Compositional Verification of Concurrency Using Past-Time Temporal Epistemic Logic

Mads Dam and Hamed Nemati

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

Abstract. Reasoning about shared variable concurrent programs poses significant challenges due to the need to account for interference between concurrently executing threads. Traditional verification approaches often fall short in terms of modularity and composability, which are essential for scalable and maintainable verification. We present a method for modular and compositional verification of concurrent programs using *past-time temporal epistemic logic*. Our approach builds on Halpern and Moses' epistemic logic framework and incorporates past-time temporal operators to capture the temporal context of thread interactions. We formalize the semantics of our logic, introduce a compositional proof system for reasoning about concurrent programs, and demonstrate its application. The expressiveness of our proposed logic provides a rigorous foundation to verify concurrent systems compositionally.

Keywords: Epistemic logic · Past-time temporal logic · Concurrency · Formal verification.

1 Introduction

Reasoning about shared-variable concurrent programs is notoriously challenging due to the need to account for interference between concurrently executing threads. Traditional approaches to verifying such programs often struggle with modularity and composability—properties critical for scalable and maintainable verification. While numerous techniques, e.g., [6,10,11,5,1,12,4,26], have been proposed to mitigate this complexity, the potential of epistemic logic—a formalism for reasoning about knowledge—remains largely underexplored in this context. In this paper, we demonstrate how *temporal epistemic logic*, a framework combining **temporal** and **epistemic reasoning**, enables modular and compositional verification of concurrent systems.

We achieve this using *past-time temporal epistemic logic*, a well-established framework in distributed systems for reasoning about knowledge and its evolution over time [2,14,32,3]. Temporal epistemic logic enriches standard temporal logic with epistemic connectives such as "**knows**" (usually depicted by K). These operators allow reasoning about how an agent's (e.g., a program thread's) local state constrains its knowledge of the global system state. Specifically, the assertion "thread A **knows** ϕ" means that ϕ holds in all global states consistent with A's (past) observations. This aligns with the possible-worlds semantics of knowledge, where an agent's knowledge is determined by the set of global states indistinguishable from its local perspective.

To illustrate, consider verifying a concurrent program where threads interact through shared variables. Each thread's actions alter the global state, and synchronization often depends on their evolving knowledge of the system. For instance, a thread may need to assert: "**Thread A knows that thread B has not written to variable x since condition C was met**". Such properties inherently involve both temporal reasoning (e.g., sequencing of events) and epistemic reasoning (e.g., what one thread infers about another's actions). By formalizing such requirements in past-time temporal epistemic logic, we provide a foundation for expressing and verifying synchronization conditions compositionally.

Our work builds on Halpern and Moses' framework [15] for reasoning about distributed processes, augmenting it with past-time temporal operators (e.g., "since", "previously at") to capture the historical context of thread interactions. This allows threads to reference their observational history when establishing knowledge, enabling precise reasoning about interference. For example, a thread can derive that it "**knows** ϕ" by analyzing its own past observations, avoiding global reasoning about all possible interleavings. Such isolation of thread-specific knowledge conditions enables local proofs of correctness that can be combined to infer global system properties, thus achieving modularity.

In this paper, we outline the formal semantics of our logic, describe a proof system for reasoning about concurrent programs, prove its soundness, and demonstrate its application through examples[1]. We additionally show how our framework can be integrated with the rely-guarantee reasoning [23] to specify and verify properties of shared variable concurrent programs in a modular and compositional manner. Overall, leveraging the expressiveness of past-time temporal epistemic logic, our aim is to provide a rigorous foundation for the verification of concurrent systems.

1.1 Related Work

The verification of concurrent programs has been extensively studied, with approaches ranging from separation logic [30] to rely-guarantee [23] reasoning. The PhD theses of Baumann and Bernhard [4,26] give an overview of various techniques used in the verification of concurrent programs, including techniques beyond the scope of epistemic logic. Here, we only focus on those studies that have used epistemic logic for concurrent program verification.

Temporal Logic and Compositionality Temporal logic, introduced by Pnueli [32,27] for reactive systems, has been widely adopted in concurrency. However, traditional linear-time (LTL) and branching-time (CTL) [8] logics lack epistemic

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

modalities, precluding assertions about what threads know based on their local states.

Epistemic Logic in Concurrency Verification Epistemic logic [19] originated as a framework for modeling knowledge in multi-agent systems, with applications in game theory, economics, distributed systems, and artificial intelligence. Halpern and Moses pioneered its use in distributed computing with their seminal work on common knowledge [14,16], which formalizes conditions under which agreement protocols can achieve consensus. Their ideas inspired extensions to probabilistic settings [18], zero-knowledge protocols [17], and broader applications in multi-agent coordination [28,29,31]. Fagin et al. [13] later systematized these concepts in a textbook, making epistemic logic a cornerstone of knowledge representation.

In the concurrency verification domain, there are only a few works that explored the usage of the epistemic logic for reasoning about the correctness of multi-threaded programs, e.g., [24,7,9]. Notable contributions include the work by Chadha, Delaune, and Kremer [7] that proposed an epistemic logic for a variant of the π-calculus that is particularly tailored for modeling cryptographic protocols. Their work focuses on reasoning about epistemic knowledge, especially in the context of security properties such as secrecy and anonymity. Dechesne et al. [9] explored the relation of operational semantics and epistemic logic using labeled transition systems. Similarly, Knight [25] studied the use of epistemic modalities as programming constructs within a process calculus, developed a dynamic epistemic logic for analyzing knowledge evolution in labeled transition systems, and introduced a game semantics for concurrent processes that allows for modeling agents with varying epistemic capabilities.

Also, Van der Hoek et al. [22] contributed to this discourse. Their work extends Halpern et al. [15,16] work on distributed systems to facilitate the verification of concurrent computations using partially ordered sets of action labels. They employed a variant of Hoare's [21] communicating sequential processes (CSP) as a case study to show the application of their theoretical framework.

2 Preliminaries

We begin by introducing the preliminaries required to understand the contribution of this work.

Let $f : A \to B$ be a mapping and $A' \subseteq A$. Then $f \restriction A' : A' \to B$ restricts the domain of f to A'. Also, let $[i, j]$, $i \leq j$, be the interval $\{k \mid i \leq k \leq j\}$.

We denote a shared resource, featureless for now, by $\sigma \in \Sigma$. Later we instantiate it with the shared memory. We denote thread id's (or tid's) by τ and a set of *control states* by *Ctl*. *Thread configurations* $\delta \in \mathit{Config}$ have the shape $\langle \tau, c \rangle$ where $c \in \mathit{Ctl}$. We define a local *next state* as a total function $\mathit{nxt} : \mathit{Config} \times \Sigma \to \mathit{Config} \times \Sigma$, and assume that the this function is not the identity on any command, that is, for all c, σ, if $\mathit{nxt}(c, \sigma) = \langle c', \sigma' \rangle$ then $c \neq c'$. This property allows us to easily detect when a thread performs a computation step and is trivially valid in any of the program models we consider later.

A *state* is a structure $s = \delta_{\tau_1} \parallel \cdots \parallel \delta_{\tau_n} \parallel \sigma \in \mathit{State}$ where $\parallel$ is assumed to be commutative and associative (on configurations), and $\delta_{\tau_i} = \langle \tau_i, c_i \rangle$ such that thread id's are unique, i.e. $i \neq j$ implies $\tau_i \neq \tau_j$. Let $\mathit{ctl}(s, \tau)$ be a function that returns the control state c associated with thread id τ in s. Also, (with a little misuse of notation) let $\sigma(s)$ to extract the shared resource from s. Then, the local transition structure is extended to global states by the condition:

$$\frac{\mathit{nxt}(\delta_{\tau_i}, \sigma) = \langle \delta'_{\tau_i}, \sigma' \rangle}{\delta_{\tau_1} \parallel \cdots \parallel \delta_{\tau_i} \parallel \cdots \parallel \delta_{\tau_n} \parallel \sigma \;\longrightarrow\; \delta_{\tau_1} \parallel \cdots \parallel \delta'_{\tau_i} \parallel \cdots \parallel \delta_{\tau_n} \parallel \sigma'}$$

which represents a *statically parallel asynchronous interleaving* model. Our transition relation determines a set of *computation paths*, or *runs*, $\pi : \omega \to \mathit{State}$, which are all paths generated by the transition relation. For $i \in \omega$, we call the state $\pi(i)$ a *point* of π. Also, we assume for now that the transition relation is total so we do not have to consider terminating runs, and we assume weak fairness so that all threads are scheduled infinitely often along each run. Considering these assumptions, we write all points $\pi(i)$ of a run π in the form $\langle \tau_1, c_{1,i} \rangle \parallel \cdots \parallel \langle \tau_n, c_{n,i} \rangle \parallel \sigma_i$ and we refer to $c_{j,i}$ as $\mathit{ctl}(\pi, \tau_j, i)$ where σ_i is $\sigma(\pi, i)$.

We additionally need to define the concept of a *previous thread state*. To this end, let $i \in \omega$ and $\tau = \tau_k$. Then, thread τ's previous state (relative to the index i) is a run index $j < i$ such that $\mathit{nxt}_\tau(c_{k,j}, \sigma_j) = \langle c_{k,j+1}, \sigma_{j+1} \rangle$ and for all $l : j < l < i$, $c_{k,l} = c_{k,l+1}$. In other words, thread τ's previous state is the execution point at which thread τ last performed a transition.

