Compositional Verification of Concurrency Using Past-Time Temporal Epistemic Logic

Hamed Nemati and Mads Dam

KTH Royal Institute of Technology, Stockholm, Sweden
{mfd, hnnemati}@kth.se

Abstract. Shared-memory concurrency is difficult to reason about because each thread executes under *interference* from other threads. At the same time, many correctness arguments for classic algorithms are *epistemic*: a thread enters a critical region only when, from its local view, it can rule out that another thread is concurrently in that region. We make such arguments explicit by introducing a past-time temporal epistemic logic interpreted over interleaving executions with perfect-recall local histories. Past-time operators support "since" reasoning, while epistemic modalities capture what a given thread can conclude from its own observation history. We give semantics and a sound proof system, instantiate the logic to a simple shared-memory language with instrumented read-/write observations, and illustrate the approach on Peterson's mutual exclusion algorithm by proving a local knowledge condition that implies mutual exclusion.

Keywords: Concurrency · Epistemic logic · Temporal logic · Rely–guarantee · Program verification

1 Introduction

Correctness of shared-memory concurrent programs is notoriously subtle [26,18,25]. Even for safety properties, a proof must account for *interference*: while one thread executes, other threads can change the shared store and invalidate reasoning that would be sound in a sequential setting. The enduring difficulty is to obtain arguments that are simultaneously (i) *local*, so that they scale to realistic code, and (ii) *robust*, so that they imply global correctness properties such as mutual exclusion [27], linearizability [13], or freedom from data races.

A recurring pattern in textbook correctness arguments is *epistemic*: threads make decisions based on what they can or cannot infer from their own actions and observations. In Peterson's mutual exclusion algorithm [27], for example, each thread enters the critical section only after it has ruled out that the other thread will also enter. In lock-free code, a failed *compare-and-swap* provides evidence that some other thread wrote a value [12]. Yet, mainstream assertion languages for concurrent program logics are extensional [26,18,25,19]: they speak about the current global state, but do not directly capture what a particular thread *knows* given its partial view. As a result, proofs often encode epistemic reasoning indirectly, e.g., via delicate global invariants or auxiliary (ghost) state.

In this paper we advocate a lightweight assertion language based on *past-time temporal epistemic logic* [6,10,11,22]. The epistemic modality $\mathbf{K}_A\varphi$ states that φ holds in all runs consistent with thread A's observations; we use a perfect-recall semantics where observations are represented as a prefix of A's local history. Past-time temporal operators (*previously* and *since*) allow us to talk about ordering and persistence of facts [28,22]. Crucially, we derive a *local-time* operator $\mathbf{Prev}_A\varphi$ that refers to the most recent moment before the current one at which A executed a step. This enables thread-centric specifications and proofs that abstract away arbitrarily many steps by other threads.

We instantiate the logic[1] to an interleaving semantics for shared-memory programs where each step is executed by a single thread and performs at most one shared-memory access (compound guards are desugared into read micro-steps). Atomic propositions in the instantiation describe control-flow locations and derived access markers. On top of this, we present a sound, sequent-style proof system combining classical reasoning, a standard S5 basis for knowledge [6], and fixed-point style rules for past-time temporal connectives [22]. We illustrate the approach on Peterson algorithm by proving an epistemic fact about the loop-exit step (expressed using our derived pre-state operator): whenever i enters the critical section, it knows that the loop guard was false in the pre-state of its last step. Combined with stability/ownership obligations, this yields global mutual exclusion; as a corollary, a thread in the critical section knows the other is not.

Beyond this argument, we show how the logic can serve as an assertion language for *compositional* reasoning. In particular, we develop an epistemic variant of rely–guarantee reasoning [18,30,5] in which rely/guarantee conditions are expressed as past-time temporal constraints on environment and component steps, and thread-local knowledge assertions are connected to global invariants via stability obligations. To summarize, we make the following contributions:

- We formalize an interleaving semantics for shared-memory programs that makes local histories explicit and supports perfect-recall epistemic reasoning.
- We define a past-time temporal epistemic logic with a derived *local-time* operator $\mathbf{Prev}_A$ and give a sound proof system.
- We instantiate the logic to a simple shared-memory language with derived read/write access markers and demonstrate the expressiveness via an epistemic proof sketch of Peterson's mutual exclusion.
- We show how the logic can be embedded into a rely–guarantee style proof method, and we use the running Peterson example to illustrate the resulting compositional reasoning obligations.

1.1 Motivation and Running Example

Epistemic view of interference. A thread reasons from a partial view: it can only observe its local state and the values it reads from shared memory. In

[1] Partial mechanization and verification of the base logic in Isabelle/HOL is available at `https://github.com/FMSecure/pttel-theory`

our semantics, the modality $\mathbf{K}_A\phi$ means that ϕ holds at all points that are indistinguishable to thread A given its entire observation history.

Last-step facts and stability. Many local proofs have the following shape. Thread A executes a step, establishes a fact ϕ (typically by reading shared state), and then the environment executes some number of steps. To use ϕ later, A needs an argument that the environment cannot have falsified it. We express "the most recent A-step" via an abbreviation $\mathbf{Prev}_A\,\phi$ (defined in §3) and use it to write

$$\mathsf{est}_A(\phi) \;\triangleq\; \mathbf{Prev}_A\,(\mathbf{K}_A\phi).$$

The key rely–guarantee style reasoning principle we develop is: "if A established ϕ at its last step and ϕ is stable under environmental interference since then, then ϕ holds now."

Running example: We use Peterson's mutual exclusion algorithm for two threads, 0 and 1, as our running example. It uses shared variables `flag[0]`, `flag[1]` $\in \{0,1\}$ and `victim`$\in \{0,1\}$:

Listing 1.1. Peterson's algorithm for thread i (with j = 1-i).

```
flag[i] = 1;
victim  = i;
while (flag[j] == 1 && victim == i) { /* spin */ }
/* critical section */
flag[i] = 0;
```

Our primary safety goal is mutual exclusion: $\mathbf{Always}\,\neg(\mathsf{InCS}_0 \wedge \mathsf{InCS}_1)$, where InCS_i abbreviates "thread i is at its critical-section label". We also use the example to illustrate epistemic properties, e.g., how a thread can justify that a guard it evaluated remains meaningful under permitted interference.

2 Program Model and Observations

We work with a simple interleaving model of shared-variable concurrency. The intent is not to propose a new operational semantics, but to make precise the semantic objects over which temporal and epistemic formulas are interpreted.

States and runs. Fix a finite set of thread identifiers $\mathit{Tid} = \{1, \dots, n\}$. A *global state* s consists of:

1. a shared store $\sigma : \mathit{Var} \to \mathit{Val}$ mapping shared variables to values, and
2. for each thread $A \in \mathit{Tid}$, a local component λ_A containing a control location (program counter) and the values of its local variables.

We write $s.\sigma$ for the shared store and $s.\lambda_A$ for thread A's local component.

The operational semantics induces a labelled transition relation $s \xrightarrow{A} s'$, meaning that s' is obtained from s by executing one atomic step of thread A. We assume *interleaving* (exactly one label per step) and consider infinite *runs* $\pi = s_0 s_1 s_2 \cdots$ where for each $i \geq 0$ there exists a unique thread $\mathsf{act}(i) \in \mathit{Tid}$ with $s_i \xrightarrow{\mathsf{act}(i)} s_{i+1}$.

Scheduling predicate. We interpret A act at position (π, i) as the fact that thread A performs the step from s_i to s_{i+1}, i.e., $\mathsf{act}(i) = A$.

Observations and perfect recall. A key modelling choice for epistemic logic is what each thread can observe. We use the standard choice for shared-memory programs: thread A observes its local component. Formally, the observation function is $\mathsf{obs}_A(s) \triangleq s.\lambda_A$.

