blocks start with an acquire operation (`sh_acquire()`) and end with either a release operation (`sh_release()`) or a stutter step (`sh_release_stutter()`). LIIs are declared at the start of a transition block, together with the guard which is used to justify the LII holds: `open` ***g*** `inv` ***E***.

ToyLang is parameterised by the abstract model (see Fig. B3). In particular, we refer to **Action**, the set of actions; **Guard**, the set of guard kinds; ***greq***, indicating which actions require which guards; **ModelState**, the set of model states; Init the initialisation predicate; Next, the step predicate; and ***mspec***, the specifications of any I/O methods. All definitions in this section are implicitly quantified over the model parameters.

ToyLang programs consists of methods, which are executed in a number of parallel threads, starting with a single thread executing the `main` method. At every step of execution, the state of the system is represented by a *configuration* (defined in Fig. B2), a triple Σ consisting of:

- $\mathrm{tm}(\Sigma)$: the thread map, mapping (arbitrary, unique) thread IDs to the state of each thread;
- $\mathrm{tr}(\Sigma)$: the execution trace, representing the sequence of model states encountered thus far;

[5]Figures B1 to B5 are found in Appendix B.

- $\mathrm{sh}(\Sigma)$: the state of the *state handle*: **U** before the model is initialised, **R** after initialisation when no thread is currently within a transition block, and **A** when a thread is within a transition block.

Thread states contained in the thread map are triples consisting of:

- the rest of the program to be executed by the thread;
- the variable store, mapping identifiers to values; and
- the guard map, mapping guards owned by the thread to tuples indicating whether the guard is *open* (only possible within a transition block) and that guard's `last_model`.

Note that we have chosen to embed the ghost state directly in the configuration, to facilitate later proofs. Showing that the ghost state can be erased without affecting the behaviour of the program is straightforward, assuming transition blocks are atomic (which we address in Sec. 3.1).

The initial configuration when executing the program $\boldsymbol{P} \in \mathbf{Program}$ is

$$\Sigma_{\mathrm{Init}} \triangleq ([\boldsymbol{t} \to (\boldsymbol{S}, \{\}, [])], [], \mathbf{U}) \in \mathbf{Conf}$$

where $\boldsymbol{t}$ is the thread ID of the initial thread and $\boldsymbol{S}$ is the statement of the `main` method of $\boldsymbol{P}$. The trace is empty, and the state handle is not initialised.

We define the small-step operational semantics for ToyLang as the relation $\to \subseteq \mathbf{Conf} \times \mathbf{Conf}$. We refer to the transitive closure of this relation as $\to^*$. Fig. B4 shows steps which only affect the local state of one thread. These steps do not affect any other threads, and are thus not observable. These rules are all standard. Fig. B5 shows steps which affect some portion of the shared state or the state handle. We assume a non-side-effectful expression evaluation relation $\to_{\mathrm{E}} \subseteq (\mathbf{Expr} \times \mathbf{Store}) \times \mathbf{Value}$, as well as $\to_{\mathrm{EM}} \subseteq (\mathbf{Expr} \times \mathbf{Store} \times \mathbf{ModelState}) \times \mathbf{Value}$, which can additionally access the `model`.

3.1 Atomicity

Our semantics assumes transition blocks are atomic, i.e., no thread can acquire the state handle using `sh_acquire()` before the current holder (if any) has released it using `sh_release()` or `sh_release_stutter()`. The operations of our methodology are ghost code, i.e., the program must behave identically when the ghost code is erased. There is thus a gap between the actual execution of a transition block (where changes to the model state may be performed across multiple steps) and our semantics (where these changes happen in a single step).

This gap is addressed by Lipton reduction [35], which centres around classifying operations as left-, right-, both-, or non-movers, depending on whether the operations can be reordered (w.r.t. the operations of other threads), while still preserving the overall behaviour of the program. By imposing that the code within a transition block is in the following general shape, we can apply reduction and reason about transition blocks as atomic operations:

- `sh_acquire()` (a right mover),
- zero or more right-mover or both-mover operations,
- zero or one non-mover operations,
- zero or more left-mover or both-mover operations, and
- `sh_release()` (a left mover).

Note that non-I/O method calls and any modifications to the local state are both-movers, since their effects are not observable by other threads, thus the shape primarily constrains the sequence of I/O operations appearing within a block.

As an example, consider the following two ToyLang snippets (similar to Lines 17–21 in Fig. 1):

- `sh_acquire(); x := recv(); _ := send(x); sh_release()`
- `sh_acquire(); x := recv(); _ := send(x); _ := send(x); sh_release()`

With the usual semantics of `send` and `recv` operations on the network and an abstract model where sending messages is modelled by adding (and keeping) the messages in a set of messages, `recv` is a right mover, whereas `send` is a non-mover[6]. As a result, the first snippet can be reduced to a single atomic action, whereas the second one cannot; thus the second snippet would not be a valid transition block in our methodology.