The observation *history* of thread τ in run π is a pair of mappings $h_{\pi,\tau} : \omega \to \mathit{Ctl}$ and $f : \omega \to \omega$, where f takes run indices to history indices, such that:

1. $h(f(i)) = \mathit{ctl}(\pi(i), \tau)$
2. If $\mathit{ctl}(\pi(i), \tau) = \mathit{ctl}(\pi(i+1), \tau)$ (i.e., τ is not scheduled) then $f(i) = f(i+1)$
3. If $\mathit{ctl}(\pi(i), \tau) \neq \mathit{ctl}(\pi(i+1), \tau)$ then $f(i+1) = f(i) + 1$

The fairness assumption ensures that h is well-defined. We usually elide f and let $\mathit{hist}(\pi, \tau)$ be the history pair $\langle h, f \rangle$. A *point* for a given run π and thread τ is a pair $p = \langle h, i \rangle$ such that h is an observation history of τ in π.

3 Proposed Logic

The logic we consider is discrete bounded past-time epistemic temporal logic and the language is $\mathcal{L}_0$. Thread variables A, B ranging over τ. Let $q \in AP$ be the atomic propositions. The abstract formula syntax of our logic includes the following constructs:

$$\phi, \psi \in \mathit{Prop} ::= q \mid A \textbf{ active} \mid \neg\phi \mid \phi \wedge \psi \mid \textbf{prev } \phi \mid \phi \textbf{ since } \psi \mid A \textbf{ knows } \phi$$

In this context, A **active** is a scheduling predicate indicating that the next transition is performed by thread A. The construct **prev** ϕ denotes that the formula ϕ held at the previous global state. The operator ϕ **since** ψ represents

- $\pi, i \models_\rho q$, if $\pi(i) \in \rho(q)$.
- $\pi, i \models_\rho A$ **active**, if $ctl(\pi(i+1), \rho(A)) \neq ctl(\pi(i), \rho(A))$.
- $\pi, i \models_\rho$ **prev** ϕ, if $i > 0$ and $\pi, i-1 \models_\rho \phi$.
- $\pi, i \models_\rho \phi$ **since** ψ, if for some $j : 0 \leq j \leq i$, $\pi, j \models_\rho \psi$ and for all $k : j < k \leq i$, $\pi, k \models_\rho \phi$.
- $\pi, i \models_\rho A$ **knows** ϕ if, and only if, for all π', i', if $hist(\pi, \rho(A)) = \langle h, f \rangle$, $hist(\pi', \rho(A)) = \langle h', f' \rangle$, and $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$ then $\pi', i' \models_\rho \phi$.

Fig. 1. Satisfaction conditions

the past-time "since" relation, where ψ held at some point in the past, and ϕ has held continuously since then, including the current state. The "since" operator quantifies over the global time axis to express global constraints, while the "previous state" operator quantifies over local time to reason about local thread behavior. Finally, A **knows** ϕ is the labeled epistemic modality saying that ϕ should hold from the local observations of thread A.

Given the operators above, we can derive several others to enhance expressiveness. These include the logical constants $\top$, $\bot$, $\vee$, $\supset$, as well as, **init** which is equivalent to $\neg$**prev** $\top$ and it denotes the initial state.

Temporal operators can also be derived, such as always in the past **always** ϕ, which signifies that ϕ has always held in the past and is defined as $\neg(\top$ **since** $\neg\phi)$. Conversely, **sometime** ϕ ($\equiv \top$ **since** ϕ) indicates that ϕ held at some point in the past. The weak since operator, ϕ $\mathbf{since}_w$ ψ ($\equiv$ (ϕ **since** ψ) $\vee$ (**always** ϕ)), conveys that either ψ held some time in the past and ϕ held since, or else no past state exists not satisfying ϕ.

We also define Lamport's "happens-before" relation, $\phi \prec \psi$ ($\equiv$ ($\neg\phi$ **since** $(\psi \wedge \neg\phi)) \wedge$ **sometime** ϕ), to expresses that ϕ and ψ both happened in the past, and ϕ has not held after the last occurrence of ψ. A weak version of this relation is given by $\phi \preceq \psi$ ($\equiv \neg\phi$ **since** $(\psi \wedge \neg\phi)$).

Additionally, we introduce the concept of a *previous A-state*, which is useful for expressing properties not of the previous global state but of the state preceding the last transition of thread A. This is denoted by **prev** A ϕ ($\equiv$ **prev** ($\neg A$ **active since** ($\phi \wedge A$ **active**))), indicating that sometime in the past, excluding the present state, ϕ held and thread A performed a computation step, with no subsequent steps by thread A.

The weak happens-before expresses that ϕ happens before ψ or else ϕ did not happen at all.

3.1 Semantics

The semantics of our discrete bounded past-time epistemic temporal logic is defined based on a satisfaction relation of the form $\pi, i \models_\rho \phi$, where $i \in \omega$ and ρ is an interpretation mapping (i) thread variables A, B to τ's and (ii) proposition variables q to sets of states. The satisfaction conditions are given in Figure 1.

Proposition 1.

1. *Suppose* $hist(\pi, \rho(A)) = \langle h, f\rangle$ *and* $f(i_1) = f(i_2)$*. If* $\pi, i_1 \models_\rho A$ **knows** ϕ *then* $\pi, i_2 \models_\rho A$ **knows** ϕ*.*
2. $\pi, i \models_\rho A$ **knows** ϕ *if, and only if, for all* π', i'*, if* $hist(\pi, \rho(A)) = \langle h, f\rangle$*,* $hist(\pi', \rho(A)) = \langle h', f'\rangle$*,* $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$*, and* $f'(i') \neq f'(i'+1)$ *then* $\pi', i' \models_\rho \phi$*.*

Proof. For the proof of 1, suppose that $\pi, i_1 \models_\rho A$ **knows** ϕ. Let π', i' be given such that $hist(\pi, \rho(A)) = \langle h, f\rangle$, $hist(\pi', \rho(A)) = \langle h', f'\rangle$, and $h \restriction [0, f(i_2)] = h' \restriction [0, f'(i')]$. But then $h \restriction [0, f(i_1)] = h' \restriction [0, f'(i')]$ and $\pi', i' \models_\rho \phi$.

Moreover, we prove part 2 as follows. Since h is well-defined such an i' exists, and $\pi', i' \models_\rho \phi$ by 1.

Say that A is active at point $\langle \pi, i\rangle$, if $\pi, i \models_\rho A$ **active**. Also, let j be the *previous thread state for* A w.r.t. $\langle \pi, i\rangle$, if A is active in $\langle \pi, j\rangle$ and A is not active in any point $\langle \pi, k\rangle$ for which $j < k < i$. We obtain:

Proposition 2. $\pi, i \models_\rho$ **prev** $A\ \phi$ *if, and only if,* $\pi, j \models_\rho \phi$*, where* j *is the previous thread state for* A *w.r.t.* $\langle \pi, i\rangle$*.*

Proof. Straightforward.

3.2 Inference System

A central result of this paper is an inference system to reason about concurrent threads behavior. We propose a proof system that is both sound and expressive to capture the complexities of concurrent executions and the knowledge of individual threads. We achieve this by developing a Gentzen-style natural deduction proof system, which enables direct proof.

The inference system is built on sequents. In our system, sequents are of the form $\Gamma \vdash \phi$, where the antecedent Γ denotes a set of propositions and $\vdash$ represents the deductive relation. We denote the semantics of the judgments by $\Gamma \models \phi$, which assert that if for all models and all π, i, ρ, if $\pi, i \models_\rho \bigwedge \Gamma$ then $\pi, i \models_\rho \phi$.

Some of the rules in our proof system are standard, but we include them for completeness:

$$\wedge\text{I}\ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \quad \wedge\text{E}\textsc{l}\ \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_1} \quad \wedge\text{E}\textsc{r}\ \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_2} \quad \vee\text{I}\textsc{l}\ \frac{\Gamma \vdash \phi_1}{\Gamma \vdash \phi_1 \vee \phi_2}$$

$$\vee\text{I}\textsc{r}\ \frac{\Gamma \vdash \phi_2}{\Gamma \vdash \phi_2 \vee \phi_2} \quad \vee\text{E}\ \frac{\Gamma \vdash \phi_1 \vee \phi_2 \qquad \Gamma, \phi_1 \vdash \psi \qquad \Gamma, \phi_2 \vdash \psi}{\Gamma \vdash \psi}$$

$$\neg\text{I}\ \frac{\Gamma, \phi \vdash \bot}{\Gamma \vdash \neg\phi} \quad \neg\text{E}\ \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \neg\phi}{\Gamma \vdash \psi} \quad \neg\neg\ \frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi}$$

