ConCRIS: Imaginary Specifications for Fine-grained Concurrency

TAEYOUNG YOON, Seoul National University, Korea
SANGHYUN YI, Seoul National University, Korea
YONGHEE KIM, Seoul National University, Korea
JAEHYUNG LEE, Seoul National University, Korea
YUJIN IM, Seoul National University, Korea
TAEYOUNG RHEE, Seoul National University, Korea
SEONHO LEE, Seoul National University, Korea
YEJI HAN, Seoul National University, Korea
CHUNG-KIL HUR, Seoul National University, Korea

Conditional Refinement with Imaginary Specifications (CRIS) enables placing separation logic assertions at arbitrary program points, supporting refinement-based and separation-logic-based reasoning. While CRIS provides benefits for reasoning about programs with I/O behaviors, it only supports sequential programs, leaving its potential for concurrent verification unrealized.

We present ConCRIS, an extension of CRIS that combines incremental verification of effectful programs with support for verifying fine-grained concurrent data structures, as in separation logics like Iris. We develop LAIS, a specification language for ConCRIS, and address several technical challenges that arise from supporting LAIS: enabling helping mechanisms without step-indexing, developing language-generic prophecy variables, and modeling multi-node computing environments. We mechanize our theory in Rocq.

1 Introduction

Program verification has evolved through two main paradigms—refinement-based and separation logic-based approaches—each with distinct advantages for modular verification. Refinement-based approaches support *incremental verification*, where correctness is established through a chain of intermediate abstractions, and *open-setting verification*, where programs are verified to work correctly in any context, including those with unverified components. Separation logic-based approaches provide *ownership-based verification*, which tracks exclusive or shared access to resources and enables modular reasoning about disjoint program components through the frame rule. Building on both paradigms, Conditional Contextual Refinement (CCR) [26, 27] was recently proposed to support incremental, open-setting, and ownership-based verification.

CRIS (Contextual Refinement with Imaginary Specifications) [19] generalizes CCR by introducing *imaginary specifications*. While CCR provides reasoning principles to function clients solely through a pair of ownership assertions (*i.e.,* pre- and postconditions) about its behavior, imaginary specifications generalize them by allowing such ownership assertions to be freely mixed with and dependent on executable code. This addresses a key limitation of CCR-style specifications: their inability to express interactions with unverified code that may involve side effects such as I/O operations or even crashes. By enabling clients to reason about such interactions, CRIS provides more expressive specifications and powerful reasoning principles while preserving CCR's support for incremental, open-setting, ownership-based verification.

Authors' Contact Information: Taeyoung Yoon, Seoul National University, Korea, taeyoung.yoon@sf.snu.ac.kr; Sanghyun Yi, Seoul National University, Korea, sanghyun.yi@sf.snu.ac.kr; Yonghee Kim, Seoul National University, Korea, yonghee.kim@sf.snu.ac.kr; Jaehyung Lee, Seoul National University, Korea, jaehyung.lee@sf.snu.ac.kr; Yujin Im, Seoul National University, Korea, yujin.im@sf.snu.ac.kr; Taeyoung Rhee, Seoul National University, Korea, taeyoung.rhee@sf.snu.ac.kr; Seonho Lee, Seoul National University, Korea, seonho.lee@sf.snu.ac.kr; Yeji Han, Seoul National University, Korea, yeji.han@sf.snu.ac.kr; Chung-Kil Hur, Seoul National University, Korea, gil.hur@sf.snu.ac.kr.

Challenge: Extending CRIS to Concurrent Verification. CRIS and CCR only support sequential execution, while separation logics like *Iris* [18] have developed extensive techniques for reasoning about fine-grained concurrency. Notably, Iris supports higher-order ghost state [15], logically atomic triples, helping, and prophecy variables [17]—techniques particularly useful when verifying the linearizability [13] of complex concurrent data structures.

To extend CRIS to support concurrency, a natural approach would be to apply Iris's concurrency techniques to CRIS, following the path taken by ReLoC [9], an Iris-based framework for refinement-style verification. However, CRIS differs from ReLoC in fundamental ways that make direct adaptation problematic. Unlike ReLoC, CRIS supports (1) conditional incremental verification, enabling transitive reasoning about a module conditional on other modules' behaviors; (2) open-setting verification, allowing reasoning about interactions with arbitrary unverified code possibly involving crashes; and (3) preservation of behaviors including finite or infinite I/O traces.

These differences create incompatibilities with Iris's techniques. Specifically, CRIS cannot use step-indexing, which is essential to the support of concurrency in Iris. Step-indexing conflicts with conditional incremental verification, as observed in prior work [14, 19]. Moreover, side-effectful behaviors such as I/O operations or crashes cannot be easily encoded as ownership assertions. Indeed, no satisfactory support of verification for programs with I/O has been presented yet.

Contributions and Paper Structure. This paper presents **ConCRIS** (Concurrent CRIS), an extension of CRIS that supports fine-grained concurrent verification while preserving CRIS's unique capabilities for conditional incremental verification and open-setting verification as well as ownership-based reasoning.