Although $\mathsf{obs}_A(s)$ exposes only A's *local* component, this is not a restriction. In a standard small-step semantics, a read from shared memory updates a local register. Hence the *result* of the read becomes part of λ_A, and therefore part of A's observation history. For example, in Peterson's algorithm (Listing 1.1), evaluating the loop guard reads `flag[j]` and `victim`; those values are reflected in the control decision and (in an instrumented semantics) can be recorded in locals. Our logic therefore does not assume that a thread "magically" observes the whole shared store; rather, it reasons from the same information a standard sequential proof would use: local state plus the values returned by reads.

To make "what was read" explicit in the state so that it can be mentioned in atomic predicates and/or used in epistemic postconditions, we assume a lightweight instrumentation of the small-step semantics. Concretely, for each thread A and each shared variable x that A may read, we assume a distinguished local variable lr_x^A ("A's last-read register for x") that is updated on every A-step that reads x. If the pre-state shared store satisfies $\sigma(x) = v$, then the corresponding read step sets $\mathsf{lr}_x^A := v$ in the post-state (in addition to any program-local register updates). Because lr_x^A is part of A's local component λ_A, the value returned by a read is (i) remembered under perfect recall, and (ii) directly addressable by state predicates such as $\mathsf{lr}_x^A = v$. This instrumentation does not add "ghost information" beyond what the operational semantics already provides to the executing thread; it only reifies read results as explicit local state.

When a high-level command (e.g., a compound guard) reads multiple shared variables, we assume it is compiled/desugared into a sequence of micro-steps, each performing at most one shared-memory access and updating the corresponding last-read register lr_x^A. The Peterson case study in §6 uses this standard desugaring.

Asynchrony and "no global clock". Because observations are histories of local states (and we do not assume a global clock is observable), a thread cannot in general distinguish whether *extra* environment steps occurred while its local state stayed the same. This stuttering-insensitivity is essential for modelling what a thread can safely know between its own steps, and it matches the standard asynchronous perfect-recall semantics used in the interpreted-systems literature.

To model *perfect recall*, we compare points in the system by comparing entire observation histories. Given a run π and time i, let $k_1 < \cdots < k_m < i$ be the indices such that $\mathsf{act}(k_j) = A$. Define the (compressed) history

$$\mathsf{Hist}_A(\pi, i) \triangleq \langle \mathsf{obs}_A(s_0),\ \mathsf{obs}_A(s_{k_1+1}),\ \ldots,\ \mathsf{obs}_A(s_{k_m+1}),\ \mathsf{obs}_A(s_i) \rangle.$$

Two points (π, i) and (π', i') are *indistinguishable* to A, written $(\pi, i) \sim_A (\pi', i')$, iff $\mathsf{Hist}_A(\pi, i) = \mathsf{Hist}_A(\pi', i')$. This is the standard epistemic accessibility relation for asynchronous systems with perfect recall.

In addition to the scheduling predicate A act, we interpret atomic propositions as predicates over global states. Concretely, we allow atoms that test the values of shared variables (e.g., $x = v$) and atoms that test the control location of a thread (e.g., A is at ℓ program location, $\mathsf{at}(A, \ell)$). This keeps the logic close to program states while remaining agnostic to a particular instruction set.

3 Past-Time Temporal Epistemic Logic

This section defines the temporal–epistemic language used throughout the paper and gives its semantics over the program model of §2.

3.1 Syntax

We assume a set AP of atomic state predicates p interpreted over global states (e.g., equalities on shared variables and control-location tests such as $\mathsf{at}(A, \ell)$). Formulas are generated as follows and we use $\vee$, $\rightarrow$, and $\leftrightarrow$ as abbreviations:

$$\phi, \psi ::= p \mid A\ \mathsf{act} \mid \neg\phi \mid \phi \wedge \psi \mid \mathbf{Prev}\,\phi \mid \phi\ \mathbf{Since}\ \psi \mid \mathbf{K}_A\,\phi.$$

3.2 State and transition formulas

For interference reasoning it is convenient to separate *state* properties from *step* properties. A *state formula* is an *extensional* predicate: it is built from atomic predicates p using Boolean connectives only (no **Prev**, **Since**, or $\mathbf{K}$). Its truth depends only on the current global state s_i (shared store and local components). In contrast, the scheduling atom A act is a predicate of the transition label $\mathsf{act}(i)$ (which thread executes the *next* step) and is therefore *not* a pure state predicate under our model. Formulas with $\mathbf{K}_A$ are also *intensional*: their truth depends on the entire $\sim_A$-information set induced by A's observation history. A *transition formula* (or *step predicate*) can mention the immediately preceding state using **Prev**. For instance, the formula $(x \neq \mathbf{Prev}\ x)$ holds exactly when the last transition changed the value of x.

As a lightweight notation, we introduce the following step abbreviations:

$$\mathsf{chg}(x) \triangleq (x \neq \mathbf{Prev}\ x) \quad \mathsf{Write}_A(x) \triangleq \mathsf{after}_A \wedge \mathsf{chg}(x)\ \text{ (with } \mathsf{after}_A \text{ defined below).}$$

Because we treat each program step as atomic, $\mathsf{Write}_A(x)$ serves as a coarse event predicate for "the most recent A-step wrote x".

Finally, we occasionally need "last occurrence" reasoning for events other than "last A-step". Given any marker W, we define $\mathsf{Last}_W(\phi) \triangleq \neg W\ \mathbf{Since}\ (W \wedge \phi)$, which is the direct generalization of (1). For example, $\mathsf{Last}_{\mathsf{Write}_A(x)}(x = v)$ states that at the state reached by the most recent A-write to x, the value of x was v.

3.3 Derived operators

We use standard derived past-time operators:

$$\textbf{Always}\ \phi \ \triangleq\ \neg(\top\ \textbf{Since}\ \neg\phi) \qquad \textbf{Sometime}\ \phi \ \triangleq\ \top\ \textbf{Since}\ \phi.$$

Intuitively, **Always** ϕ means that ϕ has held at every state in the run prefix up to the present, while **Sometime** ϕ means that ϕ held at some earlier state (possibly the present).

We also use a "happens-before" abbreviation to express that one condition occurred earlier than another and has not recurred since:

$$\phi \ \prec\ \psi \ \triangleq\ \big(\neg\phi\ \textbf{Since}\ (\psi \wedge \neg\phi)\big)\ \wedge\ \textbf{Sometime}\,\phi.$$

This formula holds when ψ occurred at some time j after which ϕ never held again, and ϕ held at some time before j. It is convenient for expressing properties such as "the last write to x by A occurred before the last write to y by B" once writes are represented as state predicates.

3.4 Last-step operator

Rely–guarantee arguments often use facts established at a thread's most recent step. To express this, we distinguish between the thread scheduled for the *next* transition and the thread that executed the *previous* transition. Define the abbreviation $\mathsf{after}_A \ \triangleq\ \textbf{Prev}\ (A\ \mathsf{act})$, so that after_A holds at time i iff the transition from s_{i-1} to s_i was performed by thread A. We then define the "last A-step" operator used in §1.1:

$$\textbf{Prev}_A\,\phi \ \triangleq\ \neg\mathsf{after}_A\ \textbf{Since}\ (\mathsf{after}_A \wedge \phi). \tag{1}$$

Thus $\textbf{Prev}_A\,\phi$ holds at (π, i) iff ϕ held at the most recent time $j \leq i$ such that after_A held at j (i.e., the most recent state reached by an A-step), and no after_A occurred strictly after j and up to i. If thread A has not yet executed any step, then after_A has never held, and $\textbf{Prev}_A\,\phi$ is false.

We also use the derived abbreviation $\mathsf{pre}_A\,\phi \triangleq \textbf{Prev}_A\,(\textbf{Prev}\,\phi)$, which refers to the *pre-state* of A's last step. In particular, if after_A holds now, then $\mathsf{pre}_A\,\phi$ is equivalent to $\textbf{Prev}\,\phi$.