The more interesting rules are as follows:

$$\textsc{Active}\text{I}\ \frac{-}{\Gamma \vdash \bigvee_{thread\ A} A\ \textbf{active}}$$

$$\textsc{ActiveE}\ \frac{\Gamma \vdash A\ \textbf{active} \qquad \Gamma \vdash B\ \textbf{active}}{\Gamma \vdash \phi}\ (A \neq B \text{ and not mapped to same } \tau)$$

$$\textsc{Prev}\ \frac{\Gamma \vdash \phi}{\textbf{prev}\ \Gamma \vdash \textbf{prev}\ \phi} \quad \mathrm{K}\ \frac{\Gamma \vdash \phi}{A\ \textbf{knows}\ \Gamma \vdash A\ \textbf{knows}\ \phi}$$

$$\mathrm{T}\ \frac{\Gamma \vdash A\ \textbf{knows}\ \phi}{\Gamma \vdash \phi} \quad 4\ \frac{A\ \textbf{knows}\ \Gamma \vdash \phi}{A\ \textbf{knows}\ \Gamma \vdash A\ \textbf{knows}\ \phi}$$

$$\mathrm{SI1}\ \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi\ \textbf{since}\ \psi} \quad \mathrm{SI2}\ \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \textbf{prev}\ (\phi\ \textbf{since}\ \psi)}{\Gamma \vdash \phi\ \textbf{since}\ \psi}$$

$$\mathrm{SE}\ \frac{\Gamma \vdash \phi_1\ \textbf{since}\ \phi_2 \qquad \Gamma, \phi_2 \vdash \psi \qquad \Gamma, \textbf{prev}\ (\phi_1\ \textbf{since}\ \phi_2) \vdash \psi}{\Gamma \vdash \psi}$$

$$\mathrm{KP1}\ \frac{\Gamma \vdash \textbf{prev}\ A\ \textbf{active} \qquad \Gamma \vdash \textbf{prev}\ A\ \textbf{knows}\ \phi}{\Gamma \vdash A\ \textbf{knows}\ \textbf{prev}\ (A\ \textbf{active}\ \supset \phi)}$$

$$\mathrm{KP2}\ \frac{\Gamma \vdash \textbf{prev}\ \neg A\ \textbf{active} \qquad \Gamma \vdash \textbf{prev}\ A\ \textbf{knows}\ \phi}{\Gamma \vdash A\ \textbf{knows}\ \textbf{prev}\ (\neg A\ \textbf{active}\ \supset \phi)}$$

$$\mathrm{KPA}\ \frac{\Gamma \vdash \textbf{prev}\ A\ (A\ \textbf{knows}\ \phi)}{\Gamma \vdash A\ \textbf{knows}\ \textbf{prev}\ A\ \phi}$$

$$\mathrm{KSRA}\ \frac{\Gamma \vdash \textbf{prev}\ A\ A\ \textbf{knows}\ (\phi\ \textbf{since}\ \psi) \qquad \Gamma \vdash A\ \textbf{knows}\ \phi}{\Gamma \vdash A\ \textbf{knows}\ (\phi\ \textbf{since}\ \psi)}$$

Our proof system does not capture complete S5 properties of K, for practical purposes. We only focus on those, particularly T and K, which are important for sound reasoning about knowledge.

Theorem 1 (Soundness). *If $\Gamma \vdash \Delta$ is derivable then $\Gamma \models \Delta$.*

Proof. Case **Prev**: If $\Gamma \models \phi$ and $\pi, i \models_\rho \textbf{prev}\ \psi$ for all $\psi \in \Gamma$ then for all $\psi \in \Gamma$, $\pi, i-1 \models_\rho \psi$. We can conclude that $\pi, i-1 \models_\rho \phi$ so $\pi, i \models_\rho \textbf{prev}\ \phi$.

Case **4**: (Standard) Suppose $A\ \textbf{knows}\ \Gamma \models \phi$ and $\pi, i \models A\ \textbf{knows}\ \Gamma$. Then for all π', i' satisfying the appropriate conditions, $\pi', i' \models_\rho A\ \textbf{knows}\ \Gamma$ as well, so $\pi', i' \models_\rho \phi$, hence $\pi, i \models_\rho A\ \textbf{knows}\ \phi$.

Case **SI1, SI2, SE**: Standard.

Case **T**: Standard. If $\pi, i \models_\rho A\ \textbf{knows}\ \phi$ then $\pi, i \models_\rho \phi$.

Case **KP1**: Assume $\Gamma \models \textbf{prev}\ A\ \textbf{active}$, $\Gamma \models \textbf{prev}\ A\ \textbf{knows}\ \phi$ and that $\pi, i \models_\rho \bigwedge \Gamma$. Then for $i > 0$, pick π', i' such that $\mathit{hist}(\pi, \rho(A)) = \langle h, f \rangle$, $\mathit{hist}(\pi', \rho(A)) = \langle h', f' \rangle$ and $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$. Also assume $\pi', i'-1 \models_\rho A\ \textbf{active}$. We know that $\pi, i \models_\rho A\ \textbf{active}$ as well. Then $h \restriction [0, f(i-1)] = h' \restriction [0, f'(i'-1)]$, so $\pi', i'-1 \models_\rho \phi$, concluding the case.

Case **KP2**: Similar.

Case **KPA**: Assume $\pi, i \models_\rho \textbf{prev}\ A\ A\ \textbf{knows}\ \phi$. Let π', i' be given such that $hist(\pi, \rho(A)) = \langle h, f\rangle$, $hist(\pi', \rho(A)) = \langle h', f'\rangle$ and $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$. By proposition 2, $\pi, j \models_\rho A\ \textbf{knows}\ \phi$, where $j < i$. Since $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$ we find $j' < i'$ such that $\pi', j' \models_\rho \phi$. Also, for all $k' : j' < k' < i'$m, $\rho(A)$ is not active. Hence $\pi', i' \models_\rho \textbf{prev}\ A\ \phi$, and hence $\pi, i \models_\rho A\ \textbf{knows prev}\ A\ \phi$.

Case **KSRA**: Assume $\Gamma \models \textbf{prev}\ A\ A\ \textbf{knows}\ (\phi\ \textbf{since}\ \psi)$, $\Gamma \models A\ \textbf{knows}\ \phi$, and $\pi, i \models_\rho \bigwedge \Gamma$. Then $\pi, i \models_\rho \textbf{prev}\ A\ A\ \textbf{knows}\ (\phi\ \textbf{since}\ \psi)$ and $\pi, i \models_\rho A\ \textbf{knows}\ \phi$. Let π', i' be given such that $hist(\pi, \rho(A)) = \langle h, f\rangle$, $hist(\pi', \rho(A)) = \langle h', f'\rangle$, and $h \restriction [0, f(i)] = h' \restriction [0, f'(i')]$. We can immediately conclude that $\pi', i' \models_\rho \phi$, and hence $\pi', i'' \models_\rho \phi$ whenever $f'(i') = f'(i'')$. We can also conclude that $\pi, j \models_\rho A\ \textbf{knows}\ (\phi\ \textbf{since}\ \psi)$ where j is the previous thread state relative to i for $\rho(A)$. Then $\pi', j' \models_\rho (A)\phi\ \textbf{since}\ \psi$ for all j' such that $f'(j') + 1 = f(i')$. But then it follows that $\pi', i' \models_\rho \phi\ \textbf{since}\ \psi$ as was to be proved.

3.3 Derived Rules

We then derive additional rules for *temporal operators* that extend our basic proof system. In particular, we use the basic axioms and inference rules from the previous section to derive rules for the “always” and “sometime” operators, as well as “since” and “weak since”.

$$alwaysI\ \frac{\Gamma \vdash \phi \qquad \Gamma \vdash \textbf{prev always}\ \phi}{\Gamma \vdash \textbf{always}\ \phi} \quad alwaysE1\ \frac{\Gamma \vdash \textbf{always}\ \phi}{\Gamma \vdash \phi}$$

$$alwaysE2\ \frac{\Gamma \vdash \textbf{always}\ \phi}{\Gamma \vdash \textbf{prev always}\ \phi} \quad sometimeI1\ \frac{\Gamma \vdash \phi}{\Gamma \vdash \textbf{sometime}\ \phi}$$

$$sometimeI2\ \frac{\Gamma \vdash \textbf{prev sometime}\ \phi}{\Gamma \vdash \textbf{sometime}\ \phi}$$

$$sometimeE\ \frac{\Gamma \vdash \textbf{sometime}\ \phi \qquad \Gamma, \phi \vdash \psi \qquad \Gamma, \textbf{prev sometime}\ \phi \vdash \psi}{\Gamma \vdash \psi}$$

$$weaksinceI1\ \frac{\Gamma \vdash \psi \quad \Gamma \vdash \psi \supset \phi}{\Gamma \vdash \phi\ \textbf{since}_w\ \psi} \quad weaksinceI2\ \frac{\Gamma \vdash \phi \quad \Gamma \vdash \textbf{prev}\ (\phi\ \textbf{since}_w\ \psi)}{\Gamma \vdash \phi\ \textbf{since}_w\ \psi}$$

$$weaksinceE\ \frac{\Gamma \vdash \phi_1\ \textbf{since}_w\ \phi_2 \quad \Gamma, \phi_2 \vdash \psi \quad \Gamma, \textbf{prev}\ (\phi_1\ \textbf{since}_w\ \phi_2) \vdash \psi}{\Gamma \vdash \psi}$$

$$hbI1\ \frac{\Gamma \vdash \phi_2 \qquad \Gamma \vdash \neg\phi_1 \qquad \Gamma \vdash \textbf{sometime}\ \phi_1}{\gamma \vdash \phi_1 \prec \phi_2}$$

$$hbI2\ \frac{\Gamma \vdash \neg\phi_1 \qquad \Gamma \vdash \textbf{prev}\ (\phi_1 \prec \phi_2)}{\gamma \vdash \phi_1 \prec \phi_2}$$

$$hbE\ \frac{\Gamma \vdash \phi_1 \prec \phi_2 \quad \Gamma, \phi_2, \neg\phi_1, \textbf{sometime}\ \phi_1 \vdash \psi \quad \Gamma, \neg\phi_1, \textbf{prev}\ (\phi_1 \prec \phi_2) \vdash \psi}{\Gamma \vdash \psi}$$

4 Program Model

Having introduced the basic proof system, we now move on to instantiate this general framework to a specific model, i.e., for a statically parallel structured assembly-like language $\mathcal{P}_1$, defined by the following abstract syntax:

$$\begin{aligned} Rn \in \mathit{regs} &::= R\{i\} \text{ for } 0 \leq i \leq 31 \\ e \in \mathit{PExp} &::= w \mid !Rn \mid Rn \mid Rn \texttt{ op } Rn \\ atm \in \mathit{ACmd} &::= Rn \ \lhd \ e \mid Rn \ \rhd \ Rn \\ c \in \mathit{Cmd} &::= atm \mid c; c \mid \texttt{if } Rn\ c\ c \mid \texttt{while } Rn\ \{c\} \end{aligned}$$

where $w \in \mathit{Word}$ is a word representing a constant value, Rn denotes the register $\#n$, ! is used for pointer dereferencing, and $\texttt{op}$ represents binary operations (left open). Atomic commands in ACmd correspond to move, load constant, and memory load and store. The primary intention is to ensure that atomic commands correspond to, at most, one memory access. Branching tests on values in the register Rn. Commands are labeled consecutively by labels $l \in \mathit{\Lambda}$, and we write $l : c$ if c is labeled l. The initial label for thread τ is $l_{\tau,0}$.