3.2 Soundness of LIIs

We use our formalisation as a basis to show that the proof obligations we have described for LIIs in Sec. 2.6 ensure soundness, i.e., that the invariant can soundly be assumed to hold in the state reached at the beginning of the transition block. We assume a standard proof system based on Hoare rules (with, e.g., rules for loops with invariants). All the rules of the system must be proven sound; such proofs are standard, with the exception of our LII rule, which we discuss next.

Theorem 3.1 (LII soundness). *The Hoare rule:*

$$\frac{\begin{array}{c}\boldsymbol{E}, \sigma, \texttt{last\_model}(\boldsymbol{g}) \rightarrow_{EM} \texttt{true} \\ \forall p, n, a.(\boldsymbol{E}, \sigma, p \rightarrow_{EM} \texttt{true}) \wedge \mathrm{Next}(p, n, a) \wedge (\boldsymbol{g}, a) \notin \boldsymbol{greq} \Rightarrow \boldsymbol{E}, \sigma, n \rightarrow_{EM} \texttt{true}\end{array}}{\Gamma \vdash \{P\}\ \texttt{open}\ \ \boldsymbol{g}\ \ \texttt{inv}\ \ \boldsymbol{E}\ \{P \wedge \boldsymbol{E}, \sigma, \mu \rightarrow_{EM} \texttt{true}\}}$$

is sound, i.e., it suffices to check the two premises to assume that the LII holds in the state at the start of a transition block.

The first premise of the rule corresponds to the base case presented in Sec. 2.6 to check that the invariant holds in the `last_model` of the guard being opened. The second premise corresponds to the inductive step to check that the invariant is preserved across steps which do not use the guard being opened. The postcondition of the Hoare rule ensures that the invariant holds in the current state, i.e., the state at the beginning of the transition block.

To prove this theorem, we make use of a number of lemmata:

Lemma 3.2. *The trace evolves monotonically during execution, i.e., states are only ever appended to the trace and never removed:*

$$\begin{aligned}&\forall \Sigma, \Sigma' \in \mathbf{Conf} \cdot \Sigma \rightarrow \Sigma' \Rightarrow \\ &\quad \mathrm{tr}(\Sigma) = \mathrm{tr}(\Sigma') \vee \exists \mu \in \mathbf{ModelState} \cdot \mathrm{tr}(\Sigma) + [\mu] = \mathrm{tr}(\Sigma')\end{aligned}$$

Proof. We proceed by induction over $\rightarrow$. Cases which modify the trace are Sh.Init and Sh.Rel. In both cases, the existing trace is extended with a new state. □

Lemma 3.3. *The* `last_model` *of every guard* $\boldsymbol{g}$ *corresponds to a state in the execution trace up to that point of execution:*

$$\begin{aligned}&\forall \Sigma \in \mathbf{Conf} \cdot \Sigma_{Init} \rightarrow^* \Sigma \Rightarrow \\ &\quad \forall \boldsymbol{g} \in \mathbf{Guard}, \mu \in \mathbf{ModelState}, \boldsymbol{t} \in \mathbf{ThreadId} \cdot \boldsymbol{g} \rightarrow (\_, \mu) \in \mathrm{tm}(\Sigma)[\boldsymbol{t}] \Rightarrow \\ &\quad\quad \exists i \in \mathbb{N} \cdot \mathrm{tr}(\Sigma)[i] = \mu\end{aligned}$$

Proof. We proceed by induction over $\rightarrow$. Cases which modify `last_model` are Sh.Init and Sh.Rel (note that LII.Op only modifies which guards are opened). Sh.Init sets the `last_model` of every guard of the system to the initial state upon initialisation. Sh.Rel sets the `last_model` of the

[6]Whether `send` is a left mover or a non-mover depends on the network semantics. If `send` sends a message on a pre-established, one-way channel, and if there is no non-blocking receive operation, then `send` can be modelled as a left mover. However, if the recipient can poll for the reception of a message (e.g. it can receive with a timeout or receive on multiple channels simultaneously), then `send` is a non-mover. We choose the latter to more closely align with usual TCP/IP semantics.

open guards held by the current thread to μ', the new state that is also appended to the trace. By Lemma 3.2, these `last_model` states remain in the trace. □

Lemma 3.4. *The* `last_model` *of a guard remains constant, unless it is opened in a transition block.*

Proof. Guards are only opened (the first component in the thread's guard map entry is set to $\top$) in LII.Op, i.e., within a transition block, since LII.Op requires that the current thread has acquired the state handle. The only cases which modify `last_model` (the second component in the thread's guard map entries) are Sh.Rel and Sh.St, both of which update the `last_model` of all open guards to the last state in the trace. □

Lemma 3.5. *Consecutive states in the trace are related by the* Next *predicate:*

$$\forall \Sigma \in \mathbf{Conf} \cdot \Sigma_{Init} \rightarrow^* \Sigma \Rightarrow$$
$$\forall i \in \mathbb{N} \cdot 0 \leq i < |\mathrm{tr}(\Sigma)| - 1 \Rightarrow \exists \boldsymbol{a} \in \mathbf{Action} \cdot \mathit{Next}(\mathrm{tr}(\Sigma)[i], \mathrm{tr}(\Sigma)[i+1], \boldsymbol{a})$$

Proof. We proceed by induction over $\rightarrow$. The only relevant case is Sh.Rel, which requires that in the new trace, the last two states are related by the Next predicate. By Lemma 3.2 the Next relation is preserved for all states in the trace. □

We can now proceed with the proof of Theorem 3.1.