3.5 Semantics

A model consists of the set of runs induced by the operational semantics (§2) together with the indistinguishability relations $\sim_A$. We define satisfaction $(\pi, i) \models \phi$ inductively:

- $(\pi, i) \models p$ iff p holds of the global state s_i.
- $(\pi, i) \models A\ \mathsf{act}$ iff $\mathsf{act}(i) = A$.
- $(\pi, i) \models \neg\phi$ iff not $(\pi, i) \models \phi$.
- $(\pi, i) \models \phi \wedge \psi$ iff $(\pi, i) \models \phi$ and $(\pi, i) \models \psi$.

- $(\pi, i) \models \mathbf{Prev}\,\phi$ iff $i > 0$ and $(\pi, i-1) \models \phi$.
- $(\pi, i) \models \phi$ **Since** ψ iff there exists j with $0 \le j \le i$ such that $(\pi, j) \models \psi$ and for all k with $j < k \le i$, $(\pi, k) \models \phi$.
- $(\pi, i) \models \mathbf{K}_A\,\phi$ iff for all (π', i') with $(\pi, i) \sim_A (\pi', i')$, we have $(\pi', i') \models \phi$.

As $\sim_A$ is defined by equality of histories, it is an equivalence relation; $\mathbf{K}_A$ satisfies the S5 principles (in particular, truth, positive introspection, and negative introspection). Moreover, if $\mathsf{act}(i) \neq A$ then $\mathsf{Hist}_A(\pi, i) = \mathsf{Hist}_A(\pi, i+1)$, so A's knowledge is unchanged between its own steps.

4 A Sound Proof System

We present a deductive system for the fragment of the logic used in the sequel. The system is intended to support paper-and-pencil proofs (and, ultimately, automation); it is not meant as a complete axiomatization of all validities of temporal–epistemic logic. Soundness is with respect to the semantics of §3.

4.1 Propositional core

A sequent has the form $\Gamma \vdash \phi$, where Γ is a finite set of formulas (assumptions) and ϕ is a formula (conclusion). We write $\bigwedge \Gamma$ for the conjunction of all formulas in Γ.

We assume a standard sound sequent-style proof system for propositional logic (conjunction, disjunction, and negation), including:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \phi \to \psi}{\Gamma \vdash \psi}\ (\textsc{mp}) \qquad \frac{\Gamma, \phi \vdash \psi \qquad \Gamma, \neg\phi \vdash \psi}{\Gamma \vdash \psi}\ (\textsc{lem}\mathrm{E})$$

and the usual introduction/elimination rules for $\wedge$ and $\vee$.

4.2 Past-time operators

The following rules capture the defining fixpoint properties of **Prev** and **Since**.

Previous.

$$\frac{\Gamma \vdash \phi}{\mathbf{Prev}\ \Gamma,\ \mathbf{Prev}\ \top \vdash \mathbf{Prev}\ \phi}\ (\textsc{prev})$$

where $\mathbf{Prev}\ \Gamma \triangleq \{\mathbf{Prev}\ \psi \mid \psi \in \Gamma\}$. The side condition $\mathbf{Prev}\top$ ensures we are not at the initial time; without it, $\vdash \mathbf{Prev}\phi$ would be unsound even when $\vdash \phi$.

Since.

$$\frac{\Gamma \vdash \psi}{\Gamma \vdash \phi\ \mathbf{Since}\ \psi}\ (\textsc{since}\mathrm{I}_1) \qquad \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \mathbf{Prev}(\phi\ \mathbf{Since}\ \psi)}{\Gamma \vdash \phi\ \mathbf{Since}\ \psi}\ (\textsc{since}\mathrm{I}_2)$$

$$\frac{\Gamma \vdash \phi\ \mathbf{Since}\ \psi \qquad \Gamma, \psi \vdash \chi \qquad \Gamma, \mathbf{Prev}(\phi\ \mathbf{Since}\ \psi), \phi \vdash \chi}{\Gamma \vdash \chi}\ (\textsc{since}\mathrm{E})$$

Rule $\textsc{since}\mathrm{I}_1$ corresponds to choosing the witness time $j = i$. Rule $\textsc{since}\mathrm{I}_2$ corresponds to extending an existing witness by one step while maintaining ϕ. Rule $\textsc{since}\mathrm{E}$ is the standard induction principle for **Since**.

We define **Sometime** and **Always** as abbreviations (§ 3); their usual derived rules follow.

4.3 Epistemic operator

The epistemic modality follows the standard S5 principles for each thread A. We present them as sequent rules.

Normality.

$$\frac{\Gamma \vdash \phi}{\mathbf{K}_A \Gamma \vdash \mathbf{K}_A \phi} \ (\mathrm{K})$$

where $\mathbf{K}_A \Gamma \triangleq \{\mathbf{K}_A \psi \mid \psi \in \Gamma\}$.

Truth and introspection.

$$\frac{\Gamma \vdash \mathbf{K}_A \phi}{\Gamma \vdash \phi} \ (\mathrm{T}) \qquad \frac{\Gamma \vdash \mathbf{K}_A \phi}{\Gamma \vdash \mathbf{K}_A \mathbf{K}_A \phi} \ (4) \qquad \frac{\Gamma \vdash \neg \mathbf{K}_A \phi}{\Gamma \vdash \mathbf{K}_A \neg \mathbf{K}_A \phi} \ (5)$$

4.4 Soundness

Theorem 1 (Soundness). *If $\Gamma \vdash \phi$ is derivable, then $\Gamma \models \phi$, i.e., for every model and every point (π, i), if $(\pi, i) \models \bigwedge \Gamma$ then $(\pi, i) \models \phi$.*

Proof (Proof sketch). Soundness of the propositional rules is standard. For $\textsc{prev}$, the premise $\mathbf{Prev}\ \top$ guarantees $i > 0$, so if $(\pi, i) \models \mathbf{Prev}\ \Gamma$ then $(\pi, i-1) \models \Gamma$; by the induction hypothesis $(\pi, i-1) \models \phi$, hence $(\pi, i) \models \mathbf{Prev}\ \phi$. Rules for **Since** are sound by unfolding the semantic definition of **Since** and using the witness time j (for $\textsc{since}\mathrm{I}_1$) or the shifted witness time (for $\textsc{since}\mathrm{I}_2$), and by a standard induction argument for $\textsc{since}\mathrm{E}$. Rules K, T, 4, and 5 are sound because each $\sim_A$ is an equivalence relation, and $\mathbf{K}_A$ is interpreted as universal quantification over $\sim_A$-accessible points.

5 Rely–Guarantee Style Reasoning via Stability

This section shows how a rely–guarantee style reasoning pattern emerges naturally in past-time epistemic logic. The key ingredients are: (i) a way to talk about the state reached by a thread's last step ($\mathbf{Prev}_A\,\cdot$, §3), and (ii) a logical characterization of *stability* under environment steps.

5.1 Step predicates and stability

Because the logic has only past-time operators, we express constraints on the *most recent* transition using **Prev** and the tag $\mathsf{after}_A \triangleq \mathbf{Prev}(A\ \mathsf{act})$. For example, the assertion "if the last step was by thread B, then shared variable x did not change" is expressed as

$$\mathbf{Always}\big(\mathsf{after}_B \to (x = \mathbf{Prev}\ x)\big).$$

Definition 1 (Stability under the environment). *Let $A \in \mathit{Tid}$ and let ϕ be a state formula (in the extensional fragment of §3). We say that ϕ is* stable under the environment of A *if*

$$\mathsf{Stable}_A(\phi)\ \triangleq\ \mathbf{Always}\big(\neg\mathsf{after}_A \to (\mathbf{Prev}\ \phi \to \phi)\big).$$

Intuitively, $\mathsf{Stable}_A(\phi)$ means that any step not performed by A preserves ϕ.