After reviewing the background on CRIS (§2), we organize our contributions as:

- **Generalized logical atomicity** (§3). We develop a flexible generalization of *logically atomic triples*: a specialized Hoare triple in Iris that provides the illusion of physical atomicity to the clients of a fine-grained data structure. Our specification method, *logically-atomic imaginary specification(LAIS)*, aligns well with CRIS's support for unverified code and I/O.
- **Helping** (§4). We develop a novel helping mechanism as a user-level module that enables helping code involving I/O operations. Helping allows threads to complete operations on behalf of others, which is essential for verifying helping-based concurrent algorithms.
- **Prophecy variables** (§5). We develop a user-level module that enables prophecy-based reasoning in a language-generic way, allowing it to be linked with arbitrary modules modeling any programming language. Prophecy variables enable reasoning about future program behavior, which is crucial for verifying specific concurrent algorithms.
- **Hybrid Scheduling and Memory** (§6). ConCRIS enables users to freely define custom scheduling mechanisms and memory models as user-level modules and compose them hierarchically. Importantly, we port the memory model and the pre- and postconditions of operations from iRC11 [6]—a program logic supporting weak memory built on the Iris framework—to ConCRIS.[1] We present examples demonstrating our points.

Finally, we discuss related work and conclude in §7.

2 Background

We review CRIS's reasoning principles through a simple example. In particular, we introduce two key components of CRIS: *the inlining principle* and `Assume`, `Guarantee` operators. We assume no prior knowledge of CRIS.

[1] Our theories are formalized in the Rocq prover [28]. Some examples are currently incomplete, but will be finalized soon.

```
1 def LP_I(l) ≜ // implementation LP_I
2   v ← load(l);
3   print v;
4   return v;
```

```
1 def LP_A(l) ≜ // specification LP_A
2   v ← take(ℤ);
3   Assume(l ↦ v);
4   print v;
5   Guarantee(l ↦ v);
6   return v;
```

```
1 // memory specification Mem_A
2 def load(l) ≜
3   v ← take(ℤ);
4   Assume(l ↦ v);
5   Guarantee(l ↦ v);
6   return v;
7 def store(l, v) ≜ ···
8 def alloc(n) ≜ ···
9 def free(l) ≜ ···
10 def cas(l, v_o, v_n) ≜ ···
```

Fig. 1. A simple load-print example.

$$E_{core}(X) \triangleq \{\textbf{take}, \textbf{choose}\} \uplus \{\textbf{IO}_X\ fn\ arg \mid fn \in String,\ arg \in Any\} \qquad E_{state}(X) \triangleq \cdots$$

$$E_{ctrl}(X) \triangleq \{\texttt{Call}\ fn\ arg \mid fn \in String,\ arg \in Any\} \qquad E_{logic}(X) \triangleq \{\textbf{Assume}(P), \textbf{Guarantee}(P) \mid P \in iProp\}$$

$$E_{mod} \triangleq E_{core} \uplus E_{state} \uplus E_{ctrl} \uplus E_{logic} \qquad fun \triangleq \{(fn,\ body) \mid fn \in String,\ body \in Any \to itree\, E_{mod}\, Any\}$$

$$Mod \triangleq \left\{(fs,\ init) \,\middle|\, fs \in List\, fun,\ init \in String \xrightarrow{\text{fin}} Any\right\} \qquad \circ \in Mod \times Mod \to Mod$$

TAKE-SRC
$$\frac{\forall x \in X.\ t \lesssim K\,x}{t \lesssim (x \leftarrow \textbf{take}(X); K\,x)}$$

ASSUME-SRC
$$\frac{P \twoheadrightarrow t \lesssim s}{t \lesssim (\textbf{Assume}(P); s)}$$

CHOOSE-SRC
$$\frac{\exists x \in X.\ t \lesssim K\,x}{t \lesssim (x \leftarrow \textbf{choose}(X); K\,x)}$$

GUARANTEE-SRC
$$\frac{P * t \lesssim s}{t \lesssim (\textbf{Guarantee}(P); s)}$$

TAKE-TGT
$$\frac{\exists x \in X.\ t \lesssim K\,x}{(x \leftarrow \textbf{take}(X); K\,x) \lesssim s}$$

ASSUME-TGT
$$\frac{P * t \lesssim s}{(\textbf{Assume}(P); t) \lesssim s}$$

CHOOSE-TGT
$$\frac{\forall x \in X.\ t \lesssim K\,x}{(x \leftarrow \textbf{choose}(X); K\,x) \lesssim s}$$

GUARANTEE-TGT
$$\frac{P \twoheadrightarrow t \lesssim s}{(\textbf{Guarantee}(P); t) \lesssim s}$$

IO
$$\frac{\forall x \in X.\ K_t\,x \lesssim K_s\,x}{(x \leftarrow \textbf{IO}_X\ fn\ arg; K_t\,x) \lesssim (x \leftarrow \textbf{IO}_X\ fn\ arg; K_s\,x)}$$

INLINE-TGT
$$\frac{(f\ arg \gg\!= K) \lesssim s \qquad \Lambda_t(fn) = f}{(\texttt{Call}\ fn\ arg \gg\!= K) \lesssim s}$$

RETURN
$$(\texttt{return}\, v) \lesssim (\texttt{return}\, v)$$

Fig. 2. Selected and simplified definitions and simulation rules of CRIS.