Proof. We start by defining a suffix $\boldsymbol{tr'}$ of the execution trace, corresponding to all the actions taken since $\boldsymbol{g}$ (the guard being opened) was used.

The first element of $\boldsymbol{tr'}$ is the state `last_model(`$\boldsymbol{g}$`)`, i.e., the state resulting from the last action that made use of $\boldsymbol{g}$. This suffix exists, because `last_model(`$\boldsymbol{g}$`)` is an element of the trace by Lemma 3.3.

The model state at the beginning of the transition block is the last element of the trace (by definition of Sh.Acq), thus it is also part of any non-empty suffix we define, including in particular $\boldsymbol{tr'}$. Furthermore, its index in the suffix is greater or equal to the index of `last_model(`$\boldsymbol{g}$`)`.

We show that the LII holds on the last state of $\boldsymbol{tr'}$ by induction over the indices of $\boldsymbol{tr'}$:

Base case ($i = 0$). By the first premise of the Hoare rule, the LII holds in the state $\boldsymbol{tr'}[0]$.

Inductive step ($i > 0$). The inductive hypothesis (IH) tells us the LII holds in the state $\boldsymbol{tr'}[i]$; it remains to show that it holds in $\boldsymbol{tr'}[i+1]$. By Lemma 3.5 we have that $\exists \boldsymbol{a} \in \mathbf{Action} \cdot \mathrm{Next}(\boldsymbol{tr}[i], \boldsymbol{tr}[i+1], \boldsymbol{a})$. By Lemma 3.4 we have that $(\boldsymbol{g}, \boldsymbol{a}) \notin \mathrm{greq}$. By combining the IH and the previous two facts, we can apply the second premise of the Hoare rule to obtain that the LII holds in $\boldsymbol{tr'}[i+1]$. □

At this point, we have shown why our LII reasoning is sound and how Lipton reduction fits into the methodology. The overall property of trace inclusion is straightforward to show in our semantics, given the atomic nature of transition blocks and the initialisation operation.

4 Rust Integration

In this section we will describe how we apply our methodology in Rust. It integrates well with a Rust programmer's workflow because it relies on common Rust idioms: the methodology is provided as a Rust library, users define models by implementing a trait, transition blocks are similar to standard lock critical sections, etc. Our methodology also relies on Rust for some of the reasoning: for example, guard ownership is expressed directly as the ownership of a token type; thus it is checked entirely by Rust's existing borrow checker. Consequently, our methodology is usable in off-the-shelf deductive Rust verifiers, e.g., Creusot [9], Prusti [4], or Verus [31].

The model definition, written by the user, is *ghost code*, i.e., code that is neither compiled into the final binary nor executed at runtime, serving only to express parts of the proof. Although some

of the types forming the Rust interface of our methodology exist in the signature of methods executed at runtime, they are *zero-sized types* (ZSTs), i.e., types that have a 0-byte representation at runtime, allowing the Rust compiler to also eliminate them during compilation. As a result, using our methodology does not incur any costs at runtime.

Our methodology is provided as a Rust library, consisting of trait, type, and method definitions, all of which are equipped with specifications understood by the Rust verifier. We use Prusti [4] and thus all of our specifications are written in its specification language, but the features we make use of are not unique to Prusti. We have not modified Prusti when implementing our methodology compared to its `master` branch, except for requiring support for *obligation resources* to be able to check that every transition block is closed[7]. To use our methodology, the user declares the library as a dependency of the implementation. Some notable exports of our library include the `Model` trait (→ Sec. 4.1), the `StateHandle` type (→ Sec. 4.2), and the `Guard` type (→ Sec. 4.3).

Many types of the library are generic, with the parameter being the type which implements `Model`. This ensures type safety and allows instances of different abstract models to co-exist.

4.1 Model Definition

Our methodology provides the `Model` trait. For a refinement proof, users first need to write a definition of the abstract model by creating an implementation of the `Model` trait, with accompanying definitions for types used as the trait implementation's associated types. Some parts of the trait implementation are optional, although most non-trivial applications will generally provide definitions for all items. The items forming the implementation are:

- the associated type `AbsState`, representing the abstract model state;
- the associated type `EnvState`, representing the observable subset of `AbsState`;
- the associated type `Action`, representing the set of all possible actions;
- the associated type `GuardKind`, representing the set of guards;
- the **`init`** function accepting an abstract state and deciding if it forms a valid initial state;
- the **`next`** function accepting a pair of abstract states and an action and deciding if the former state can transition to the latter state by the given action; and
- the `action_requires_guard` function accepting an action and a guard kind and deciding if the given action can only be taken if a guard of the given kind was opened.

In practice, `AbsState` and `EnvState` are defined as structs, and `Action` and `GuardKind` as enums. All four associated types should be `Copy` types, and thus cannot contain any heap allocations[8].

The Rust definition of the model from Fig. 2 is shown in Fig. 5.