5.2 Algebra of stable assertions

Stability is the logical form of the familiar rely–guarantee side condition "P is stable under the rely". A useful feature of Definition 1 is that stability can be manipulated propositionally, which makes stability checks modular.

Lemma 1 (Closure properties). *For any thread A and state formulas ϕ, ψ:*

1. *If* $\mathsf{Stable}_A(\phi)$ *and* $\mathsf{Stable}_A(\psi)$*, then* $\mathsf{Stable}_A(\phi \wedge \psi)$*.*
2. *If* $\mathsf{Stable}_A(\phi)$ *and* $\mathsf{Stable}_A(\psi)$*, then* $\mathsf{Stable}_A(\phi \vee \psi)$*.*

Proof (Proof sketch). Unfold Definition 1. For (1), use that $\mathbf{Prev}(\phi\wedge\psi)$ is equivalent to $(\mathbf{Prev}\ \phi)\wedge(\mathbf{Prev}\ \psi)$: on an environment step, $\mathbf{Prev}(\phi\wedge\psi)$ implies both $\mathbf{Prev}\ \phi$ and $\mathbf{Prev}\ \psi$, so the two stability assumptions yield ϕ and ψ now. For (2), use that $\mathbf{Prev}(\phi \vee \psi)$ implies $(\mathbf{Prev}\ \phi) \vee (\mathbf{Prev}\ \psi)$ and split cases.

Note that stability is not closed under logical consequence in general; a short counterexample is given in Appendix A.

Frame conditions yield stability. A very common rely condition is a *frame* property saying that the environment does not change some variable x:

$$\mathsf{Frame}_A(x) \triangleq \mathbf{Always}\big(\neg\mathsf{after}_A \to (x = \mathbf{Prev}\ x)\big).$$

From $\mathsf{Frame}_A(x)$ we can derive $\mathsf{Stable}_A(x = v)$ for any value v, and by Lemma 1 we can build stable assertions about tuples of frame-protected variables. In §6 we exploit exactly this pattern for the ownership assumptions on `flag[i]`.

5.3 From last-step facts to current facts

The last-step operator $\mathbf{Prev}_A\, \phi$ is designed so that, if it holds at time i, then there exists a unique time $j \leq i$ that is the most recent state reached by an A-step, and ϕ held at j. Between j and i, all steps are by the environment (i.e., $\neg\mathsf{after}_A$ holds on each intermediate state).

Lemma 2 (Stability lifting). *For any thread A and any state formula ϕ,*

$$\mathsf{Stable}_A(\phi) \;\wedge\; \mathbf{Prev}_A\, \phi \;\models\; \phi.$$

Proof (Proof sketch). Let (π, i) satisfy $\mathsf{Stable}_A(\phi) \wedge \mathbf{Prev}_A\, \phi$, and let $j \leq i$ be the witness time for $\mathbf{Prev}_A\, \phi$ in the semantics of **Since** (cf. (1)). By definition of $\mathbf{Prev}_A\, \phi$, we have $(\pi, j) \models \phi$ and $(\pi, k) \models \neg\mathsf{after}_A$ for all $j < k \leq i$. We show by induction on $k = j, j+1, \ldots, i$ that $(\pi, k) \models \phi$. The base case $k = j$ is immediate. For the step $k \to k+1$ with $k+1 \leq i$, we have $(\pi, k+1) \models \neg\mathsf{after}_A$ and by the induction hypothesis $(\pi, k) \models \phi$, i.e., $(\pi, k+1) \models \mathbf{Prev}\ \phi$. Stability gives $(\pi, k+1) \models \phi$. Thus $(\pi, i) \models \phi$.

5.4 What knowledge buys us: persistence without re-reading

A purely temporal statement $\mathbf{Prev}_A\, \phi$ says that ϕ held at A's last step, but it does *not* say that A is entitled to use ϕ as a justified local assumption. The epistemic modality makes that justification explicit: if $\mathbf{Prev}_A\, (\mathbf{K}_A\phi)$ holds, then A previously had enough observational evidence to rule out all alternatives where ϕ fails.

A key consequence of our asynchronous perfect-recall semantics is that a thread's information set does not change between its own steps. Therefore, if A knew something at its last step, then it still knows it now (even if it has not executed any further instructions).

Lemma 3 (Knowledge persistence between A-steps). *For any thread A and any formula ϕ,*

$$\mathbf{Prev}_A\, (\mathbf{K}_A\phi) \;\models\; \mathbf{K}_A\phi.$$

Proof (Proof sketch). Let $(\pi, i) \models \mathbf{Prev}_A\, (\mathbf{K}_A\phi)$, and let $j \leq i$ be the witness time. By the definition of $\mathbf{Prev}_A\, \cdot$, no A-step occurred strictly between j and i, hence A's local component (and thus its observation history) is unchanged on that interval. Consequently $\mathsf{Hist}_A(\pi, i) = \mathsf{Hist}_A(\pi, j)$ and $(\pi, i) \sim_A (\pi, j)$, so the set of points indistinguishable to A is the same at i and at j. Since A knew ϕ at j, it knows ϕ at i.

Lemma 3 explains why stability conditions are unavoidable: a thread can carry forward *knowledge*, but to carry forward the *truth* of a state formula ϕ it must also argue that the environment cannot falsify ϕ between its own steps.

Objective stability vs. epistemic quantification. $\mathsf{Stable}_A(\phi)$ is evaluated on the *actual* run (every environment step preserves ϕ), whereas $\mathbf{K}_A\phi$ quantifies over *all* points compatible with A's history, which in an asynchronous semantics may include points with extra unobserved environment steps. Accordingly, Corollary 1 concludes only that ϕ is *true now* (it does not claim $\mathbf{K}_A\phi$ now): we use factivity at A's last step to obtain ϕ then, and transport it to the present using objective stability. If one wishes to reason about relies/guarantees *inside* $\mathbf{K}_A$, a common modelling choice is to treat the rely/guarantee clauses as part of the public specification (e.g., add hypotheses $\mathbf{K}_A\mathsf{R}_A$).

5.5 Epistemic rely–guarantee

In practice, a thread often establishes facts via local reasoning about what it knows at its last step. Combining Lemma 2 with factivity of knowledge (the S5 axiom $\mathbf{K}_A\phi \Rightarrow \phi$) yields the following derived rule.

Corollary 1 (Epistemic stability lifting). *For any thread A and any state formula ϕ,* $\mathsf{Stable}_A(\phi) \ \wedge\ \mathbf{Prev}_A\,(\mathbf{K}_A\phi) \ \models\ \phi$.

This corollary can be read as a rely–guarantee principle: “ if thread A established ϕ at its last step (by knowing it then), and ϕ is stable under the environment since that step, then ϕ holds now. ”

5.6 Encoding rely and guarantee conditions

A classic rely–guarantee proof assigns to each thread A a *guarantee* describing what A may do in one step and a *rely* describing what the environment may do. In our setting, a guarantee for A is naturally expressed as a formula of the shape

$$\mathsf{G}_A \ \triangleq\ \mathbf{Always}\big(\mathsf{after}_A \rightarrow G_A^{\mathsf{step}}\big),$$

where G_A^{step} is a predicate over the current state and the previous state (expressible using **Prev**). Similarly, a rely for A may be expressed as

$$\mathsf{R}_A \ \triangleq\ \mathbf{Always}\big(\neg\mathsf{after}_A \rightarrow R_A^{\mathsf{step}}\big).$$

Compatibility of rely/guarantee assumptions becomes logical entailment obligations between these formulas (e.g., $\mathsf{G}_B \models \mathsf{R}_A$ for $B \neq A$ in the two-thread case); the resulting parallel-composition step is summarized in §5.8. We use this encoding in §6 to phrase the standard “ownership” assumptions of Peterson's algorithm (only thread i writes `flag[i]`, and each thread writes `victim` only to its own identifier).