2.1 CRIS primer

Fig. 1 has three modules: $\texttt{LP}_I$, $\texttt{LP}_A$, $\texttt{Mem}_A$.

In CRIS, every program is modeled as *interaction trees (ITrees)* [33]. ITrees are coinductively defined data structures for modeling programs that interact with the environment. We omit the theoretical details of ITrees; readers may view ITrees as a domain specific language with special operators. These include: **take**, **choose**, **Assume** and **Guarantee** operators used primarily for specifications, an **IO** operator to model I/O behaviors of a program, and `Call`, `return` operators as usual call and return. [2]

$\texttt{LP}_I$ *and* $\texttt{LP}_A$. $\texttt{LP}_I$ is a simple function that loads from the given memory location l, prints the value v and returns. A specification of $\texttt{LP}_I$ should inform users that: (1) $\texttt{LP}_I$ requires the *ownership* of the given location for `load` to be a safe operation, and (2) prints the value v (**print** v) read from

[2]We use **IO** and **print** interchangeably, as well as `Call` and the actual function names (*e.g.*, `load`).

location l. LP_A is precisely such a specification. It **Assume**s the ownership of $l \mapsto v$ (Line 3) and prints the value v taken from the resource (Line 4). In this sense, CRIS supports an effective specification of programs with I/O: we write a specification that freely mixes I/O *and* separation logic assertions.

Given a specification LP_A for LP_I, we have two aspects regarding LP_A, that is: (1) how do we *prove* that LP_I satisfies LP_A, and (2) how do we *use* this specification for verification of programs that use LP_I? With these two in mind, we illustrate the proof of the simulation relation *i.e.,* $\mathsf{LP}_I \lesssim \mathsf{LP}_A$, a standard technique for showing contextual refinement $\mathsf{LP}_I \sqsubseteq_{ctx} \mathsf{LP}_A$. This allows clients of LP_I to *link and use* LP_A, in a sense that will be clarified in the following paragraphs.

Proof of $\mathsf{LP}_I \lesssim \mathsf{LP}_A$. The bottom half of Fig. 2 shows CRIS's simulation rules. We refer to the implementation (*i.e.,* the left-hand side of the relation) as the *target* and the right-hand as the *source*.

In proving $\mathsf{LP}_I \lesssim \mathsf{LP}_A$, first note that the rule TAKE-SRC is applicable. LP_A takes an *imaginary value* from the context; applying TAKE-SRC gives us the argument $v \in \mathbb{Z}$. We must show the continuation of the simulation relation, for *all values* v that were taken from the context.

The case is similar for the **Assume** operator in Line 3, with rule ASSUME-SRC. **Assume** takes an *ownership* of the resource $l \mapsto v$. In establishing the continuing simulation relation $\mathsf{LP}_I \lesssim \mathbf{print}\, v;\ \cdots$, we are given $l \mapsto v$, yielding the obligation $l \mapsto v \mathrel{-\!\!*} \mathsf{LP}_I \lesssim \mathbf{print}\, v;\ \cdots$.

After executing **take** and **Assume** operators in the source, the remaining simulation is:

$$\forall v \in \mathbb{Z}.\, l \mapsto v \mathrel{-\!\!*} (v \leftarrow \mathtt{load}(l);\ \mathbf{print}\ v;\ \mathtt{Ret}\ v) \lesssim (\mathbf{Guarantee}(l \mapsto v);\ \mathbf{print}\ v;\ \mathtt{Ret}\ v)$$

With ownership of $l \mapsto v$, we execute the `load` operation at the target, showing that it is a valid memory access, via the *inlining rule*. The principle is straightforward: we substitute the actual function code `load` in Mem_A with the function call in LP_I (INLINE-TGT). In this way, we *use* the specification Mem_A against which memory heap implementations are verified. Note the parameter Λ_t in INLINE-TGT, a list of inlinable functions.

After substitution, we have **take**, **Assume**, **Guarantee**, respectively, at the target-side. The proof obligation for **take** in the target-side, *i.e.* TAKE-TGT is mathematically dual to TAKE-SRC. We *instantiate* a value for `v`, which represents *imaginary argument passing* to the `load` function (in this case, the value taken via **take** at line 2 in LP_A). We pass the ownership of the points-to predicate via ASSUME-TGT, and immediately retrieve it via GUARANTEE-TGT. The specification of `load` ensures that the returned value v is the one we passed from the start.

After executing **Guarantee** at the source by returning the ownership of $l \mapsto v$ to the context, we have a simple remaining goal: $(\mathbf{print}\, v;\ \mathtt{return}\, v) \lesssim (\mathbf{print}\, v;\ \mathtt{return}\, v)$, which is trivial since both sides are identical. We end the proof by IO and RETURN.

Summary. CRIS *operationalizes* separation logic assertions so that they can be placed in arbitrary points of a specification. This enables users to (1) perform standard separation logic ownership-based reasoning in a simulation proof through **Assume** and **Guarantee** operators, (2) *use* specifications via the *inlining principle*.