4.2 Initialisation

Any usage of the abstract model, such as when performing transitions, is expressed in Rust using the `StateHandle` type, which we provide. To obtain an instance of this type, the user provides an instance of the abstract state (i.e., of the associated type `AbsState` chosen in the trait implementation) that satisfies the `Init` predicate. This corresponds to the **`init`** statement in RustRef.

```
let handle: StateHandle<Example> = StateHandle::new(ExState {
  a_ctr: 0,
  b_work: None,
  a_to_b: AbsSeq::empty(),
  b_to_a: AbsSeq::empty(),
});
```

[7]This is implemented in Prusti commit `79d48686`, available at https://github.com/viperproject/prusti-dev/pull/1408.
[8]As a result, unbounded data structures like sets or sequences are represented using mathematical types that the verifier can understand but that have no actual memory footprint.

```rust
struct Example;
struct ExState {
  a_ctr: i32,
  b_work: Option<i32>,
  a_to_b: AbsSeq<i32>,
  b_to_a: AbsSeq<(i32, i32)>,
}
struct ExEnvState {
  a_to_b: AbsSeq<i32>,
  b_to_a: AbsSeq<(i32, i32)>,
}
enum ExAction {
  ASend, ARecv, ALoss
  BSend(i32), BRecv,  BLoss,
}
enum ExGuardKind {
  GA, GB,
}
impl Model for Example {
  type AbsState = ExState;
  type EnvState = ExEnvState;
  type Action = ExAction;
  type GuardKind = ExGuardKind;
  fn init(s: ExState) -> bool {
       s.a_ctr == 0
    && s.b_work == None
    && s.a_to_b == AbsSeq::empty()
    && s.b_to_a == AbsSeq::empty()
  }
  fn next(p: ExState, n: ExState, a: ExAction) -> bool {
    match a {
      ExAction::ASend => n == ExState {
        a_to_b: p.a_to_b.concat(a_ctr),
        ..p
      },
      ExAction::ARecv =>
        p.b_to_a.len() > 0 && n == ExState {
          b_to_a: p.b_to_a.tail(),
          a_ctr: (p.b_to_a.head().0 + 1).max(p.a_ctr),
          ..p
        },
      // ..
    }
  }
  fn action_requires_guard(action: Action, guard: ExGuardKind) -> bool {
    match action {
      ExAction::ASend | ExAction::ARecv | ExAction::ALoss => guard == ExGuardKind::GA,
      ExAction::BSend | ExAction::BRecv | ExAction::BLoss => guard == ExGuardKind::GB,
    }
  }
}
```

```
Vars :=
  a_ctr,
  b_work,
  a_to_b,
  b_to_a

Observable :=
  a_to_b,
  b_to_a

Next :=
   ∨ ASend ∨ ARecv ∨ ALoss
   ∨ (∃n. BSend(n)) ∨ BRecv ∨ BLoss

Guards :=
  GA, GB

Init :=
   ∧ a_ctr = 0
   ∧ b_work = None
   ∧ a_to_b = []
   ∧ b_to_a = []

ASend :=
   ∧ a_to_b' = a_to_b ++ [a_ctr]
   ∧ UNCHANGED a_ctr, b_work, b_to_a

ARecv :=
   ∧ |b_to_a| > 0
   ∧ b_to_a' = tail(b_to_a)
   ∧ a_ctr'=max(head(b_to_a)+1,a_ctr)
   ∧ UNCHANGED b_work, a_to_b
```

Fig. 5. Rust version of the model definition from Fig. 2. The corresponding RustRef definitions are shown on the right for comparison. We omit some action definitions, and some boilerplate annotations, such as deriving Copy.

The method `StateHandle::new` is specified to check the proof obligation from Sec. 2.7, i.e.:

```
impl<M: Model> StateHandle<M> {
  #[requires(M::init(state))]
  // ...
  fn new(state: M::AbsState) -> Self;
}
```

The omitted postcondition guarantees that all the guards of the system are available and their `last_model` is initialised to the given initial state.

4.3 Guards

In RustRef, guards were not declared in method signatures; ownership was passed implicitly by using guards in method contracts. In the Rust integration, we use instances of the provided generic `Guard` type to represent guard instances. We rely on the Rust type system and borrow checker to ensure exclusive ownership of guards, because the type is defined as non-`Copy` and non-`Clone`, thus passing it by value (or even by mutable reference) to a method means that method owns it exclusively (temporarily, in the mutable reference case). This approach seamlessly integrates guard ownership with the rest of the code.

A freshly initialised instance of `StateHandle` owns all the guards of the system. They can be removed from it (and later transferred to other methods) with the `StateHandle::take_guard` method, which returns instances of the `Guard` type, but only if that guard was not previously requested:

```
let ga: Guard<Example> = handle.take_guard(ExGuardKind::GA);
```

As an example, node A's implementation takes an instance of the guard in its signature:

```
#[requires(ga.kind() == GuardKind::GA)]
#[requires(ga.last_model().a_ctr == 0)]
fn node_a(ch: Channel<i32, (i32, i32)>, ga: Guard<Example>) { .. }
```

Calling `node_a` thus requires transferring the ownership of the guard. Note that the exact *kind* of the guard is specified in a precondition, because Rust does not support using arbitrary enum instances as type parameters. The `last_model` variable is available as the `Guard::last_model` method.