Rely/guarantee assumptions vs. *model constraints.* Rely/guarantee (and ownership) clauses can be used in two distinct ways in epistemic reasoning. *(i) As assumptions about the actual run* (open-system reasoning): the clause is added to the antecedent of an entailment or a local proof obligation but does *not* restrict the epistemic alternatives quantified over by $\mathbf{K}_A$. In this regime, knowledge claims that depend on a rely must explicitly assume that the rely is known (e.g., by adding hypotheses of the form $\mathbf{K}_A \mathsf{R}_A$). *(ii) As constraints defining the model* (closed-program verification): one restricts attention to runs that satisfy a fixed interface specification Φ (typically a conjunction of **Always**-guarded step constraints such as guarantees/ownership). Then $\mathbf{K}_A$ quantifies only over alternatives that also satisfy Φ, and relies may be used inside knowledge without extra $\mathbf{K}$-wrapping. In particular, any formula that holds at all points of the (restricted) model is automatically known by every thread. Unless stated otherwise, our Peterson development treats ownership/guarantee clauses derived from code as model constraints in this sense; we still write them as formulas so they can also be used as explicit assumptions when discussing open-system relies.

5.7 From rely–guarantee obligations to global invariants

A standard use of rely–guarantee is to prove that some global safety property I is an *invariant*. In our past-time setting, the inductive step "I is preserved by each transition" can be stated as the step predicate

$$\mathsf{Pres}(I) \triangleq \mathbf{Always}\big(\mathbf{Prev}\ \top \rightarrow (\mathbf{Prev}\ I \rightarrow I)\big),$$

Saying whenever there is a previous state, if I held previously then it holds now.

Lemma 4 (Invariant from step preservation). *Write* $\mathsf{Init} \triangleq \neg\mathbf{Prev}\ \top$ *for the initial time (the unique point with no predecessor). If I holds at the initial time and* $\mathsf{Pres}(I)$ *holds, then I has always held:*

$$\mathbf{Sometime}(\mathsf{Init} \wedge I)\ \wedge\ \mathsf{Pres}(I)\ \models\ \mathbf{Always}\ I.$$

Proof (Proof sketch). Unfold **Always** I as $\neg(\top$ **Since** $\neg I)$ and argue by contradiction. Assume $\top$ **Since** $\neg I$, and let j be the witness time where $\neg I$ first becomes true. The premise $\mathbf{Sometime}(\mathsf{Init} \wedge I)$ provides the base case I at the initial point. Using $\mathsf{Pres}(I)$ to propagate I forward one step at a time yields I at all times up to j, contradicting $\neg I$ at j. This can be packaged as an instance of the **Since**-induction rule SINCEE (§4).

Compositional preservation checks. To establish $\mathsf{Pres}(I)$ compositionally, we can split by which thread performed the last step. Define

$$\mathsf{Pres}_A(I) \triangleq \mathbf{Always}\big(\mathsf{after}_A \rightarrow (\mathbf{Prev}\ I \rightarrow I)\big).$$

By the interleaving semantics, at every time $i > 0$ exactly one after_A holds, so $\bigwedge_{A \in Tid} \mathsf{Pres}_A(I)$ entails $\mathsf{Pres}(I)$. This is the direct analogue of the classic rely–guarantee proof obligation: " show that each thread preserves I on its own steps, under assumptions about how the environment may have affected the shared state. "

A proof recipe. In the examples, we repeatedly use the following pattern.

1. Choose a global invariant I expressing the safety goal.
2. For each thread A, prove a local step obligation $\mathsf{Pres}_A(I)$ using sequential reasoning about A's code, *assuming* the rely constraints R_A
3. Discharge the rely assumptions by proving compatibility: the guarantees of the other threads imply R_A.
4. Conclude **Always** I by Lemma 4.

The stability-lifting lemma (Lemma 2) is the workhorse for step (2) whenever the local proof needs to carry facts across unboundedly many environment steps, as in spin-wait loops.

5.8 Parallel composition as a derived rule

Classic presentations of rely–guarantee include an explicit *parallel composition* rule. In our setting, interleaving composition is built into the semantics, so the analogous step is a derived meta-theorem stated directly over the trace predicates R_A and G_A.

Write $\mathsf{G}_{-A} \triangleq \bigwedge_{B\in \mathit{Tid}\setminus\{A\}} \mathsf{G}_B$ for the environment guarantee seen by thread A. For any invariant candidate I, the following derived rule packages the usual rely–guarantee side conditions:

$$\frac{\forall A \in \mathit{Tid}.\ \ (\mathsf{G}_{-A} \models \mathsf{R}_A) \qquad \forall A \in \mathit{Tid}.\ \ (\mathsf{G}_A \wedge \mathsf{R}_A) \models \mathsf{Pres}_A(I)}{\left(\bigwedge_{A\in \mathit{Tid}} \mathsf{G}_A\right) \models \mathsf{Pres}(I)}$$

The second premise is the model-relative local proof obligation: $\mathsf{Pres}_A(I)$ is evaluated only at points where after_A holds, so it combines sequential reasoning about A-steps (summarized by G_A) with the rely assumptions. In the two-thread case, this specializes to the familiar compatibility obligations $\mathsf{G}_0 \models \mathsf{R}_1$ and $\mathsf{G}_1 \models \mathsf{R}_0$, together with the local preservation checks $(\mathsf{G}_i \wedge \mathsf{R}_i) \models \mathsf{Pres}_i(I)$. Combined with Lemma 4, this yields the classic rely–guarantee structure: each thread is proved correct under a rely, and the parallel system is correct once each rely is justified by the other threads' guarantees.

6 Case Study: Peterson's Algorithm

We sketch how the temporal–epistemic and stability principles can be used to structure a compositional proof of Peterson's mutual exclusion algorithm).

6.1 Abstract control locations

We assume each thread $i \in \{0, 1\}$ has control locations ℓ^{flag}, ℓ^{victim}, ℓ^{waitF}, ℓ^{waitV}, ℓ^{cs}, ℓ^{exit}. The two waiting locations encode the standard desugaring of the compound guard into read micro-steps (one shared-memory access per step): from ℓ^{waitF} a step reads `flag[j]` (where $j = 1-i$) and enters ℓ^{cs} if it reads 0; otherwise it moves to ℓ^{waitV}, reads `victim`, and enters ℓ^{cs} iff $\texttt{victim} \neq i$ (else loops back). We use the state predicate $\mathsf{at}(i, \ell)$ to test the current location of thread i, and we define

$$\mathsf{InCS}_i \ \triangleq\ \mathsf{at}(i, \ell^{\mathsf{cs}}).$$

6.2 Event predicates for key instructions

To connect the logical proof to program steps, it is convenient to name a few derived *event predicates*. Because control locations are part of the state, we can recognize that a specific instruction just executed by looking at the *previous* location of the active thread.

For thread $i \in \{0, 1\}$, define: $\mathsf{SetFlag}_i \triangleq \mathsf{after}_i \wedge \mathbf{Prev}\,\mathsf{at}(i, \ell^{\mathsf{flag}})$, $\mathsf{SetVictim}_i \triangleq \mathsf{after}_i \wedge \mathbf{Prev}\,\mathsf{at}(i, \ell^{\mathsf{victim}})$, $\mathsf{EnterCS}_i \triangleq \mathsf{after}_i \wedge \mathsf{at}(i, \ell^{\mathsf{cs}}) \wedge \mathbf{Prev}\,(\mathsf{at}(i, \ell^{\mathsf{waitF}}) \vee \mathsf{at}(i, \ell^{\mathsf{waitV}}))$, $\mathsf{ClearFlag}_i \triangleq \mathsf{after}_i \wedge \mathbf{Prev}\,\mathsf{at}(i, \ell^{\mathsf{exit}})$. These are all transition formulas in the sense of §3: each one refers to the immediately preceding state via **Prev**.