3 Logical Atomicity in Imaginary Specifications

We present a new specification method called *logically atomic imaginary specification (LAIS)* for ConCRIS. We use a priority queue as our motivating example (§3.1) and show how it can be specified in ConCRIS (§3.2).

3.1 Motivating example: a concurrent priority queue

Fig. 3 shows an implementation of a concurrent priority queue [12]. Note the $\mathcal{Y}$ operators: ConCRIS models concurrency as *cooperative multithreading*, where $\mathcal{Y}$ is an explicit yield call to the scheduler.

```
1  def PQ.New(range) ≜
2    𝒴; l ← alloc(range + 1);
3    𝒴; store(l, range);
4    for (i = 0; i < range; ++i);
5      𝒴; b ← Bin.New();
6      𝒴; store(l + i + 1, b);
7    𝒴; return l

8  def PQ.Add(l, n, v) ≜
9    𝒴; b ← load(l + n + 1);
10   𝒴; r ← Bin.Add(b, v);
11   𝒴; return r

12 def PQ.RemoveMin(l) ≜
13   𝒴; range ← load(l);
14   for (i = 0; i < range; ++i);
15     𝒴; b ← load(l + i + 1);
16     𝒴; r ← Bin.Remove(b);
17     if r = null then
18       𝒴; continue;
19     else
20       𝒴; return r
21   𝒴; return null
```

Fig. 3. Implementation I_{Queue} of a concurrent priority queue with fixed range of priorities.

$\mathcal{Y}$ enables flexible, fine-grained specification of concurrent data structures, as we demonstrate below. We defer the formal definition of $\mathcal{Y}$ until §6.

A priority queue maintains items where each item has an associated priority. I_{Queue} provides three methods: `PQ.New` for initialization of the queue, `PQ.Add` for adding an element with priority n, and `PQ.RemoveMin` for removing an element with the minimal priority.

`PQ.New` initializes the queue with priority bound *range*. It first records the bound at the first entry of the allocated list (Line 3), then iteratively allocates bins, a multiset of elements, that correspond to each priority (Lines 4–6). We then have a list l of pointers to each bin returned by the `PQ.New` method. We assume that a thread-safe (*i.e.,* linearizable) bin module with functions (*i.e.*, `Bin.New`, `Bin.Add` and `Bin.Remove`) and their specifications similar to Mem_A in Fig. 1 (*i.e.,* a pair of **Assume** and **Guarantee**) is provided.

`PQ.Add` is simple: it adds v to the bin at index n.

`PQ.RemoveMin` removes the element in the queue with the highest priority (a smaller index indicates higher priority), by iteratively checking each priority bin.

Linearizability and logical atomicity. First, note that I_{Queue} is *not linearizable.*

Linearizability [12] is a strong, canonical correctness property for concurrent data structures. Informally, a linearizable data structure ensures that although multiple operations (*e.g.*, push and pop methods of a stack) may concurrently overlap, each method call has a *linearization point* where it behaves *as if* they are atomically executed at their linearization points.

Linearizability is important since it is easy to reason with invariants. Invariants in separation logic can be viewed as a global storage that threads access: a thread can claim the resource stored in the invariant for an operation, but have to show right after that it can re-establish the invariant, so that other threads can rely on them. INV is a rule that reflects this intuition. For a physically atomic operation e, with the *knowledge* that R is stored as an invariant, *i.e.,* $\boxed{R}$, we are able to access R before e and *have to* establish R after e.

The strength of linearizability is that we are able to access the invariant between its linearization point *as if* it were a physically atomic operation, although it may be composed of multiple steps.

$$\textsc{logatom-inv}\ \frac{\langle R * P\rangle\ e\ \langle v.\ R * Q(v)\rangle}{\boxed{R} \vdash \langle P\rangle\ e\ \langle v.\ Q(v)\rangle} \qquad \textsc{inv}\ \frac{\{R * P\}\ e\ \{v.\ R * Q(v)\} \qquad \mathsf{phys\_atomic}(e)}{\boxed{R} \vdash \{P\}\ e\ \{v.\ Q(v)\}}$$

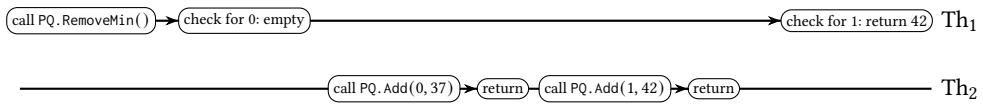

Fig. 4. A history of invocations and responses that shows why I_{Queue} is not linearizable.

```
1  def PQ.New(range) ≜
2    Assume(0 < range);
3    𝒴;
4    (γ, l) ← choose(gname × Val);
5    Guarantee(isPQ_γ(range, l) ∗ PQ_γ(∅));
6    return l;

7  def PQ.Add(l, n, val) ≜
8    (γ, range) ← take(gname × ℕ);
9    Assume(isPQ_γ(range, l) ∗ n < range);
10   𝒴;
11   s ← take([0..range] → list Val);
12   Assume(PQ_γ(s));
13   Guarantee(PQ_γ(s[n := val :: s[n]]));
14   𝒴; return null;

15 def PQ.RemoveMin(l, range) ≜
16   (γ, range) ← take(gname × ℕ);
17   Assume(isPQ_γ(range, l));
18   for (i = 0; i < range; ++i) {
19     𝒴;
20     s ← take([0..range] → list Val);
21     Assume(PQ_γ(s));
22     Guarantee(PQ_γ(s[i := tail(s[i])]));
23     𝒴;
24     match s[i] with
25     | [] => continue;
26     | v :: l => return v;
27     end
28   }
29   return null;
```

Fig. 5. A_{Queue}: the LAIS of I_{Queue}.