We give a rewrite semantics of static thread pools and map it into the model structure introduced in Section 2. The shared resource is now a store of the shape $\sigma : \mathit{\Sigma} = \mathit{Word} \to \mathit{Word}$ along with an event α recording the most recent memory access, if any, and which thread performed it:

$$\alpha \in E ::= \varepsilon \mid A : w_1 \lhd w_2 \mid A : w_1 \rhd w_2$$

The events provide information needed for a specification that cannot be reliably inferred from other parts of the control state. For example, a transition by a thread A may read w_1 from w_2 and assign w_1 to $R3$. There is no information in the control state outside the event itself to indicate that this control state change was due to a read and not some other computation step, and this information can be crucial for verification.

States now have the shape $s = \delta_{\tau_1} \parallel \cdots \parallel \delta_{\tau_n} \parallel \sigma \parallel \alpha$. Configurations have the shape $\delta = \langle \tau, c, r \rangle$ where the control state now consists of a command c with a register assignment $r : Rn \mapsto w \in \mathit{Word}$ of words to registers Rn. With s and δ as above we let $\alpha(s) = \alpha$, $\sigma(s) = \sigma$, $\mathit{cmd}(s, \tau) = c$, and $\mathit{regs}(s, \tau) = r$.

The local transition structure is defined in Figure 2. It is rather straightforward, despite the symbol pushing, and assumes that ";" is associative, for simplicity.

For the axiomatization, we will introduce a couple of helper functions later. First, a (partial) function that extracts the branching condition, when it exists:

$$\mathit{cond}(c', c) = \begin{cases} \top & \text{if } c' = \alpha; c \\ Rn \neq 0 & \text{if } c' = \texttt{if } Rn\ c_1\ c_2; c'' \text{ and } c = c_1; c'' \\ Rn = 0 & \text{if } c' = \texttt{if } Rn\ c_1\ c_2; c'' \text{ and } c = c_2; c'' \\ Rn \neq 0 & \text{if } c' = \texttt{while } Rn\ c_1; c'' \text{ and } c = c_1; \texttt{while } Rn\ c_1; c'' \\ Rn = 0 & \text{if } c' = \texttt{while } Rn\ c_1; c'' \text{ and } c = c'' \\ \bot & \text{otherwise} \end{cases}$$

$$
\begin{aligned}
&nxt(\langle \tau, Rn \ \lhd \ w; c, r\rangle, \sigma \parallel \alpha) \ \ = \langle\langle \tau, c, r[Rn \mapsto w]\rangle, \sigma \parallel \varepsilon\rangle \\
&nxt(\langle \tau, Rn \lhd {!Rm}; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, Rn \lhd \sigma(r(Rm)); c, r\rangle, \sigma \parallel \tau : Rn \lhd \sigma(r(Rm))\rangle \\
&nxt(\langle \tau, Rn \lhd Rm; c, r\rangle, \sigma \parallel \alpha) \ \ = \langle\langle \tau, Rn \lhd r(Rm); c, r\rangle, \sigma \parallel \varepsilon\rangle \\
&nxt(\langle \tau, Rn \lhd Rm \ \mathtt{op} \ Rk; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, Rn \lhd r(Rm) \ \mathtt{op} \ r(Rk)); c, r\rangle, \sigma \parallel \varepsilon\rangle \\
&nxt(\langle \tau, Rn \rhd Rm; c, r\rangle, \sigma \parallel \alpha) \ \ = \langle\langle \tau, c, r\rangle, \sigma[r(Rm) \mapsto r(Rn)] \parallel \tau : r(Rn) \ \rhd r(Rm)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\text{If } r(Rn) \neq 0 \text{ then } nxt(\langle \tau, \mathtt{if} \ Rn \ c_1 \ c_2; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, c_1; c, r\rangle, \sigma \parallel \varepsilon\rangle \\
&\text{If } r(Rn) = 0 \text{ then } nxt(\langle \tau, \mathtt{if} \ Rn \ c_1 \ c_2; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, c_2; c, r\rangle, \sigma \parallel \varepsilon\rangle \\
&\text{If } r(Rn) \neq 0 \text{ then } nxt(\langle \tau, \mathtt{while} \ Rn \ c'; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, c'; \mathtt{while} \ Rn \ c'; c, r\rangle, \sigma \parallel \varepsilon\rangle \\
&\text{If } r(Rn) = 0 \text{ then } nxt(\langle \tau, \mathtt{while} \ Rn \ c'; c, r\rangle, \sigma \parallel \alpha) = \langle\langle \tau, c, r\rangle, \sigma \parallel \varepsilon\rangle
\end{aligned}
$$

Fig. 2. Thread nextstate function

Secondly, a symbolic version of the transition relation returning the prestate label and branching condition:

$$prv(l) = \{\langle l', cond(c', c)\rangle \mid l' : c', l : c\}$$

5 Logic Instantiation

We now instantiate the atomic propositions of $\mathcal{L}_0$ to reflect the structure of $\mathcal{P}_1$. The resulting language, $\mathcal{L}_1$, has thread variables A, B ranging over τ, and program variables x, y ranging over words. Abstract formula syntax then becomes:

$$
\begin{aligned}
e \in LExp &::= w \ \mid \ e \ \mathtt{op} \ e \mid {!e} \ \mid \ x \mid Rn \\
q \in AP &::= A \ \mathbf{at} \ l \mid A \ \mathbf{said} \ e_1 = e_2 \mid A : e_1 \ \lhd \ e_2 \mid A : e_1 \ \rhd \ e_2
\end{aligned}
$$

We refer to formulas of the shape $A : e_1 \ \lhd \ e_2$ or $A : e_1 \ \rhd \ e_2$ as read or write statements, respectively. Also, we call an expression of the shape $!e$ a reference, and references of the shape $!w$ are fully evaluated.

The primary intention of these constructs is to enable reasoning about thread behavior and its interactions. For example, the proposition A **at** l indicates that thread A is at a control state labeled l. The proposition A **said** $e_1 = e_2$ means that, when evaluated with respect to thread A's local register assignment, e_1 is equal to e_2. If neither e_1 nor e_2 mentions registers, and so is dependent only on the store, we can abbreviate A **said** $e_1 = e_2$ by just $e_1 = e_2$. Instantaneous (strong) reading and writing events are captured by propositions such as $A : e_1 \ \lhd \ e_2$ and $A : e_1 \ \rhd \ e_2$.

To enhance expressiveness, we introduce a few abbreviations. Weak versions of the reading and writing primitives are defined to express properties over past states. For instance, A **read** (A **wrote**) indicates that there exists the memory location x from which thread A read (resp. writes into x) some value at some point in the past. Other reading and writing primitives include:

– $\pi, i \models_\rho A$ **at** l, if $l : cmd(\pi(i), \rho(A))$,
– $\pi, i \models_\rho A$ **said** $e_1 = e_2$, if $w_1 = w_2$,
– $\pi, i \models_\rho A : e_1 \ \lhd \ e_2$, if $\alpha(\pi, i+1) = \rho(A) : w_1 \lhd w_2$,
– $\pi, i \models_\rho A : \ e_1 \rhd \ e_2$, if $\alpha(\pi, i+1) = \rho(A) : w_1 \rhd w_2$,

where, in the above clauses, $w_j = [\![e_j]\!](\sigma(\pi(i)), r(\pi(i), \rho(A)), \rho)$, $j \in \{1, 2\}$.

Fig. 3. Satisfaction conditions for $\mathcal{L}_1$

– A **read** e_1 **from** $e_2 \equiv$ **prev** A $(A : e_1 \ \lhd \ e_2)$
– A **wrote** e_1 **to** $e_2 \equiv$ **prev** A $(A : e_1 \ \rhd \ e_2)$
– A **read** $x \equiv \exists\, y.\ A$ **read** y **from** x
– A **wrote** $x \equiv \exists\, y.\ A$ **wrote** y **to** x
– Similarly, the recently-wrote connective: A **recently_wrote** e **to** x indicates that thread A performed an assignment to x in the past, and since then, has not written to x again, i.e., $\neg(A$ **wrote** $x)$ **since** A **wrote** e **to** x .

The extended semantics is shown in Figure 3 and uses the function $[\![e]\!](\sigma, r, \rho)$ to evaluate expressions in the obvious way.

We then move on to get to the interesting part, which concerns the epistemic modalities. The epistemic modalities allow us to reason about the knowledge acquired by threads based on their observations. For example, suppose A **read** x. In this case, if a thread ever wrote something to x not equal to $!x$ then this must have happened before $!x$ was written, by the same thread or a different one. And, this is known by A. In other words:

$$\begin{aligned} A\ \textbf{read}\ y\ \textbf{from}\ x \supset\ & \textbf{sometime}\ B\ \textbf{wrote}\ z\ \textbf{to}\ x\ \wedge\ z \neq y \\ & \supset A\ \textbf{knows}\ (\exists\, C.\ B\ \textbf{wrote}\ z\ \textbf{to}\ x \prec C\ \textbf{wrote}\ y\ \textbf{to}\ x \\ & \quad \prec\ A\ \textbf{read}\ y\ \textbf{from}\ x) \end{aligned}$$

Note that since we are working with a static and finite collection of threads, the existential quantifier is immediately eliminated in favour of a big disjunction.