Intuitively, $\mathsf{SetFlag}_i$ marks the post-state of executing `flag[i]=1`, $\mathsf{SetVictim}_i$ marks the post-state of executing `victim=i`, and $\mathsf{EnterCS}_i$ marks the post-state of taking the loop-exit transition into the critical section. We use these predicates both as *markers* for last-occurrence reasoning (§3) and as convenient names for thread guarantees.

The operational semantics ensures that each step updates a single thread's location according to the control-flow graph.

6.3 Ownership-style guarantees and derived stability

We encode the standard ownership assumptions of Peterson's algorithm as step properties.

(G_{flag}) Only thread i writes `flag[i]`. For $i \in \{0, 1\}$, define

$$\mathsf{OwnFlag}_i \ \triangleq\ \mathbf{Always}\big(\neg\mathsf{after}_i \rightarrow (\texttt{flag[i]} = \mathbf{Prev}\,\texttt{flag[i]})\big).$$

This implies that facts about `flag[i]` are stable under the environment of thread i. In particular, we obtain $\mathsf{Stable}_i(\texttt{flag[i]} = 1)$ and $\mathsf{Stable}_i(\texttt{flag[i]} = 0)$ as instances of Definition 1.

(G_{victim}) Threads write `victim` only to their own id. We express this at the level of control locations:

$$\mathsf{OwnVictim}_i \ \triangleq\ \mathbf{Always}\Big(\mathsf{after}_i \wedge \mathbf{Prev}\,\mathsf{at}(i, \ell^{\mathsf{victim}}) \rightarrow (\texttt{victim} = i)\Big).$$

This says: if the most recent step was by i and that step executed the `victim=i` command, then the resulting state satisfies $\texttt{victim} = i$.

6.4 Rely/guarantee summary

It is useful to make the rely/guarantee interfaces of Peterson explicit. For each thread i (with $j = 1 - i$), we can view the following as a rely/guarantee pair over shared variables.

Guarantee of thread i. Thread i never writes `flag[j]`, and only writes `victim` when executing `victim=i`:

$$\mathsf{G}_i^{\texttt{flag}} \triangleq \mathbf{Always}\big(\mathsf{after}_i \rightarrow (\texttt{flag[j]} = \mathbf{Prev}\,\texttt{flag[j]})\big),$$

$$\mathsf{G}_i^{\texttt{victim}} \triangleq \mathbf{Always}\Big((\mathsf{after}_i \wedge \neg\mathsf{SetVictim}_i) \rightarrow (\texttt{victim} = \mathbf{Prev}\,\texttt{victim})\Big).$$

(The deterministic postcondition $\mathbf{Always}(\mathsf{SetVictim}_i \rightarrow \texttt{victim} = i)$ complements the second formula.)

Rely of thread i. Dually, thread i relies on the environment (here, thread j) not writing `flag[i]`:

$$\mathsf{R}_i^{\texttt{flag}} \triangleq \mathbf{Always}\big(\neg\mathsf{after}_i \rightarrow (\texttt{flag[i]} = \mathbf{Prev}\,\texttt{flag[i]})\big),$$

which coincides with the ownership-style assumption $\mathsf{OwnFlag}_i$ in the two-thread setting. For `victim`, i relies only on the weak fact that the environment changes it *only* at $\mathsf{SetVictim}_j$, captured by the frame condition introduced above.

Compatibility. The classic rely/guarantee compatibility check "G_j implies R_i" becomes a simple entailment between step predicates. For example, $\mathsf{G}_j^{\texttt{flag}}$ entails $\mathsf{R}_i^{\texttt{flag}}$ because if j never writes `flag[i]` on its own steps, then `flag[i]` is preserved on all non-i steps. This is exactly the sort of modular interference argument that our stability operator $\mathsf{Stable}_A(\cdot)$ is designed to encapsulate.

In a more detailed development one would also record frame conditions describing which variables may change on each step; we keep the presentation lightweight.

6.5 Local guarantees and what a thread can know

Two kinds of facts are used in the standard Peterson argument: (i) *local* facts about what a thread just executed and what branch it took, and (ii) *shared-state* facts that must be shown stable under interference. The temporal–epistemic view makes this split explicit.

Deterministic postconditions of atomic steps. From the sequential semantics of a single step we get simple guarantees such as:

$$\mathbf{Always}(\mathsf{SetFlag}_i \rightarrow \texttt{flag[i]} = 1)$$

$$\mathbf{Always}(\mathsf{ClearFlag}_i \rightarrow \texttt{flag[i]} = 0)$$

$$\mathbf{Always}(\mathsf{SetVictim}_i \rightarrow \mathtt{victim} = i).$$

These are not epistemic statements; they are plain step facts about the operational semantics. In addition, we use the simple frame condition that `victim` changes only at these assignments:

$$\mathbf{Always}\Big(\neg(\mathsf{SetVictim}_0 \vee \mathsf{SetVictim}_1) \rightarrow (\mathtt{victim} = \mathbf{Prev}\,\mathtt{victim})\Big).$$

Why knowing a shared fact requires a rely. Even if thread i just executed `flag[i]=1`, it does *not* automatically follow that i knows $\mathtt{flag[i]} = 1$ at the resulting state under our asynchronous perfect-recall semantics. Intuitively, if the environment were allowed to change `flag[i]` without affecting i's local state, then i cannot rule out that additional environment steps have already occurred since its assignment. Thus $\mathbf{K}_i(\mathtt{flag[i]} = 1)$ becomes valid only once we combine the local postcondition with an interference restriction such as $\mathsf{OwnFlag}_i$. Formally, this can be understood in either of the two ways discussed in §5: (i) as a *model constraint* (so all epistemic alternatives satisfy it), or (ii) as an *assumption* about the actual run together with an explicit hypothesis that the rely itself is known (e.g., $\mathbf{K}_i\mathsf{OwnFlag}_i$).

Knowing a past observation. In contrast, facts about *what the thread observed at its own step* are robust: the past does not change. Under the desugaring above, $\mathsf{EnterCS}_i$ denotes the post-state of the *read micro-step* that exits the spin loop. Such a step can enter ℓ^{cs} only if it observed either $\mathtt{flag[j]} = 0$ or $\mathtt{victim} \neq i$ in its pre-state, hence the compound guard was false there. Because the read result updates i's local component (and thus its observation history), the same sequential justification yields an epistemic guarantee about that pre-state (using $\mathsf{pre}_i \cdot$ from §3):

$$\mathbf{Always}\Big(\mathsf{EnterCS}_i \rightarrow \mathbf{K}_i\, \mathsf{pre}_i\, \neg(\mathtt{flag[j]}{=}1 \wedge \mathtt{victim}{=}i)\Big).$$

This is the form of knowledge we actually need: it records what i learned when it evaluated the guard, rather than asserting knowledge of the current shared-state values.

6.6 Key epistemic/stability reasoning steps

We highlight the two proof obligations where the temporal–epistemic view is most useful.

Carrying forward a local fact. When thread i executes `flag[i]=1`, the post-state satisfies $\mathtt{flag[i]} = 1$ by the deterministic postcondition of the step. Moreover, under the ownership constraint $\mathsf{OwnFlag}_i$ (the environment cannot change `flag[i]` without i noticing), this fact is also something i can *justify* at that step and then keep using between its own steps (Lemma 3). In the closed-program reading where $\mathsf{OwnFlag}_i$ is part of the model, this is captured directly as $\mathbf{K}_i(\texttt{flag[i]} = 1)$ at the post-state of $\mathsf{SetFlag}_i$; in an open-system reading one would additionally assume that the rely itself is known. Using Corollary 1 and $\mathsf{OwnFlag}_i$, we can transport this to later times:

$$\mathsf{OwnFlag}_i \wedge \mathbf{Prev}_i\, \mathbf{K}_i(\texttt{flag[i]} = 1) \;\models\; \texttt{flag[i]} = 1.$$

This is the simplest instance of the rely–guarantee pattern: the guarantee of the environment (it does not write $\texttt{flag[i]}$) provides the stability needed to keep a fact established at the last i-step valid while the environment executes.