Given a logically atomic triple (LAT) specifying e, denoted $\langle P\rangle\ e\ \langle v.\ Q(v)\rangle$, observe that LOGATOM-INV does not require the code e to be physically atomic. Regardless of the number of physically atomic steps e takes, LOGATOM-INV allows the user to *access* the invariant, satisfy its precondition $R * P$, and re-establish the invariant from the postcondition $R * Q(v)$, *as if* one were applying INV.

Why is I_{Queue} not linearizable? The key reason I_{Queue} is not linearizable is that other threads can intervene during the iterative trials by the remover thread (*i.e.*, during Lines 14–20 in Fig. 3). Fig. 4 shows a possible non-linearizable history of I_{Queue}. Right after Th_1 observes that the bin at index 0 is empty, Th_2 adds a value 37 at index 0 and then a value 42 at index 1. Then Th_1 observes that the bin at index 1 has 42 and returns it. This behavior would be impossible in any linearizable history: at whichever point `PQ.RemoveMin` is executed, it cannot observe emptiness at index 0 and non-emptiness at index 1 because 37 was added to index 0 before 42 was added to index 1.

Thus, a naive specification of `PQ.RemoveMin` with LATs would fail to capture the functional essence of `PQ.RemoveMin`, *i.e.* that it tries to remove the minimal element:

REMOVEMIN-NAIVE

$$\langle \mathsf{PQ}(q)\rangle\ \texttt{PQ.RemoveMin}(l)\ \langle v.v = \texttt{null} * \mathsf{PQ}(q) \lor \exists n,\ v = \mathit{head}(q[n]) * \mathsf{PQ}(q[n := \mathit{tail}(q[n])])\rangle$$

Indeed, REMOVEMIN-NAIVE is a valid specification for any concurrent library which removes an element from the queue, regardless of the priority.

We end this section by noting that these implementation of a priority queue is known to be quiescently consistent [8], a relaxed correctness condition weaker than linearizability.

$$\textsc{inv-access}\ \frac{\boxed{P}^{\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad (P * (P \mathrel{-\!\!*} {}_{\mathcal{E}\setminus\mathcal{N}}\Rightarrow_{\mathcal{E}} \mathsf{True})) \mathrel{-\!\!*} t \lesssim_{\mathcal{E}\setminus\mathcal{N}} s}{t \lesssim_{\mathcal{E}} s} \qquad \textsc{yield-tgt}\ \frac{t \lesssim_{\top} (\mathcal{Y}; s)}{(\mathcal{Y}; t) \lesssim_{\top} (\mathcal{Y}; s)}$$

$$\textsc{yield-src}\ \frac{t \lesssim_{\mathcal{E}} s}{t \lesssim_{\mathcal{E}} (\mathcal{Y}; s)} \qquad \textsc{fupd-mon}\ \frac{P \vdash Q}{{}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} P \vdash {}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} Q} \qquad \textsc{sim-fupd}\ \frac{{}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2} t \lesssim_{\mathcal{E}_2} s}{t \lesssim_{\mathcal{E}_1} s} \qquad \textsc{inv-alloc}\ \frac{\mathcal{N} \subseteq \mathcal{E}}{P \vdash \Rightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N}}}$$

Fig. 6. Rules of ConCRIS related to $\mathcal{Y}$ and invariants.

3.2 LAIS to the rescue

We present a new specification called LAIS of I_{Queue} in Fig. 5. Each LAIS in A_{Queue} is a specification of its counterpart in I_{Queue}. $\mathsf{isPQ}_\gamma : \mathbb{N} \times \mathit{Val} \to \mathit{iProp}$ is a persistent *i.e.,* duplicable predicate, that typically arise in the specification of Iris. $\mathsf{isPQ}_\gamma(\mathit{range}, l)$ states that l is a valid pointer to a queue of size *range* and γ is the associated *ghost location* [16]. It comes with an associated exclusive predicate $\mathsf{PQ}_\gamma : ([0..\mathit{range}]^3 \to \mathit{list}\ \mathit{Val}) \to \mathit{iProp}$. Specifically, $\mathsf{PQ}_\gamma(s)$ represents the content s of the queue and its ownership, where the ghost location γ relates the two predicates.

***Explanation of* `PQ.New`.** We first note that `PQ.New` of A_{Queue} may be read as a direct translation of Hoare triples in separation logic:

$$\forall\, \mathit{range}, \{0 < \mathit{range}\}\ \mathsf{PQ.New}(\mathit{range})\ \{l.\ \mathsf{isPQ}(\mathit{range}, l) * \mathsf{PQ}(\emptyset)\}$$

The user of this triple must guarantee the precondition *i.e.,* $0 < \mathit{range}$, and can assume ownership of the two predicates, isPQ and PQ, required for further operations on the queue.