4.4 Transition Blocks

Transition blocks are represented by two separate operations: an *acquire* and a later *release*[9]. This representation lets one release the invariant with different actions depending on the control flow executed within the transition block. As an example, the second transition block of node A may perform a stutter step, an `ALoss` action, or an `ARecv` action, depending on whether the receive operation timed out or not, and whether it received the response to the last request.

```
handle.acquire();
handle.open_guard_with_invariant(&mut ga, |model| model.a_ctr == ctr); Ⓑ
match a_recv_timeout(&mut ch) {
  Some((n, resp)) if ctr == n => {
    handle.model().a_ctr += 1; Ⓐ
    ctr += 1;
    handle.release(ExAction::ARecv);
  }
  Some(_) => handle.release(ExAction::ALoss),
  None => handle.release_stutter(),
}
```

[9]We borrow terminology from locks, since the invariant can be seen as a ghost lock around the model state.

As seen above Ⓐ, the special RustRef variable `model` is replaced with access using the `StateHandle::model` method, which provides a mutable reference to the model state, but only within the scope of a transition block.

The example also demonstrates Ⓑ how a locally inductive invariant is expressed in the Rust integration. As noted in Sec. 4.3, a mutable reference to a guard serves as a proof that the current node indeed owns that guard, hence the `&mut ga` argument. The guard is *opened* here such that the later `release` operations can perform actions protected by that guard. The second argument of the open call declares the locally inductive invariant, expressed in Rust as a closure over a `model` input.

The methods `StateHandle::release` and `release_stutter` are specified to check the proof obligations described in Sec. 2.4: that the model was modified according to the given action, or left unchanged, respectively. Their postcondition also updates the `Guard::last_model` value to the updated model state. Additional well-formedness conditions are also checked: namely, that transition blocks cannot be nested, and that each `acquire` is followed by a release in a finite amount of steps.

4.5 Atomicity

In Sec. 3.1, we describe how we make use of Lipton reduction to ensure transition blocks are atomic. To automatically check whether each block is reducible, we add a Boolean `has_acted` field to the `StateHandle` type: within a transition block, its value is initially `false`, representing that we are before the non-mover action. We specify whether each I/O operation is a left-mover and/or a right-mover using pre- and postconditions as follows:

- Right-mover: `requires !handle.has_acted`, `ensures !handle.has_acted`
- Non-mover: `requires !handle.has_acted`, `ensures handle.has_acted`
- Left-mover: `ensures handle.has_acted`
- Both-mover: `ensures handle.has_acted == old(handle.has_acted)`

Note that the left-mover specification has no precondition: this accounts for transition blocks which do not contain a non-mover. Statements other than I/O operations need not be treated specially, since they can by definition only operate on local state and are thus both-movers.

Finally, we ensure that, at any point in an execution, each node running in the system has at most a single instance of the `StateHandle` type for a given model. This guarantees that no node can maintain two incompatible views of the system, and use them to derive contradictions.

4.6 Model Composition

Our Rust integration supports model *composition* and *decomposition* using the same mechanism. The key idea is that it is possible to define separate models (consisting of separate implementations of the `Model` trait), then define how they relate to each other using *projections*, where parts of the state and actions of a larger model correspond to a smaller component model (entirely). Here, we focus in particular on how model decomposition can be used to specify standard library methods once and for all, independently of any concrete application of our methodology.

For example, the variable `a_to_b` in the running example defines the state of a one-way communication channel, with the actions `ASend`, `BRecv`, and `BLoss` performing updates to that state that are, in principle, independent of the application itself.

We combine the idea of model projections with Rust's module system to modularly declare reusable components of the system. The latter allows us to define types, such as the state of communication channels, which allow read-only access to their fields (via getters), but forbid external modifications. For example, we can define a `ChannelModel`, representing a transition system for all channel-based communications. It is an implementation of `Model`, and its `AbsState` type is called `ChannelState`:

```
impl Model for ChannelModel {
  type AbsState = ChannelState;
  // ...
}
```

`ChannelState` itself provides a mapping from channel IDs (integers for simplicity) to sequences of arbitrary types, and for convenience also a way to obtain the channel ID (for each channel direction) given a `Channel` instance:

```
impl ChannelState {
  fn packets<T>(&self, channel_id: u32) -> AbsSeq<T> { .. }
  fn id_outgoing<X, Y>(channel: &Channel<X, Y>) -> u32 { .. }
  fn id_incoming<X, Y>(channel: &Channel<X, Y>) -> u32 { .. }
}
```

Many program verifiers support *external specifications* to attach specifications and ghost arguments to methods defined outside the verified codebase, especially library methods. We use such external specifications to connect standard library methods (or external methods in general) with transition systems by equipping them with specifications and a ghost argument of a `StateHandle` type. For example, the send operation is specified as follows:

```
#[extern_spec]
impl Channel<X, Y> {
  #[ensures(handle.model().packets::<T>(ChannelState::id_outgoing(self))
    == old(handle.model().packets::<T>(ChannelState::id_outgoing(self)).concat(value)))]
  fn send<T>(&mut self, value: T, ghost handle: &mut StateHandle<ChannelModel>);
}
```