The "last entrant" argument. We now make the textbook proof more explicit in the temporal–epistemic language. Fix an arbitrary run π and suppose, for contradiction, that at some time t both threads are in the critical section: $(\pi, t) \models \mathsf{InCS}_0 \wedge \mathsf{InCS}_1$. Let $i \in \{0, 1\}$ be the thread that entered the critical section *last*, and let $j = 1 - i$. Let $e \leq t$ be the entry point of i, characterized by $(\pi, e) \models \mathsf{EnterCS}_i$ and $(\pi, e-1) \models \mathsf{InCS}_j$.

Step 1: j being in the critical section forces $\texttt{flag[j]} = 1$. By control-flow, thread j executes `flag[j]=1` before reaching ℓ^{cs} and only executes `flag[j]=0` after leaving the critical section. This gives the safety-side guarantee

$$\mathbf{Always}(\mathsf{InCS}_j \rightarrow \texttt{flag[j]} = 1).$$

Moreover, by $\mathsf{OwnFlag}_j$ (only j writes $\texttt{flag[j]}$), the fact $\texttt{flag[j]} = 1$ is stable under the environment of j. Hence, at the moment just before i entered (time $e - 1$) we have $(\pi, e-1) \models \texttt{flag[j]} = 1$, and therefore at the entry step we obtain

$$(\pi, e) \models \mathbf{Prev}(\texttt{flag[j]} = 1). \tag{2}$$

Step 2: entering the critical section records what i learned from the loop guard. By the epistemic guarantee for $\mathsf{EnterCS}_i$ from above, at time e thread i knows that in the *pre-state* of its entry step the loop guard was false:

$$(\pi, e) \models \mathbf{K}_i\, \mathsf{pre}_i\, \neg(\texttt{flag[j]}{=}1 \wedge \texttt{victim}{=}i).$$

By epistemic truth (T, §4) we can drop $\mathbf{K}_i$ and obtain $\mathsf{pre}_i\, \neg(\texttt{flag[j]}{=}1 \wedge \texttt{victim}{=}i)$. Since $\mathsf{EnterCS}_i$ implies after_i, $\mathsf{pre}_i \cdot$ coincides with the ordinary previous-state operator here, giving

$$(\pi, e) \models \mathbf{Prev}\big(\neg(\texttt{flag[j]}{=}1 \wedge \texttt{victim}{=}i)\big), \tag{3}$$

Combining (2) and (3) yields

$$(\pi, e) \models \mathbf{Prev}(\texttt{victim} \neq i),$$

and since there are only two threads this simplifies to

$$(\pi, e) \models \mathbf{Prev}(\texttt{victim} = j). \tag{4}$$

Step 3: (4) forces j's victim write to occur after i's. Thread i executes `victim=i` immediately before entering the loop, i.e., $\mathsf{SetVictim}_i$ occurs strictly before $\mathsf{EnterCS}_i$. Equation (4) says that right before the entry step the shared variable `victim` held j. On the other hand, immediately after $\mathsf{SetVictim}_i$ we have $\texttt{victim} = i$ by the deterministic postcondition of `victim=i`. Hence the value of `victim` must have changed from i to j at some time between $\mathsf{SetVictim}_i$ and i's entry. In Peterson's algorithm, the only commands that can change `victim` are the two assignments `victim=0` and `victim=1`, i.e., $\mathsf{SetVictim}_0$ or $\mathsf{SetVictim}_1$. Therefore there exists an occurrence of $\mathsf{SetVictim}_j$ after $\mathsf{SetVictim}_i$, which we summarize using the happens-before abbreviation as

$$\mathsf{SetVictim}_i \prec \mathsf{SetVictim}_j.$$

(If one wishes to make this step fully explicit, it suffices to add a simple frame condition stating that `victim` is unchanged on all steps except $\mathsf{SetVictim}_0$ and $\mathsf{SetVictim}_1$.)

Step 4: the ordering contradicts that j was already in the critical section. Consider the moment when $\mathsf{SetVictim}_j$ occurred. By the structure of the algorithm, thread i had already executed `flag[i]=1` and had not yet executed `flag[i]=0` (since `flag[i]=0` is executed only after leaving the critical section). Using the ownership guarantee $\mathsf{OwnFlag}_i$ and a stability argument (Lemma 2), we obtain that $\texttt{flag[i]} = 1$ still held at the time of $\mathsf{SetVictim}_j$. Therefore, immediately after $\mathsf{SetVictim}_j$, thread j's loop guard $\texttt{flag[i]} == 1 \land \texttt{victim} == j$ was true, so j could not have passed the loop into ℓ^{cs} before i eventually cleared `flag[i]`. This contradicts our assumption that j was already in the critical section at time $e - 1$.

Summary. The formal structure mirrors the informal proof, but the logic forces a clean separation: *knowledge* is used only to justify the pre-state guard fact (3), whereas *stability* (rely/guarantee) transports `flag`-facts across unbounded environment interference. Moreover, once mutual exclusion is established as a global invariant, the advertised epistemic safety property follows as a corollary: since InCS_i is determined by i's local control state, from $\mathbf{Always}\neg(\mathsf{InCS}_0 \land \mathsf{InCS}_1)$ we obtain $\mathbf{Always}(\mathsf{InCS}_i \to \mathbf{K}_i \neg \mathsf{InCS}_j)$.

Discussion. The proof above is the familiar argument for Peterson's algorithm, but the temporal–epistemic presentation isolates two reusable reasoning patterns: (i) carry forward a fact established at the last step using stability, and (ii) separate what a thread can deduce at its own loop-exit point (an epistemic fact) from what remains invariant under environmental steps (a stability fact). This separation is precisely what rely–guarantee proofs enforce by design.

A fully formal proof would add explicit predicates for "thread i just executed line ℓ" and would discharge the local control-flow facts using an ordinary Hoare-style reasoning for the sequential fragment of each thread. Our goal here is to show that the interference reasoning can be expressed and organized cleanly in the temporal–epistemic logic.

7 Related Work

Compositional concurrency reasoning. Owicki–Gries introduced non-interference reasoning for proving assertions about parallel programs by checking that each thread preserves the assertions of the others [26]. Jones' rely–guarantee method made environmental assumptions explicit and remains a foundational technique for modular interference reasoning [18]. These ideas have been refined and mechanized in modern frameworks, including concurrent separation logic (CSL) [2,25,19] and rely–guarantee/separation-logic hybrids such as RGSep [30,7,5,4,29,3].

Our contribution is orthogonal to these program logics: we use past-time epistemic assertions to make thread-local inference explicit, and we package interference control as stability obligations (§5), mirroring the classical 'stable under rely' checks.

Epistemic logic and verification. We adopt the interpreted-systems semantics of knowledge with asynchronous perfect recall [6,11]. Epistemic perspectives on concurrent computations and proof systems for parallel processes were explored early on [15,17,16,20]. Temporal–epistemic logics also have a substantial automated-verification literature and tool support, e.g., MCK and MCMAS [8,24] (see surveys [23]). Knowledge-based specifications are also common in information-flow security, including concurrent settings [1].

Knowledge-based correctness characterizations. Knowledge has been used to characterize consistency conditions such as sequential consistency [21] and linearizability [13] (e.g., [9,14]). These works use knowledge to specify properties of executions as observed by clients or groups of observers, whereas we use (single-thread) knowledge as a proof principle inside compositional safety arguments.