This rely/guarantee reasoning applies in exactly the same way in ConCRIS: the user will inline `PQ.New` in their implementation code, which appears in the target side of the simulation, leading to the application of rule ASSUME-TGT. After dealing with the $\mathcal{Y}$ in the target (the rule for this will be introduced shortly), we run into **`choose`** and **`Guarantee`** in the target. It is time to reap the benefits: we achieve the related predicates via rules CHOOSE-TGT and GUARANTEE-TGT.

***Explanation of* `PQ.Add`.** If `PQ.New` of A_{Queue} corresponds to Hoare triples in Iris, `PQ.Add` is a specification that corresponds to an LAT in Iris:

$$\forall\, l\ n\ v\ \mathit{range}.\ \mathsf{isPQ}(\mathit{range}, l) \vdash \langle s.\ \mathsf{PQ}(s)\rangle\ \mathsf{Add}(l, n, \mathit{range})\ \langle \mathsf{PQ}(s[n := v :: s[n]])\rangle$$

Before proceeding to explain why `PQ.Add` in A_{Queue} corresponds to an LAT in Iris, we first give rules of ConCRIS related to $\mathcal{Y}$ and invariants in Fig. 6.

A brief detour: invariants. First observe that the simulation relation ($\lesssim$) is annotated with masks ($\mathcal{E}$). Masks avoid *reentrancy* of invariants: it would be unsound to claim a resource stored in the invariant multiple times, as it is the whole point of separation logic to exploit the *exclusivity* of resource ownership. Thus the simulation is annotated with masks indicating which invariants can be opened during the proof of it. That is, we are only allowed to open $\boxed{P}^{\mathcal{N}}$ and access P in proving $t \lesssim_{\mathcal{E}} s$, only if $\mathcal{N} \subseteq \mathcal{E}$ (INV-ACCESS) [4].

Of course, we should *close the invariants* for other threads before yielding to the scheduler. This obligation is encoded in the rule YIELD-TGT: to execute the $\mathcal{Y}$ at the target, we have to close all

[3] $[0..\mathit{range}]$ is a finite set of natural numbers from 0 to $\mathit{range} - 1$.

[4] To support higher-order invariants without step-index, we adopt the approach of Nola, *i.e.,* stratified propositions and thus avoid the later modality. (P for stratified proposition)

the invariants we opened during our operation, *i.e.,* establish $\top$ mask of the simulation relation. A passionate reader may check that the combination of FUPD-MON and SIM-FUPD, together with the proposition $P \twoheadrightarrow {}_{\mathcal{E}\setminus\mathcal{N}}\Rightarrow_{\mathcal{E}} \mathsf{True}$ we are given in INV-ACCESS, enables the ability to close the masks. [5]

***Back to* `PQ.Add`.** Let us now illustrate how to *use* `PQ.Add` as a specification. As in §2, a user of `PQ.Add` would inline the code into their own code and observe that `PQ.Add` demands the predicate isPQ(*range*, *l*) (Lines 8–9). Proving this should be easy since isPQ(*range*, *l*) is a persistent predicate provided by the `PQ.New` method. After closing all invariants and removing the $\mathcal{Y}$ at the target side, we see that what essentially remains is a pair of **`Assume`** and **`Guarantee`** (Lines 11–13) without any $\mathcal{Y}$ between them. This is why our spec `PQ.Add` corresponds to an LAT in Iris: the user can access the predicate PQ from opening an invariant, have it updated via the pair of **`Assume`** and **`Guarantee`** and close the invariant with the updated predicate after the linearization point. This effectively provides the user the illusion that `PQ.Add` is an atomic operation.

***Explanation of* `PQ.RemoveMin`.** After going through how the user of `PQ.Add` deals with the *atomic ghost update* provided by LAIS of `PQ.Add` (*i.e.,* Lines 11–13), understanding what reasoning principles `PQ.RemoveMin` provides to its user is rather straightforward. The same pattern arises in Lines 20–22, but repeated *range* times in `PQ.RemoveMin`! The user would inline `PQ.RemoveMin` and proceed by induction on *range*: this would require the user to provide PQ for every bin from 0 to *range* − 1, which captures the right reasoning principle for the user.

Compared to REMOVEMIN-NAIVE, a naive specification of `PQ.RemoveMin` in the LAT style, LAIS properly specifies `PQ.RemoveMin` in a natural way. In each trial of `Bin.Remove`, the algorithm moves to the next bin *only if* it checks that the current bin is empty: such decisions are well reflected in Lines 24–27 in the specification.

4 Helping

In §3, we demonstrated how LAIS enables a natural specification for fine-grained concurrent data structures (FCDs). However, having the specification as a program in the realm of concurrency raises a challenge that must be addressed: namely *helping*. In this section, we explain why supporting helping is challenging in ConCRIS (§4.1), and provide a thread-local reasoning principle (§4.2).