5.1 Example: Peterson Algorithm

In this section, we illustrate the application of our logic and program model using the well-known Peterson's algorithm, a mutual exclusion algorithm designed for two threads. Peterson's algorithm ensures that two threads do not enter their critical sections simultaneously. This example helps demonstrate how our logic can be used to specify and verify concurrent programs.

As Figure 4 depicts, the Peterson's algorithm involves two threads, each trying to enter a critical section. This program can be easily translated into the program model of Section 4. The threads communicate through shared variables, specifically an array `flag` and a variable `victim`. Each thread follows a sequence of steps to set its intention to enter the critical section and then waits until it is safe to proceed. The logic for each thread involves setting a `flag`, updating the

```
Code for thread A:
work:      while True {
lock:      flag[0] = 1 ;
           victim = 0 ;
           while (flag[1] = 1 && victim = 0) {} ;
body:      noop ;
unlock:    flag[0] = 0 } ;

Code for thread B:
work:      while True {
lock:      flag[1] = 1 ;
           victim = 1 ;
           while (flag[0] = 1 && victim = 1) {} ;
body:      noop ;
unlock:    flag[1] = 0 } ;
```

Fig. 4. The Peterson algorithm.

`victim` variable, and then spinning in a loop until it is safe to enter the critical section.

To specify the desired properties of the algorithm, we focus on two main aspects: (1) the knowledge each thread has about its environment and (2) the specification of the global properties we wish to verify. For example, we want to ensure that if one thread has entered its critical section, the other thread cannot enter its critical section simultaneously.

We use the constructs of our logic to formalize these properties. For thread A, we define the conditions under which it knows certain facts about the shared variables. For instance, thread A knows the value of the `flag` and `victim` variables based on its own actions and the actions of thread B. We express this knowledge using epistemic modalities, allowing us to reason about what each thread can infer from its observations. For Peterson, there would seem to be two important pieces of knowledge A has:

1. $!\texttt{flag[0]} = x \supset A$ **knows** $(!\texttt{flag[0]} = x$ **since** $(A$ **wrote** y **to** $\texttt{flag[0]} \vee$ $($**init** $A \wedge x = 0))$
2. $!\texttt{victim} = 0 \supset A$ **knows** $(!\texttt{victim} = 0$ **since** $(A$ **wrote** 0 **to** $\texttt{victim} \vee$ $($**init** $A \wedge (!\texttt{victim} = 0))))$

The intuition should be clear. For instance, for 1, due to the MRSW[2] property of `flag[0]`, if `flag[0]` has the value x then x has remained the value of `flag[0]` since it was written to by A (or else x is 0 and A has not written to `flag[0]` yet).

[2] Here by MRSW, we mean multiple reader single writer.

We first try to prove mutual exclusion. For the global proof goal, define first:

$$\begin{aligned} A\ \mathit{enteredCS} \equiv\ & ((!\texttt{flag[1]} \neq 1) \vee (!\texttt{victim} \neq 0)) \wedge \\ & A\ \textbf{recently\_wrote}\ 1\ \textbf{to}\ \texttt{flag[0]} \wedge \\ & A\ \textbf{recently\_wrote}\ 0\ \textbf{to}\ \texttt{victim} \wedge \\ & A\ \textbf{wrote}\ 1\ \textbf{to}\ \texttt{flag[0]} \prec A\ \textbf{wrote}\ 0\ \textbf{to}\ \texttt{victim} \\ B\ \mathit{enteredCS} \equiv\ & ((!\texttt{flag[0]} \neq 1) \vee (!\texttt{victim} \neq 1)) \wedge \\ & B\ \textbf{recently\_wrote}\ 1\ \textbf{to}\ \texttt{flag[1]} \wedge \\ & B\ \textbf{recently\_wrote}\ 1\ \textbf{to}\ \texttt{victim} \wedge \\ & B\ \textbf{wrote}\ 1\ \textbf{to}\ \texttt{flag[1]} \prec B\ \textbf{wrote}\ 1\ \textbf{to}\ \texttt{victim} \end{aligned}$$

Observe that these two properties do not quite capture "being in the critical region" in the sense of the program counter having taken the spin loop exit branch and not yet having started to unlock. For instance, A *enteredCS* also holds in a state where A has completed its writes to `flag[0]` and `victim` and where the spin loop exit conditions hold, but where the while loop has not yet been entered, which is not normally regarded as part of the critical section. However, from such a state (where A has completed its writes, etc.), it is possible for A to enter the critical section without any involvement by the environment, which is why we have to eliminate such states in an account such as here where we are able to speak only about threads local view of their execution.

Now we can express mutual exclusion quite simply as follows:

$$A\ \mathit{enteredCS} \supset \neg(B\ \mathit{enteredCS})\ . \tag{1}$$

This statement is non-epistemic. In fact, we will prove, in Section 6.4, instead the epistemic property A *enteredCS* $\supset A$ **knows** $\neg(B$ *enteredCS*$)$ from which (1) follows by T.

6 Extended Inference System

We next extend the proof system introduced in section 3.2. The class of models is now specialized to those supported by the program model of section 4.

6.1 Equations

First, we have logical omniscience, i.e. any universally valid equation is known:

$$=\mathrm{I}\ \frac{}{\Gamma \vdash A\ \textbf{said}\ e_1 = e_2}$$

The side-condition here is that $e_1 = e_2$ is universally valid (valid for any assignment to variables, registers, memory locations). Similarly, we have that:

$$\neq\mathrm{I}\ \frac{}{\Gamma \vdash A\ \textbf{said}\ e_1 \neq e_2}$$

with a side-condition that $e_1 \neq e_2$ is universally valid.

Second, we have some highly circumscribed abilities to substitute equals for equals. Let us call a formula $\phi(x)$ an *A-context*, if x occurs only in the scope of an equality A **said** $e_1 = e_2$ or a read or write statement for thread A, and not in the scope of one of the modal operators **knows**, **prev**, or **since**. We obtain:

$$=\mathrm{E}\ \frac{\Gamma \vdash A\ \mathbf{said}\ e = e' \quad \Gamma \vdash \phi[e/x]}{\Gamma \vdash \phi[e'/x]}\ (\phi(x) \text{ is an } A\text{-context})$$

Using the K operator, we also easily derive the following rule:

$$=\mathrm{EK}\ \frac{\Gamma \vdash A\ \mathbf{knows}\ A\ \mathbf{said}\ e = e' \quad \Gamma \vdash A\ \mathbf{knows}\ \phi[e/x]}{\Gamma \vdash A\ \mathbf{knows}\ \phi[e'/x]}\ (\phi(x) \text{ is an } A\text{-context})$$

Note that more general versions of =E where x is allowed to appear in a modal context are unsound. For instance, we may obtain that $\vdash A$ **said** $4 = !3$ and $\vdash A$ **knows** $x = 4[4/x]$, but $\vdash A$ **knows** $!3 = 4$ is false (because some other thread might have written to location 3). Similar examples may be given for **prev** and **since**.

6.2 Label Statements

We then introduce inference rules related to label statements within our program model to reason about the control flow of threads by tracking their locations and transitions.

$$\textsc{Label}\mathrm{I1}\ \frac{\Gamma \vdash \mathbf{init}}{\Gamma \vdash A\ \mathbf{at}\ l_{A,0}} \quad \textsc{Label}\mathrm{I2}\ \frac{-}{\Gamma \vdash \bigvee\{A\ \mathbf{at}\ l \mid l \in \Lambda\}}$$

$$\mathrm{KAt}\ \frac{\Gamma \vdash A\ \mathbf{at}\ l}{\Gamma \vdash A\ \mathbf{knows}\ A\ \mathbf{at}\ l}$$

$$\textsc{PrAt}\mathrm{I1}\ \frac{\Gamma \vdash A\ \mathbf{at}\ l \qquad \Gamma \vdash \mathbf{prev}\ A\ \mathbf{active}}{\Gamma \vdash \bigvee\{\mathbf{prev}\ (A\ \mathbf{at}\ l' \ \wedge \phi) \mid \langle l', \phi\rangle \in \mathit{prv}(l)\}}$$

$$\textsc{PrAt}\mathrm{I2}\ \frac{\Gamma \vdash A\ \mathbf{at}\ l \qquad \Gamma \vdash \mathbf{prev}\ \neg(A\ \mathbf{active})}{\Gamma \vdash \mathbf{prev}\ A\ \mathbf{at}\ l}$$

$$\textsc{PrAAt}\mathrm{E}\ \frac{\Gamma \vdash \mathbf{prev}\ A\ (A\ \mathbf{at}\ l \wedge \mathit{cond}(c, c'))}{\Gamma \vdash A\ \mathbf{at}\ l'}\ (l : c, l' : c')$$

Soundness. Here we only show the soundness of PrAAtI1; others rules can be proved, similarly.

Assume $\pi, i \models_\rho A$ **at** l, i.e. $l : \mathit{cmd}(\pi(i), \rho(A))$. Let j be the previous thread state for A w.r.t. $\langle \pi, i\rangle$. There must be some $\langle l', \phi\rangle \in \mathit{prv}(l)$ such that $l' : \mathit{cmd}(\pi(j), \rho(A))$ and $\pi, j \models_\rho A$ **said** ϕ. But then by Proposition 2, the antecedent of PrAAtI1 holds. The proof for rule PrAAtE is equally simple.