8 Conclusion

We developed a past-time temporal epistemic logic for reasoning about shared-variable concurrent programs under an interleaving semantics with perfect recall. The central technical point is that epistemic statements about what a thread knew at its last step can be lifted to current-state facts when those facts are stable under environmental interference. This yields a clean rely–guarantee style reasoning principle formulated directly in the logic.

Current scope and limitations. Our semantic instantiation uses sequentially-consistent interleaving with single-access micro-steps. This is a deliberate baseline: it isolates the interaction between observation-based knowledge and interference control, but it does not yet address read–modify–write atomics (e.g., CAS), fences, or weak-memory reorderings. Likewise, we interpret formulas over infinite runs; terminating threads can be accommodated by standard stuttering conventions (e.g., self-loops in a final control location), but we have not developed dedicated proof rules for termination/liveness.

Acknowledgments. We thank Robert Künnemann and Yufei Liu for their contributions to the mechanization of the basic logic. This work has been supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

1. Balliu, M., Dam, M., Le Guernic, G.: Epistemic temporal logic for information flow security. In: PLAS 2011. ACM (2011)
2. Brookes, S.D.: A semantics for concurrent separation logic. Theoretical Computer Science **375**(1–3), 227–270 (2007)
3. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M.J., Yang, H.: Views: Compositional reasoning for concurrent programs. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 287–300. ACM (2013)
4. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: European Conference on Object-Oriented Programming (ECOOP). pp. 504–528. Lecture Notes in Computer Science, Springer (2010)
5. Dodds, M., Feng, X., Parkinson, M.J., Vafeiadis, V.: Deny-guarantee reasoning. In: Programming Languages and Systems (ESOP). Lecture Notes in Computer Science, vol. 5502, pp. 363–377. Springer (2009)
6. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press (1995)
7. Feng, X.: Local rely-guarantee reasoning. Tech. Rep. TTIC-TR-2008-1, Toyota Technological Institute at Chicago (Oct 2008)
8. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: CAV 2004. Lecture Notes in Computer Science, vol. 3114, pp. 479–483. Springer (2004)
9. Gleissenthall, K.v., Rybalchenko, A.: An epistemic perspective on consistency of concurrent computations. In: CONCUR 2013. Lecture Notes in Computer Science, vol. 8052, pp. 212–226. Springer (2013)
10. Halpern, J.Y., Moses, Y.: A guide to the modal logics of knowledge and belief: Preliminary draft. In: Joshi, A.K. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985. pp. 480–490. Morgan Kaufmann (1985), `http://ijcai.org/Proceedings/85-1/Papers/094.pdf`
11. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM **37**(3), 549–587 (1990)
12. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
13. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(4), 463–492 (1990)
14. Hirai, Y.: An intuitionistic epistemic logic for sequential consistency on shared memory. In: LPAR-16 2010. Lecture Notes in Computer Science, vol. 6355, pp. 272–289. Springer (2010)
15. van der Hoek, W., van Hulst, M., Meyer, J.J.C.: Towards an epistemic approach to reasoning about concurrent programs. In: REX Workshop 1992: Semantics: Foundations and Applications. pp. 261–287 (1992)

16. van Hulst, M., Meyer, J.J.C.: A knowledge-based compositional proof system for parallel processes. Tech. Rep. UU-CS-1996-19, Utrecht University (1996)
17. van Hulst, M., Meyer, J.J.C.: An epistemic proof system for parallel processes. In: Proceedings of TARK 1994. pp. 243–254. Morgan Kaufmann (1994)
18. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems **5**(4), 596–619 (1983)
19. Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., Dreyer, D.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). pp. 637–650. ACM (2015)
20. Knight, S.: The Epistemic View of Concurrency Theory. Ph.D. thesis, École Polytechnique (2013)
21. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **28**(9), 690–691 (1979)
22. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs. Lecture Notes in Computer Science, vol. 193, pp. 196–218. Springer (1985)
23. Lomuscio, A., Penczek, W.: Symbolic model checking for temporal-epistemic logic. In: Logic Programs, Norms and Action, Lecture Notes in Computer Science, vol. 7360, pp. 172–195. Springer (2012)
24. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: CAV 2009. Lecture Notes in Computer Science, vol. 5643, pp. 682–688. Springer (2009)
25. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science **375**(1–3), 271–307 (2007)
26. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**(4), 319–340 (1976)
27. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters **12**(3), 115–116 (1981)
28. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS). pp. 46–57. IEEE Computer Society (1977)
29. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: European Conference on Object-Oriented Programming (ECOOP). pp. 207–231. Lecture Notes in Computer Science, Springer (2014)
30. Vafeiadis, V., Parkinson, M.: A marriage of rely-guarantee and separation logic. In: CONCUR 2007. Lecture Notes in Computer Science, vol. 4703, pp. 256–271. Springer (2007)

A Appendix: A Simple Normal Form for Past-Time Formulas

This appendix sketches a normal-form transformation for the pure past-time fragment (without knowledge). The goal is to support future automation efforts; it is not required for the soundness results in the main text.

A.1 Eliminating derived operators

Because **Always**ϕ and **Sometime**ϕ are defined from **Since**, they can be eliminated directly:

$$\mathbf{Always}\,\phi \equiv \neg(\top\ \mathbf{Since}\ \neg\phi), \qquad \mathbf{Sometime}\,\phi \equiv \top\ \mathbf{Since}\ \phi.$$

A.2 Fixpoint unfolding for Since

The operator **Since** satisfies the standard unfolding equivalence

$$\phi\ \mathbf{Since}\ \psi\ \equiv\ \psi\ \vee\ (\phi \wedge \mathbf{Prev}(\phi\ \mathbf{Since}\ \psi)).$$

This equivalence is sound under our semantics because at time 0 the **Prev** subformula is false, so the base case reduces to ϕ **Since** $\psi \equiv \psi$ at the initial state, as expected.

A.3 Negation normal form

For the past-time fragment, a standard negation-normal-form (NNF) transformation pushes negations to atomic predicates by using:

$$\neg\mathbf{Prev}\phi \equiv (\mathbf{Prev}\top \wedge \mathbf{Prev}\neg\phi)\ \vee\ \neg\mathbf{Prev}\top,$$

and by rewriting $\neg(\phi$ **Since** $\psi)$ using the unfolding above and De Morgan's laws. In practice, one often avoids expanding $\neg(\phi$ **Since** $\psi)$ eagerly and instead introduces auxiliary predicates and uses induction/recursion (e.g., in a model checker).

A.4 Comment on epistemic operators

A full normal form for the temporal–epistemic logic would need to account for the semantics of $\mathbf{K}_A$ (universal quantification over $\sim_A$-classes), which is orthogonal to the temporal unfolding. One possible direction is to restrict to syntactic fragments where epistemic operators occur only at "local time" points (e.g., immediately after a thread step) and to combine the temporal unfolding with standard S5 reasoning. We leave this to future work.

A.5 Counterexample: stability is not closed under consequence

The closure properties in Lemma 1 hold for Boolean connectives, but stability is *not* monotone w.r.t. logical consequence. For a concrete example, fix a thread A and a shared variable x ranging over integers, and let $\phi \triangleq (x = 0)$ and $\psi \triangleq (x = 0 \vee x = 1)$, so that $\models (\phi \to \psi)$. Consider a run in which thread A never acts and the environment performs a step that changes x from 1 to 2. At that environment step, $\mathbf{Prev}\,\psi$ holds but ψ does not, so $\mathsf{Stable}_A(\psi)$ fails. On the other hand, $\mathsf{Stable}_A(\phi)$ can hold vacuously on the same run if the antecedent $\mathbf{Prev}\,\phi$ is never true on environment steps (e.g., x is never 0 at those points). Thus $\mathsf{Stable}_A(\phi)$ and $\models (\phi \to \psi)$ do not imply $\mathsf{Stable}_A(\psi)$.