4.1 What is helping, and why is helping a challenge in ConCRIS?

In §3, we saw how the verification of FCDs is conducted in ConCRIS. It includes proving correctness properties such as linearizability, since the specification provides the illusion of atomicity to its users. Usually, such proofs are essentially reduced to identifying the *linearization point*, and such points can be determined through thread-local reasoning.

However, there are classes of FCDs whose linearization points cannot be determined locally, namely FCDs with *external linearization points*. One well-known example is the elimination-backoff stack (ES) [11]. Fig. 7 presents an implementation of ES, where details are omitted for simplicity. Note that ES is the `Bin` module used by the $I_{\texttt{Queue}}$ example in §3.1 and that is why we call *push* and *pop* on stack as `Bin.Add` and `Bin.Remove`. Also, $I_{\texttt{Queue}}$ being verified in isolation with our stack implementation demonstrates the modularity of ConCRIS.

Explanation of the elimination stack. The distinguishing feature of ES is that it employs a *side channel* to avoid contention, and this is the main reason its linearization points are external. Let us examine `Bin.Add` to see what it does. First, it simply tries to push the value at the head of the internal linked list via a `cas` operation (Lines 5–8). The success of `cas` would imply that no

[5] We omit the definition and detailed explanations of ${}_{\mathcal{E}_1}\Rightarrow_{\mathcal{E}_2}$ modality in the rules, known as *fancy updates* in Iris.

```
 1 module Bin_I.
 2 def Bin.New() ≜ alloc(2)
 3 def Bin.Add(l, v) ≜
 4   while (true) {
 5     h_old ← load(l);
 6     h_new ← alloc(2);
 7     store(h_new, v); store(h_new + 1, h_old);
 8     r ← cas(l, h_old, h_new); // try push
 9     if (r) break; // success
10     ofr ← alloc(2); // try offer
11     store(ofr, v); store(ofr + 1, 0);
12     store(l + 1, ofr); store(l + 1, null);
13     r ← cas(ofr + 1, 0, 2); // try revoke
14     if (!r) break; // success
15   }
16   return null;
17 def Bin.Remove(l) ≜
18   while (true) {
19     h_old ← load(l);
20     if (h_old = null) continue;
21     h_next ← load(h_old + 1); // next head
22     r ← cas(l, h_old, h_next); // try pop
23     if (r) { // success pop
24       v ← load(h_old); return v;
25     }
26     ofr ← load(l + 1);
27     if (ofr = null) continue;
28     r ← cas(ofr + 1, 0, 1); // try pop
29     if (r) { // success pop
30       v ← load(ofr); return v;
31     } else { continue; }
32   }
```

Fig. 7. Implementation $\texttt{Bin}_\texttt{I}$ of an elimination-backoff stack (simplified and $\mathcal{Y}$s omitted).

other thread has tried to push or pop an element to the stack and we can safely return, but it is possible that there is contention and we must retry pushing, since we failed to commit.

In case of such failure, rather than simply repeating the entire process to push onto the stack, `Bin.Add` uses the side channel to push the element (Lines 10–13). Observe how the side channel is being exploited: the first store of Line 12 places an offer on the side channel (`store(`$l+1$`,` ofr`)`) and the second store revokes the offer from the side channel (`store(`$l+1$`, null)`). While this may seem pointless, note that any other thread can be scheduled between two successive `store`s. This in turn means that a thread attempting to `Bin.Remove` from the stack can *kick in* and take the offer from the side channel, accomplishing both `Bin.Add` and `Bin.Remove` operations at the same time.

It is exactly this case that makes the linearization point *non-local*. The instant when the thread invoking `Bin.Add` realizes that its offer has been taken is when the `cas` operation of Line 13 fails, while the *actual linearization point* is the moment when the thread invoking `Bin.Remove` successfully took the offer. Specifically, after trying to pop from the linked list and detecting contention (Lines 19–25), the thread executing `Bin.Remove` attempts to `cas` the state of the offer from 0 to 1 (Lines 26–28). The success of this `cas` leads to the offer being taken, and thus, the linearization point of the `Bin.Add`ing thread should be identified with this moment, right before the linearization point of `Bin.Remove`.

Call for helping. Now, suppose we are to verify that $\texttt{Bin}_\texttt{I}$ refines $\texttt{Bin}_\texttt{A}$ given in Fig. 8 using the thread-local simulation rules given in Fig. 2 and Fig. 6—we find that the given rules are insufficient! As explained in §3, such proofs involve identifying the linearization point and *updating* the ghost state by executing source-side operations at the *instant* when the push operation occurs.

However, this is not possible, since there is no way for the helping thread (*i.e.*, `Bin.Remove`) to execute the source operations on behalf of the `Bin.Add`ing thread. It would be too late for the `Bin.Add`ing thread to execute its source ghost updates when it takes control. To summarize, to support verification of FCDs with external linearization points in ConCRIS, we need a mechanism to *help* other threads with their jobs (*i.e.*, *to execute* the source specifications of other threads).