Finally, our library provides the trait `Project`, which defines how a `StateHandle` of one model type can be projected to another model type and back. For the running example:

```
struct AbsState {
  channels: ChannelState, .. // a_ctr, b_work as before
}
impl Project<ChannelModel> for Example {
  fn get_module(state: ExState) -> ChannelState { state.channels }
  fn put_module(state: ExState, updated: ChannelState) -> ExState {
    ExState { channels: updated, ..state } // other fields unchanged
  }
}
```

The `get_module` and `put_module` methods above return the portion of the state corresponding to the state of all channels[10], and override that portion with an updated value, respectively. The `Project` implementation is then used to obtain temporary references to projected state handles using the `StateHandle::project` method, which is specified to correctly propagate the changes made to the projected model back to the original model:

```
impl<M: Model> StateHandle<M> {
  #[ensures(M::get_module(self.model()) == result.model())]
  #[after_expiry(self.model() ==
    M::put_module(old(self.model(), before_expiry(result.model()))))]
  fn project<P: Model>(&mut self) -> &mut StateHandle<P>
    where M: Project<P>;
}
```

The updated node A implementation then makes use of the methods we defined and the model projection as follows:

[10]This projection system is similar to *optics* or *lenses* used in functional programming.

Table 1. 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. 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 2.40 GHz CPU with 16 GiB of RAM.

Sec.	Program		$\#_C$	$\#_M$	$\#_P$	VT (s)
5.1	Memcached [13]	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 [31]	SeqCst	125	169	247	392.8
		AcqRel	125	169	428	654.5
5.3	Paxos [29, 30]		105	111	205	217.7
5.4	Lock-free hash set [12, 27]		54	134	361	280.6

```
handle.acquire();
handle.open_guard_with_invariant(&mut ga, |model| model.a_ctr == ctr);
ch.send(ctr, handle.project());
handle.release(ExAction::ASend);
```

Using this projection approach we can thus specify external methods independently of any concrete application. Compared to the channels model, the modules we used in Sec. 5 were far more complex, such as the acquire/release atomics module (→ Sec. 5.2), more clearly showing the benefits of a separate codebase of specifications.

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 Table 1.