6.3 Activity Statements

Additionally, we introduce a few rules to reason about the active state of threads.

$$\text{ActiveI2}\ \frac{\Gamma \vdash A : e_1 \ \lhd \ e_2}{\Gamma \vdash A\ \textbf{active}} \quad \text{ActiveI3}\ \frac{\Gamma \vdash A\ :\ e_1\ \rhd\ e_2}{\Gamma \vdash A\ \textbf{active}}$$

$$\text{ActiveE}\ \frac{\Gamma \vdash A\ \textbf{active} \qquad \Gamma \vdash B\ \textbf{active} \qquad \Gamma \vdash A \neq B}{\Gamma \vdash \phi}$$

6.4 Extend Proof System: Peterson Algorithm

We now turn to our running example and try to complete the proof using the inference rules we have derived in the previous sections.

Lemma 1. *For any thread A, the following holds:*

$$\vdash A\ \mathit{enteredCS}\ \supset\ A\ \textbf{knows}\ \neg(B\ \mathit{enteredCS})$$

Proof. We proceed through the following steps using the extended proof system:

1. **Assumption Decomposition**: Assume $A\ \mathit{enteredCS}$. By conjunction elimination ($\wedge$E):

   1. $(\neg!\texttt{flag[1]} \neq 1) \vee (\neg!\texttt{victim} \neq 0)$
   2. A **wrote** 1 **to** $\texttt{flag[0]}\ \prec\ A$ **wrote** 0 **to** $\texttt{victim}$
   3. $\neg(A$ **wrote** $\texttt{flag[0]})$ **since** A **wrote** 1 **to** $\texttt{flag[0]}$
   4. $\neg(A$ **wrote** $\texttt{victim})$ **since** A **wrote** 0 **to** $\texttt{victim}$

2. **Epistemic Foundation**: When A **at** l_6 (critical section entry):

   Proposition 3 (Local Knowledge).

   $$A\ \textbf{at}\ l_6\ \vdash\ A\ \textbf{knows}\ (!\mathit{flag[0]} = 1 \wedge !\mathit{victim} \neq 1)$$

   Proof. Apply =E rule to A's write operations, using:
   – From A **wrote** 1 **to** $\texttt{flag[0]}$, we know A **knows** A **said** $!\texttt{flag[0]} = 1$
   – From A **wrote** 0 **to** $\texttt{victim}$, we know A **knows** A **said** $!\texttt{victim} = 0$
   Since $!\texttt{flag[0]} = 1$ and $!\texttt{victim} = 0$ are universally valid in A's local context (no interference after writes), combining via $\wedge$I and =EK we can get the final result.

3. **Temporal Constraints**: From A **wrote** 1 **to** $\texttt{flag[0]}\ \prec\ A$ **wrote** 0 **to** $\texttt{victim}$:

   $$A\ \textbf{knows}\ (B\ \textbf{wrote}\ 1\ \textbf{to}\ \texttt{flag[1]}\ \prec\ B\ \textbf{wrote}\ 1\ \textbf{to}\ \texttt{victim})$$

   Because, by the definition of $\prec$, B's actions cannot interleave between A's writes (fair scheduling). Combined with Proposition 3, this precludes $B\ \mathit{enteredCS}$.

4. **Final Derivation**: Combine results from the previous steps and T axiom:

   $$A\ \textbf{knows}\ \neg(B\ \mathit{enteredCS})$$

   By $\supset$-introduction:

   $$\vdash A\ \mathit{enteredCS}\ \supset\ A\ \textbf{knows}\ \neg(B\ \mathit{enteredCS})$$

7 Normal Forms

We introduce a *normal form* representation for formulas in our logic. This normal form captures the essential properties of the original formula in a more structured representation and enables a more efficient verification. Technically, the goal is to show that for each formula $\phi \in \mathcal{L}_1$, it is possible to find a formula $NF(\phi)$ in a suitable normal form to be defined, such that $NF(\phi) \models \phi$. The reverse entailment $\phi \models NF(\phi)$ does not and is not intended to hold by design—the normal form is intentionally less restrictive to ensure sufficiency for verification, not equivalence. This asymmetry arises from the inherent non-reversibility of epistemic modalities under rules like KPA (knowledge propagation after activity) and KSRA (knowledge stability across runs), particularly when combined with past-time operators like "since". Specifically, temporal-epistemic properties cannot always be decomposed into component formulas due to the monotonic growth of knowledge over time in concurrent executions.

While constructing some normal form is straightforward—for instance, $\top$ (truth) trivially entails any formula—such simplistic forms lack practical utility. The criterion are that the candidate normal forms should satisfy their formal requirements (i.e., they entail the original formula) and that they are actually useful in that they can be used to efficiently verify practical code in an understandable and efficient manner, like any other instrument for static analysis.

For now we restrict attention to what we call *positive A formulas*, i.e., formulas ϕ for which:

1. All epistemic connectives occur within an even number of negations and
2. The only thread variable used is A.

Definition 1 (Positive A State Formula, Normal Form).

1. *A positive A state formula is a positive A formula built using only formulas of the form A* **said** $e_1 = e_2$*, A* **said** $e_1 \neq e_2$*, boolean connectives, and the epistemic modality.*
2. *A formula in normal form has the following shape*

$$\bigwedge\{\textbf{prev}\ A\ A\ \textbf{at}\ l' \wedge A\ \textbf{at}\ l \wedge \\ (A\ \textbf{active}\ \supset \bigvee_{j \in J_1}(\psi_{1,j,l',l} \wedge \textbf{prev}\ \psi_{2,j,l',l})) \wedge \\ (\neg A\ \textbf{active}\ \supset \bigvee_{j \in J_2}(\psi_{3,j,l',l} \wedge \textbf{prev}\ \psi_{4,j,l',l})) \mid l, l' \in \Lambda\} \tag{2}$$

where the formulas $\psi_{i,j,l',l}$ *satisfy the following requirements:*
- $\psi_{1,j,l',l}$, $\psi_{3,j,l',l}$ *are positive A state formulas.*
- $\psi_{2,j,l',l}$, $\psi_{4,j,l',l}$ *are positive A formulas.*

In the sequel we refer to the formulas $\psi_{i,j,l',l}$ as the *component formulas*, and to the index sets J_1, J_2 as the *disjunctive index sets*.

Theorem 2. *For each positive A formula ϕ exists a formula $NF(\phi)$ in normal form such that $NF(\phi) \models \phi$.*

Proof. First, we can rewrite each formula ϕ to a positive form with all subformulas of the forms A **said** $e_1 = e_2$, A **active**, $\phi_1 \wedge \phi_2$, **prev** ϕ, ϕ_1 **since** ϕ_2, A **knows** ϕ, A **at** l, $A : e_1 \lhd e_2$, $A : e_1 \rhd e_2$, or a negation thereof, with the exception of the epistemic modality that only occurs positively.

Second, we note that it is trivial to rewrite each ϕ to the following form:

$$\hat{\phi} == \bigwedge\{\textbf{prev } A\ A \textbf{ at } l' \wedge A \textbf{ at } l \supset \phi_{l',l} \wedge \textbf{prev } \top \mid l, l' \in \Lambda\} \ , \tag{3}$$

simply by choosing $\phi_{l',l} = \phi$ and noting that $\top \models \textbf{prev } \top$ and $\hat{\phi} \models \phi$.

The main proof is done by By induction on ϕ:

- Base case. For atomic propositions (e.g., A **at** l), the normal form directly encodes the satisfaction condition.
- Inductive case. For composite formulas (e.g., A **knows** ϕ), the normal form's structure preserves the semantics of the original formula.

We now proceed by considering each subformula type in turn:

Case $\phi \equiv A \textbf{ at } l$: Define $\phi_{l',l''} = \top$ if $l'' = l$, and $\bot$ otherwise. By the satisfaction condition for A **at** l, $\pi, i \models_\rho A \textbf{ at } l$ iff $l : \mathit{cmd}(\pi(i), \rho(A))$. The normal form ensures A **at** l holds at the current state, and **prev** A (A **at** l') captures the previous control state.
Case $\phi \equiv \neg(A \textbf{ at } l)$: Dual of the previous case.
Case $\phi \equiv A \textbf{ said } e_1 = e_2, \neg(A \textbf{ said } e_1 = e_2)$: Use (3), and a little boolean reasoning.
Case $\phi \equiv \phi_1 \wedge \phi_2$: By the induction hypothesis each ϕ_i can be rewritten to normal form with component formulas $\psi_{k,i,j,l',l}$, $k \in \{1, 2\}$, $i \in \{1, 2, 3, 4\}$, and j in the disjunctive index sets $J_{k,1}$, $J_{k,2}$. Then, $\mathit{NF}(\phi)$ can be constructed with disjunctive index sets $J_1 = J_{1,1} \times J_{2,1}$ and $J_2 = J_{1,2} \times J_{2,2}$ and with component formulas $\psi_{i,(j_1,j_2),l',l} == \psi_{i,j_1,l',l} \wedge \psi_{i,j_2,l',l}$ we obtain $\mathit{NF}(\phi) \models \phi$ as desired.
Case $\phi \equiv \neg(\phi_1 \wedge \phi_2)$: In this case we can choose $J_1 = J_1 \cup J_2$ (assuming the index sets are disjoint) and the $\psi_{i,(j_1,j_2),l',l}$ accordingly.
Case $\phi \equiv \textbf{prev } \phi'$: Trivial.
Case $\phi \equiv \neg(\textbf{prev } \phi')$: Use the equivalence $\neg(\textbf{prev } \phi') \equiv (\textbf{prev } \neg\phi') \vee \neg(\textbf{prev } \top)$.
Case $\phi \equiv \phi_1 \textbf{ since } \phi_2$: Use the equivalence

$$\phi_1 \textbf{ since } \phi_2 \equiv \phi_2 \vee (\phi_2 \wedge \textbf{prev } (\phi_1 \textbf{ since } \phi_2)) \ .$$