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. 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, 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 Get, Set, 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 Get, Set, 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:

```
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>>>,
}
```

```
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 invariant, 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 locally inductive invariants used when opening guards are trivial, since the respective parts of the model state are fully determined by local state.

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 bucket is protected by a separate lock. To implement this, we cannot use `HashMap`, and instead need our own implementation. Our hash table is a vector of buckets, each inside a lock. Each bucket contains a linked list of items, storing 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 locally inductive invariant used when opening a `StorageBucket` guard is not a simple equality anymore, but a quantifier:

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

Although the structure of our implementation changed significantly – we added concurrency and fine-grained locking to the initially sequential version – the model remains unchanged, apart from the guard definitions. This shows that our approach enables great program structure flexibility.

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, 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 [31]. This case study demonstrates that our refinement approach extends to various concurrency primitives, including atomics with weak ordering guarantees. We also demonstrate (with the `VerifiedCell` component below) that our refinement methodology is suitable for ensuring safety in the presence of Rust **`unsafe`** code.

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 handles 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 type which allows *interior mutability* (e.g., `Mutex`) 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 [21]. *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 Paxos Consensus Algorithm

We specified and verified a version of the Paxos [29] consensus algorithm, basing our specification on the TLA$^+$ version due to Lamport [30]. Our implementation defines roles for *leaders* and *acceptors* interacting over the same network, where the network is modelled as a *message soup*, i.e., a set of all messages transmitted thus far. The phases of this algorithm are:

- Phase 1A: leaders propose themselves as leaders with increasing ballot counters;
- Phase 1B: acceptors *promise* to vote for leaders with the highest observed ballot counter;
- Phase 2A: a leader accepts the promises given by a quorum of acceptors;
- Phase 2B: acceptors accept the chosen leader.

Different types of messages are sent over the network in each phase of the algorithm. The leader implementation in phase 1A repeatedly sends a message to propose itself as the leader, then waits to see if it receives promises from a quorum of acceptors, triggering phase 2A if so. In this latter step, we use LIIs to remember that certain promises have already been observed on the network. Concretely, the implementation enters a loop to wait for multiple responses, maintaining a set of acceptors and a loop invariant that states that all of these acceptors have a corresponding phase 1B message in the message soup. When entering transition blocks, this set of messages must be a subset of the updated message soup, thus showing that the acceptors' promises have not disappeared.

5.4 Lock-free Hash Set

We specified and verified a lock-free hash-set [27], used in the implementation of the TLC model checker. A similar data structure appeared as a challenge in the VerifyThis 2022 competition [12]. We verified that the implementation refines a high-level model, and that its methods satisfy the specifications of an abstract set data structure. The verification of this data structure made heavy use of atomics and memory order reasoning, already discussed in Sec. 5.2.

6 Related Work

Various approaches [34, 41, 44, 51] develop implementations that are correct by construction by refining abstract models within Rocq and then extracting executable OCaml programs. Similarly, Liu et al. [36] 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 [48] 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 ours. Furthermore, because Trillium is based on Iris and formalised in Rocq, it provides an expressive specification language and enables foundational correctness proofs. We do not claim that our methodology is as expressive as arbitrary Iris invariants, where the full power of the theorem prover is available; on the other hand, proofs in Rocq-based frameworks are not easily automatable and require extensive manual work. Our methodology represents abstract models in first-order logic and automates verification using an SMT solver.

The notion of abstract separation logic-like predicates protecting abstract model state predates Iris and traces back at least to CAP [11] or CaReSL [49]. Our treatment of invariants discussed in Sec. 4 follows this approach; our novelty lies in integrating such an approach with a real-world language and addressing automation.

Compared to our locally inductive invariants, the monotonic-state monad used by F★ [2], must be picked globally on a per-application basis, limiting proof modularity. Our methodology similarly addresses monotonic properties (with the two-state LII generalisation discussed in Sec. 2.6), but also allows properties which are only monotonic as long as a particular guard is not used.

Armada [37] 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 [33] proofs showing a connection between two state machines. Our methodology does not convert programs to state machines and the coupling between the abstract model and the implementation can be much looser. Civl [18, 25] also organises the refinement proof into multiple layers. Each layer is a structured concurrent program, where the concurrent behaviour is reflected in the program structure. This structure simplifies the proof obligations and allows automation, but 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 [46] connects abstract models to concrete implementations via dedicated I/O specifications [40]. Similarly to our work, they support a variety of separation logics to reason about concrete implementations. Their approach requires annotating methods that perform system transitions with suitable specifications, but they do not address how to do so in a reusable, application-independent manner.

Similar to our methodology, IronFleet [17] 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 the program structure. In particular, the programs must be sequential and their structure must mirror the structure of the abstract model.

IronSync [15] and Verus [31] embed the abstract model as ghost state, and, like us, use 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; the user of the methodology provides *localised transition systems*, which are transition systems that model the abstract behaviour of the system with respect only to a subset of the state (protected by owned shards). This transition system can be used directly, because none of its state can be modified by the environment. The system must then be connected to the behaviour of the global transition system, which requires additional proof code, also provided by the user. By contrast, our LIIs perform such reasoning without the need to explicitly define the localised transition systems: by focusing on specific properties when opening a transition block, we can automatically check the stability of such properties.

The refinement technique [24] used in DeepSpec [3] is based on the VST framework [6] for verifying C programs via a separation logic embedded in Rocq. Instead of transition systems, they specify the intended system behaviour using interaction trees [52], 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 [39] embed process calculus models into a concurrent SL, which is automated using Viper [22]. Their refinement approach preserves state assertions, but it is unclear whether arbitrary trace properties are preserved.