Case $\phi \equiv A \textbf{ knows } \phi$: Assume that ϕ has the form (2). First, use the equivalence

$$A \textbf{ knows } (\phi \wedge \psi) \equiv A \textbf{ knows } \phi \wedge A \textbf{ knows } \psi$$

and

$$A \textbf{ knows } A \textbf{ at } l \equiv A \textbf{ at } l$$

to rewrite ϕ to the form

$$\begin{gathered} \bigwedge\{\textbf{prev } A\ A \textbf{ at } l' \wedge A \textbf{ at } l \wedge \\ A \textbf{ knows } (A \textbf{ active } \supset \textstyle\bigvee_{j \in J_1} (\psi_{1,j,l',l} \wedge \textbf{prev } \psi_{2,j,l',l})) \wedge \\ A \textbf{ knows } (\neg A \textbf{ active } \supset \textstyle\bigvee_{j \in J_2} (\psi_{3,j,l',l} \wedge \textbf{prev } \psi_{4,j,l',l})) \mid l, l' \in \Lambda\} \end{gathered}$$

We now use the entailments

$$A\ \mathbf{active}\ \supset\ A\ \mathbf{knows}\ \phi \models A\ \mathbf{knows}\ (A\ \mathbf{active}\ \supset\ \phi)$$

(cf. the rules KP1 and KP2), and, even more crudely:

$$A\ \mathbf{knows}\ \phi_1 \vee A\ \mathbf{knows}\ \phi_2 \models A\ \mathbf{knows}\ (\phi_1 \vee \phi_2)$$

Now rewrite to:

$$\begin{gathered}\bigwedge\{\mathbf{prev}\ A\ A\ \mathbf{at}\ l' \wedge A\ \mathbf{at}\ l \wedge \\ (A\ \mathbf{active}\ \supset \bigvee_{j\in J_1}(A\ \mathbf{knows}\ \psi_{1,j,l',l} \wedge A\ \mathbf{knows\ prev}\ \psi_{2,j,l',l})) \wedge \\ (\neg A\ \mathbf{active}\ \supset \bigvee_{j\in J_2}(A\ \mathbf{knows}\ \psi_{3,j,l',l} \wedge A\ \mathbf{knows\ prev}\ \psi_{4,j,l',l})) \mid l, l' \in \Lambda\}\end{gathered}$$

and then finally to

$$\begin{gathered}\bigwedge\{\mathbf{prev}\ A\ A\ \mathbf{at}\ l' \wedge A\ \mathbf{at}\ l \wedge \\ (A\ \mathbf{active}\ \supset \bigvee_{j\in J_1}(A\ \mathbf{knows}\ \psi_{1,j,l',l} \wedge \mathbf{prev}\ A\ \mathbf{knows}\ \psi_{2,j,l',l})) \wedge \\ (\neg A\ \mathbf{active}\ \supset \bigvee_{j\in J_2}(A\ \mathbf{knows}\ \psi_{3,j,l',l} \wedge \mathbf{prev}\ A\ \mathbf{knows}\ \psi_{4,j,l',l})) \mid l, l' \in \Lambda\}\end{gathered}$$

which is valid by suitably pushing around the activity statements.
Case $A : e_1 \ \lhd\ e_2$: Use the equivalence

$$\mathbf{prev}\ A\ A\ \mathbf{at}\ l' \wedge A\ \mathbf{active} \wedge A\ \mathbf{at}\ l \wedge A\ \mathbf{said}\ Rn = e_1 \wedge A\ \mathbf{said}\ Rm = e_2 \equiv A : e_1 \lhd e_2 \quad (4)$$

provided that $prv(l) = \langle l', \top\rangle$ and $l' : Rn \lhd\ !Rm; c$.
Case $A : \neg(e_1 \ \lhd\ e_2)$: Use (4) again.
Case $A : e_1 \ \rhd\ e_2,\ A : \neg(e_1 \ \rhd\ e_2)$: Similar to the read case.

8 Extension to Rely-Guarantee Reasoning

The rely-guarantee (RG for short) reasoning [23] is a compositional technique used to verify large systems, including those with concurrent threads. It extends Hoare logic [20] by incorporating assumptions about the behavior of the environment (rely conditions) and guarantees about the behavior of the executing (sub-)program, a.k.a, thread, (guarantee conditions). Using this technique, conditions have the form $\mathcal{R}$ (rely), $\mathcal{G}$ (guarantee), α (precondition), β (postcondition), and, we use $\mathbf{knows}_\tau\ \alpha$ for the epistemic modality. The satisfaction condition is as expected:

Proposition 4. *We say $p \models_\pi \mathbf{knows}_\tau \alpha$ holds, if for all points $p' = \langle h', i\rangle$ for π and τ, if $p = \langle h, i\rangle$ and $h \restriction [0, i] = h' \restriction [0, i]$ then $p' \models \alpha$.*

Moreover, to ensure that the thread τ's knowledge of a given property ϕ persists under all environment actions allowed by $\mathcal{R}$:

Definition 2. *We say a formula ϕ is stable under $\mathcal{R}$ if*

$$\mathcal{R} \vdash\ \mathbf{knows}_\tau\ \phi \implies \mathbf{always}(\mathbf{knows}_\tau\ \phi)$$

In the rely-guarantee reasoning, an assertion is a judgment of the form $\mathcal{R}, \mathcal{G} \vdash \alpha\ \{\tau : c\}\ \beta$. The intention is that in the context of a given large and multithreaded program (giving the context that allows us to evaluate the epistemic modality) and under the assumption that the environment satisfies the rely condition $\mathcal{R}$, thread τ sometimes starts executing command associated with c in a state satisfying α, throughout the execution the condition $\mathcal{G}$ hold and the final state satisfy the property β. To put this satisfaction condition formally,

Proposition 5. *We say $\mathcal{R}, \mathcal{G} \vdash \alpha\{\tau : c\}\beta$ is satisfied, if for all runs π, and the given execution point i such that $ctl(\pi(i), \tau) = c$ and $hist(\pi, \tau) = \langle h, f\rangle$ and $\langle h, f(i)\rangle \models_{\pi,\tau} \alpha$:*

1. *If the environment satisfies $\mathcal{R}$ for all $j \geq f(i)$*
2. *Then, τ maintains $\mathcal{G}$ throughout the execution, i.e., $\langle h, j\rangle \models_{\pi,\tau} \mathcal{G}$ for all $j \geq f(i)$*
3. *If execution terminates, say in k, then $\langle h, k\rangle \models_{\pi,\tau} \beta$*
4. *And, $\forall \pi, i.\ \pi, i \models\ \alpha, \textbf{always knows}_\tau \mathcal{R} \implies \pi, i \models \beta, \textbf{always knows}_\tau\ \mathcal{G}$*

We can now define the inference rules, of which here we only present the parallel composition of threads, which is more interesting; defining other rules is straightforward.

$$\frac{\begin{array}{l}\textbf{always knows}_{\tau_1}\mathcal{R}_1, \textbf{always knows}_{\tau_1}\mathcal{G}_1 \vdash \alpha_1\ \{\tau_1 : c_1\}\ \beta_1 \quad \mathcal{G}_1 \to \mathcal{R}_2 \\ \textbf{always knows}_{\tau_2}\mathcal{R}_2, \textbf{always knows}_{\tau_2}\mathcal{G}_2 \vdash \alpha_2\ \{\tau_2 : c_2\}\ \beta_2 \quad \mathcal{G}_2 \to \mathcal{R}_1\end{array}}{\begin{array}{l}\textbf{always knows}_{\tau_1}(\mathcal{R}_1 \wedge \textbf{stable}(\mathcal{G}_2)), \textbf{always knows}_{\tau_2}(\mathcal{R}_2 \wedge \textbf{stable}(\mathcal{G}_1)) \\ \quad \vdash \alpha_1 \wedge \alpha_2 \{\{\tau_1 : c_1\} \parallel \{\tau_2 : c_2\}\}\ \beta_1 \wedge \beta_2\end{array}}$$

Where, $\textbf{stable}(\mathcal{G}_i)$ asserts that $\mathcal{G}_i$ is epistemically stable under $\mathcal{R}_j$. In shared-variable concurrency, postconditions often depend on the global state, such as mutual exclusion or shared-variable invariants. While epistemic modalities ($\textbf{knows}_\tau$) are useful for local reasoning within each thread, the parallel composition rule must ensure that the combined behavior of threads satisfies global properties. To this end, we define the postcondition of the parallel composition rule as $\beta_1 \wedge \beta_2$, where β_1 and β_2 are the postconditions of threads τ_1 and τ_2, respectively. This is justified by the rely-guarantee conditions ($\mathcal{G}_1 \to \mathcal{R}_2$ and $\mathcal{G}_2 \to \mathcal{R}_1$), which ensure that each thread's actions preserve the other's assertions, allowing us to lift local guarantees to global correctness. For example, in Peterson's algorithm, the mutual exclusion property is a global invariant that cannot be inferred solely from thread-local knowledge but follows from the compatibility of τ_1's and τ_2's rely-guarantee contracts.

8.1 Applying the RG Method to the Peterson Algorithm

We now try to apply the RG method to the Peterson algorithm and show how we can verify threads in isolation. As we discussed before the RG method involves specifying conditions under which each thread operates, known as rely and guarantee conditions. These conditions describe the assumptions a thread makes about its environment and the commitments it guarantees to the system. For the Peterson algorithm, these conditions are defined for two threads, A and B, which use shared variables `flag[0]`, `flag[1]`, and `victim` to manage access to the critical section.

To prove that the Peterson algorithm guarantees mutual exclusion, we define the following rely and guarantee conditions:

– Rely condition for A (R_1):

$$\textbf{always knows}_{\tau_1}(!\texttt{flag[0]} = 1 \wedge !\texttt{victim} = 1) \supset \neg(\tau_2\ \mathit{enteredCS})$$

– Guarantee condition for A (G_1):

$$\textbf{always knows}_{\tau_1}(!\texttt{flag[0]} = 0 \supset \neg(\tau_1 \mathit{enteredCS}))$$

Moreover, since $\mathcal{G}_1 \supset \mathcal{R}_2$, we know that B's actions (under $\mathcal{R}_2$) cannot write to `flag[0]`, therefore, A's knowledge of $\texttt{flag[0]} = 0$ remains stable. Similarly, for thread B these conditions can be defined. Then, each thread proves its guarantee using its rely condition. For example A proves that $\vdash \alpha_1\ \{\tau_1 : c_1\}\ \beta_1$ where the proof uses $\mathcal{R}_1$ to assume τ_2's behavior and $\mathcal{G}_1$ to ensure τ_1's actions $\beta_1 = \texttt{flag[0]} = 0 \supset \neg(\tau_1\ \mathit{enteredCS})$ preserve. Similarly, for the B, with $\beta_2 = \texttt{flag[1]} = 0 \supset \neg(\tau_2\ \mathit{enteredCS})$.

Lemma 2. *If τ_i knows ϕ , and ϕ is stable under interference (ensured by $\mathcal{G}_j \rightarrow \mathcal{R}_i$, then ϕ holds globally.*

Proof. – τ_i knows ϕ means ϕ holds in all states consistent with τ_is observations.
– $\mathcal{G}_j \rightarrow \mathcal{R}_i$ ensures τ_j's actions preserve ϕ.
– Since ϕ is stable and τ_i knows ϕ, ϕ holds in all reachable global states.

For thread A this means:

– A knows β_1 locally, i.e., $\textbf{knows}_{\tau_1}\ beta_1$
– $\mathcal{G}_2 \rightarrow \mathcal{R}_1$ ensure that B's actions do not invalidate β_1.
– Thus, β_1 holds globally.

Similar reasoning also holds for β_2. Now we can apply the parallel composition rule, and by Lemma 2 we know the β_1 and β_2 hold globally, which in turn imply that $\neg(\tau_1\ \mathit{enteredCS}\ \wedge\ \tau_2\ \mathit{enteredCS})$.

Finally, we prove mutual exclusion. Assume $\tau_1\ \mathit{enteredCS}$ holds. We show $\neg(\tau_2\ \mathit{enteredCS})$:

1. We first expand $\tau_1\ \mathit{enteredCS}$:

$$(!\texttt{flag[1]} \neq 1 \vee !\texttt{victim} \neq 0) \wedge (\tau_1 \textbf{recently\_wrote} 1 to \texttt{flag[0]}) \wedge ...$$

2. From $\mathcal{G}_1 \rightarrow \mathcal{R}_2$, we know that A's guarantee ensures that $!\texttt{flag[0]} = 1$ persists until it exits, so B's rely $rely_2$ (which assumes $!\texttt{flag[0]} = 1 \supset \neg\tau_2\ \mathit{enteredCS}$) is triggered.
3. From B's perspective, B observes $!\texttt{flag[0]} = 1$ and $!\texttt{victim} = 1$, so by $\mathcal{R}_2$, it cannot enter critical section.

9 Concluding Remarks

We introduced a novel approach for the compositional verification of concurrent programs using past-time epistemic temporal logic. Using the expressiveness of this logic, we have established a sound framework that addresses the challenges of modularity and composability in verifying concurrent systems. Our methodology extends the seminal work of Halpern and Moses by incorporating past-time operators. The use of past-time temporal epistemic logic enables precise reasoning about the historical context of thread operations, which is crucial for ensuring the correctness of concurrent programs. Moreover, our proof system has been shown to be sound, providing a solid foundation for further research and potential extensions.

References

1. Abadi, M., Plotkin, G.D.: A model of cooperative threads. Log. Methods Comput. Sci. **6**(4) (2010), `https://doi.org/10.2168/LMCS-6(4:2)2010`
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM **49**(5), 672–713 (2002), `https://doi.org/10.1145/585265.585270`
3. Balliu, M., Dam, M., Guernic, G.L.: Epistemic temporal logic for information flow security. In: Askarov, A., Guttman, J.D. (eds.) Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011, San Jose, CA, USA, 5 June, 2011. p. 6. ACM (2011), `https://doi.org/10.1145/2166956.2166962`
4. Baumann, C.: Ownership-based order reduction and simulation in shared-memory concurrent computer systems (2014)
5. Benton, N., Hofmann, M., Nigam, V.: Effect-dependent transformations for concurrent programs. In: Cheney, J., Vidal, G. (eds.) Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016. pp. 188–201. ACM (2016), `https://doi.org/10.1145/2967973.2968602`
6. Brookes, S.D.: Full abstraction for a shared-variable parallel language. Inf. Comput. **127**(2), 145–163 (1996), `https://doi.org/10.1006/inco.1996.0056`
7. Chadha, R., Delaune, S., Kremer, S.: Epistemic logic for the applied pi calculus. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5522, pp. 182–197. Springer (2009), `https://doi.org/10.1007/978-3-642-02138-1_12`
8. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer (1981), `https://doi.org/10.1007/BFb0025774`
9. Dechesne, F., Mousavi, M.R., Orzan, S.: Operational and epistemic approaches to protocol analysis: Bridging the gap. In: Dershowitz, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings.

Lecture Notes in Computer Science, vol. 4790, pp. 226–241. Springer (2007), https://doi.org/10.1007/978-3-540-75560-9_18
10. Dingel, J.: A trace-based refinement calculus for shared-variable parallel programs. In: Haeberer, A.M. (ed.) Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1548, pp. 231–247. Springer (1998), https://doi.org/10.1007/3-540-49253-4_18
11. Dingel, J.: A refinement calculus for shared-variable parallel and distributed programming. Formal Aspects Comput. **14**(2), 123–197 (2002), https://doi.org/10.1007/s001650200032
12. Dvir, Y., Kammar, O., Lahav, O.: An algebraic theory for shared-state concurrency. In: Sergey, I. (ed.) Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13658, pp. 3–24. Springer (2022), https://doi.org/10.1007/978-3-031-21037-2_1
13. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press (1995), https://doi.org/10.7551/mitpress/5803.001.0001
14. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. In: Kameda, T., Misra, J., Peters, J.G., Santoro, N. (eds.) Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984. pp. 50–61. ACM (1984), https://doi.org/10.1145/800222.806735
15. 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
16. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. J. ACM **37**(3), 549–587 (1990), https://doi.org/10.1145/79147.79161
17. Halpern, J.Y., Moses, Y., Tuttle, M.R.: A knowledge-based analysis of zero knowledge (preliminary report). In: Simon, J. (ed.) Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA. pp. 132–147. ACM (1988), https://doi.org/10.1145/62212.62224
18. Halpern, J.Y., Tuttle, M.R.: Knowledge, probability, and adversaries. In: Rudnicki, P. (ed.) Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989. pp. 103–118. ACM (1989), https://doi.org/10.1145/72981.72988
19. Hintikka, K.J.J.: Knowledge and belief: An introduction to the logic of the two notions (1962)
20. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969), https://doi.org/10.1145/363235.363259
21. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978), https://doi.org/10.1145/359576.359585
22. van der Hoek, W., van Hulst, M., Meyer, J.C.: Towards an epistemic approach to reasoning about concurrent programs. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) Sematics: Foundations and Applications, REX Workshop, Beekbergen, The Netherlands, June 1-4, 1992, Proceedings. Lecture Notes in Computer Science, vol. 666, pp. 261–287. Springer (1992), https://doi.org/10.1007/3-540-56596-5_37

23. Jones, C.B.: Developing methods for computer programs including a notion of interference. Ph.D. thesis, University of Oxford, UK (1981), https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064
24. Knight, S.: The Epistemic View of Concurrency Theory. Ph.D. thesis, École Polytechnique, Palaiseau, France (2013), https://tel.archives-ouvertes.fr/tel-00940413
25. Knight, S.: The epistemic view of concurrency theory. Ph.D. thesis, Ecole Polytechnique X (2013)
26. Kragl, B.: Verifying concurrent programs: Refinement, synchronization, sequentialization. Ph.D. thesis (2020)
27. Manna, Z., Pnueli, A.: Temporal verification of reactive systems - safety. Springer (1995)
28. Neiger, G., Toueg, S.: Substituting for real time and common knowledge in asynchronous distributed systems. In: Schneider, F.B. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987. pp. 281–293. ACM (1987), https://doi.org/10.1145/41840.41864
29. Neiger, G., Tuttle, M.R.: Common knowledge and consistent simultaneous coordination. In: van Leeuwen, J., Santoro, N. (eds.) Distributed Algorithms, 4th International Workshop, WDAG '90, Bari, Italy, September 24-26, 1990, Proceedings. Lecture Notes in Computer Science, vol. 486, pp. 334–352. Springer (1990), https://doi.org/10.1007/3-540-54099-7_23
30. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings. Lecture Notes in Computer Science, vol. 3170, pp. 49–67. Springer (2004), https://doi.org/10.1007/978-3-540-28644-8_4
31. Panangaden, P., Taylor, K.: Concurrent common knowledge: A new definition of agreement for asynchronous systems. In: Dolev, D. (ed.) Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988. pp. 197–209. ACM (1988), https://doi.org/10.1145/62546.62579
32. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977), https://doi.org/10.1109/SFCS.1977.32