7 Conclusion

In this paper, we introduce a novel methodology for refinement proofs of Rust programs that refine an abstract transition system model. The methodology centres around the use of invariants to allow flexible program structure and concurrency. We implemented our approach in Prusti, 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 and liveness properties.

Acknowledgements

This work was funded in part by an Amazon Research Award. We would like to thank Jan Schär; the implementation and evaluation of this work is based in large part on his Master's thesis [43].

Data Availability Statement

The Rust library implementing our methodology and our evaluation, including the case studies and sufficient instructions to reproduce our results as an artefact was submitted as an artefact [5].

References

[1] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theoretical Computer Science* 82, 2 (1991), 253–284. doi:10.1016/0304-3975(91)90224-P

[2] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness: Foundations and Applications of Monotonic State. *Proc. ACM Program. Lang.* 2 (2018), 1–30. Issue POPL. doi:10.1145/3158153

[3] 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. doi:10.1098/rsta.2016.0331

[4] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3 (2019), 1–30. Issue OOPSLA. doi:10.1145/3360573

[5] Aurel Bílý, João Pereira, and Peter Müller. 2025. *A Refinement Methodology for Distributed Programs in Rust (artefact).* doi:10.5281/zenodo.15753818

[6] 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. doi:10.1007/s10817-018-9457-5

[7] Coq Contributors. 1989. *Coq Proof Assistant.* https://coq.inria.fr/

[8] Isabelle Contributors. 1986. *Isabelle Proof Assistant.* https://isabelle.in.tum.de/

[9] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Vol. 13478. Springer International Publishing, Cham, 90–105. doi:10.1007/978-3-031-17244-1_6

[10] Thomas Dinsdale-Young, Pedro Da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper: Automatic Verification for Fine-Grained Concurrency. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Vol. 10201. Springer Berlin Heidelberg, Berlin, Heidelberg, 420–447. doi:10.1007/978-3-662-54434-1_16

[11] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Vol. 6183. Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528. doi:10.1007/978-3-642-14107-2_24

[12] Marie Farrell, Peter Lammich, Marieke Huismann, Rosemary Monahan, Peter Müller, and Mattias Ullbrich. 2022. VerifyThis 2022 Program Verification Competition. (2022). https://www.pm.inf.ethz.ch/research/verifythis/Archive/2022.html

[13] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5. doi:10.5555/1012889.1012894

[14] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (2018-07-09). ACM, Oxford United Kingdom, 442–451. doi:10.1145/3209108.3209174

[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

[16] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015-10-04). ACM, Monterey California, 1–17. doi:10.1145/2815400.2815428

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (2017), 83–92. doi:10.1145/3068608

[18] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Vol. 9207.

Springer International Publishing, Cham, 449–465. doi:10.1007/978-3-319-21668-3_26

[19] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (Aug. 2022), 31 pages. doi:10.1145/3547647

[20] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

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

[22] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Miloš Nováček, Malte Schwerhoff, and Alexander J. Summers. 2014. Viper: A Verification Infrastructure for Permission-Based Reasoning. 21 p. pages. http://hdl.handle.net/20.500.11850/85086

[23] 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. *SIGPLAN Not.* 50, 1 (2015), 637–650. doi:10.1145/2775051.2676980

[24] 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 (CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 234–248. doi:10.1145/3293880.3294106

[25] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2020. Refinement for Structured Concurrent Programs. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Vol. 12224. Springer International Publishing, Cham, 275–298. doi:10.1007/978-3-030-53288-8_14

[26] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems*, Peter Müller (Ed.). Vol. 12075. Springer International Publishing, Cham, 336–365. doi:10.1007/978-3-030-44914-8_13

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

[28] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923. doi:10.1145/177492.177726

[29] Leslie Lamport. 1998. *The Part-Time Parliament.* Association for Computing Machinery. https://dl.acm.org/citation.cfm?id=3335939

[30] Leslie Lamport. 2019. *The Paxos Algorithm or How to Win a Turing Award.* https://lamport.azurewebsites.net/tla/paxos-algorithm.html

[31] 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 (SOSP '24)*. Association for Computing Machinery, Austin, TX, USA, 438–454. doi:10.1145/3694715.3695952

[32] K. Rustan M. Leino and Peter Müller. 2006. A Verification Methodology for Model Fields. In *Programming Languages and Systems*, Peter Sestoft (Ed.). Vol. 3924. Springer Berlin Heidelberg, Berlin, Heidelberg, 115–130. doi:10.1007/11693024_9

[33] K. R. M. Leino and V. Wüstholz. 2014. The Dafny Integrated Development Environment. In *1st Workshop on Formal Integrated Development Environment* (2014) *(Electronic Proceedings in Theoretical Computer Science, Vol. 149)*, Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry (Eds.). Open Publishing Association, 3–15. doi:10.4204/EPTCS.149.2

[34] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 357–370. doi:10.1145/2837614.2837622

[35] Richard J. Lipton. 1975. Reduction: a new method of proving properties of systems of processes. (1975), 78–86. doi:10.1145/512976.512985

[36] 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*, Ritchie Lee, Susmit Jha, Anastasia Mavridou, and Dimitra Giannakopoulou (Eds.). Springer International Publishing, Cham, 22–40. doi:10.1007/978-3-030-55754-6_2

[37] 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* (2020-06-11). ACM, London UK, 197–210. doi:10.1145/3385412.3385971

[38] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (2014), 103–104. doi:10.1145/2692956.2663188

[39] Wytse Oortwijn and Marieke Huisman. 2019. Practical Abstractions for Automated Verification of Message Passing Concurrency. In *Integrated Formal Methods*, Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa (Eds.). Springer

International Publishing, Cham, 399–417. doi:10.1007/978-3-030-34968-4_22

[40] Willem Penninckx, Bart Jacobs, and Frank Piessens. 2015. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–182. doi:10.1007/978-3-662-46669-8_7

[41] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 619–650. doi:10.1007/978-3-319-89884-1_22

[42] J.C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002). IEEE Comput. Soc, Copenhagen, Denmark, 55–74. doi:10.1109/LICS.2002.1029817

[43] Jan Schär. 2023. *Proving Refinement in a Rust Verifier*. Master's thesis. ETH Zurich.

[44] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2 (2018), 1–30. Issue POPL. doi:10.1145/3158116

[45] 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* (2023-10-23). ACM, Koblenz Germany, 113–129. doi:10.1145/3600006.3613172

[46] 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. *Proc. ACM Program. Lang.* 4 (2020), 1–31. Issue OOPSLA. doi:10.1145/3428220

[47] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (2011-09-19). ACM, Tokyo Japan, 266–278. doi:10.1145/2034773.2034811

[48] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8 (2024), 241–272. Issue POPL. doi:10.1145/3632851

[49] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (2013-09-25). ACM, Boston Massachusetts USA, 377–390. doi:10.1145/2500365.2500600

[50] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015-06-03). ACM, Portland OR USA, 357–368. doi:10.1145/2737924.2737958

[51] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. doi:10.1145/2854065.2854081

[52] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (Dec. 2019), 32 pages. doi:10.1145/3371119

Received 2025-03-26; accepted 2025-08-12