Although existing binary logic frameworks such as ReLoC have working support for helping, the nature of ConCRIS makes it difficult to directly adapt these solutions. In short, solutions such as the

```
1  module Bin_A.
2  def Bin.New() ≜
3    𝒴;
4    γ ← choose(gname); l ← choose(Val);
5    Guarantee(isBin_γ(l) ∗ Bin_γ(∅));
6    return l
7  def Bin.Add(l, v) ≜
8    γ ← take(gname);
9    Assume(isBin_γ(l));
10   𝒴;
11   s ← take(list Val);
12   Assume(Bin_γ(s));
13   Guarantee(Bin_γ(v :: s));
14   𝒴; return null;
15 def Bin.Remove(l) ≜
16   γ ← take(gname);
17   Assume(isBin_γ(l));
18   𝒴;
19   s ← take(list Val);
20   Assume(Bin_γ(s));
21   Guarantee(Bin_γ(tail(s)));
22   𝒴;
23   match s with
24   | v :: l => return v;
25   | [] => return null;
26   end
```

Fig. 8. Specification $\texttt{Bin}_\texttt{A}$ of $\texttt{Bin}_\texttt{I}$ (simplified).

```
1  param ID_job, R : Type.
2  param job : ID_job → itree E_help R.
3  module Help_on.
4  var req : list (ID_job × option R)
5  def Help.TryRun(hid) ≜
6    match lookup(req, hid) with
7    | Some (jid, Some ret) =>
8        r ← ret;
9    | Some (jid, None) =>
10       r ← job(jid);
11       req := update(req, hid, (jid, Some r));
12   | None => choose(∅)
13   end;
14   return r;
15 def Help.Help() ≜
16   hid ← choose(ℕ);
17   Help.TryRun(hid);
18 def Help.Run(jid) ≜
19   hid ← length(req);
20   req := req ++ [(jid, None)];
21   𝒴; r ← Help.TryRun(hid); 𝒴;
22   return r;
```

```
1  module Help_off.
2  def Help.Run(jid) ≜
3    𝒴; r ← job(jid);
4    𝒴; return r
5  def Help.Help() ≜ 𝒴
```

Fig. 9. The helping modules $\texttt{Help}_\texttt{on}$ and $\texttt{Help}_\texttt{off}$.

traditional *specification-as-resource* solution [30, 31] share the specification to be helped *through the invariants*. However, sharing our LAIS, which may include arbitrary separation logic assertions including invariants and quantification on invariants, again in the invariant, introduces essential circularity that cannot be addressed even by stratified techniques we employ such as Nola [21, 25].

4.2 How did we solve it?

Instead of having a step-indexed logic, we develop an operational solution: the *helping* module. Fig. 9 presents two modules, $\texttt{Help}_\texttt{on}$ and $\texttt{Help}_\texttt{off}$. As we can infer from their names, $\texttt{Help}_\texttt{on}$ is a module with its helping ability turned *on*, while $\texttt{Help}_\texttt{off}$ has it turned *off*. The key theorem establishes refinement between the two modules in the presence of a nondeterministic scheduler:

```
job(γ, v) ≜
  s ← take(list Val);
  Assume(Bin_γ(s));
  Guarantee(Bin_γ(v :: s))
```

```
1  module Bin_M.
2  def Bin.New() ≜
3    ···
4  def Bin.Add(l, v) ≜
5    γ ← take(gname);
6    Assume(isBin_γ(l));
7    𝒴;
8    Help.Run((γ, v));
9    𝒴; return null
```

```
10 def Bin.Remove(l) ≜
11   γ ← take(gname);
12   Assume(isBin_γ(l));
13   𝒴;
14   Help.Help();
15   s ← take(list Val);
16   Assume(Bin_γ(s));
17   Guarantee(Bin_γ(tail(s)));
18   𝒴; ···
```

Fig. 10. $\texttt{Bin}_\texttt{M}$, the intermediate LAIS of $\texttt{Bin}_\texttt{I}$ (simplified and omitted).

Theorem 4.1. $\texttt{Help}_{\text{on}} \circ \texttt{NDSch}_\texttt{I} \sqsubseteq_{ctx} \texttt{Help}_{\text{off}} \circ \texttt{NDSch}_\texttt{I}$
where $\texttt{NDSch}_\texttt{I}$ *is a scheduler module with nondeterministic scheduling policy.* [6]

Given Theorem 4.1, our key motto is: $\texttt{Help}_{\text{on}}$ *for verification*, $\texttt{Help}_{\text{off}}$ *for specification.*

Verification side: explanation of $\texttt{Help}_{\text{on}}$. Let us first focus on the verification side, *i.e.,* $\texttt{Help}_{\text{on}}$. `Help.Help` and `Help.Run` are the two main methods of $\texttt{Help}_{\text{on}}$, which provide the helping functionality to users. `Help.Help` **choose**s a help identifier *hid* for a job it will help with and attempts to execute it by calling `Help.TryRun`(*hid*). On the other hand, `Help.Run`(*jid*) first registers its job, identified by *jid*, by appending it to the end of the request list *req* (at index *hid*), yields to the scheduler to give other threads a chance to help, and when it regains control, it also attempts to run the same job registered at index *hid*, which may have already been helped by another thread or will be performed by this thread.

Then what does `Help.TryRun`(*hid*) do? It looks up the request list *req* at index *hid*. If there is a job *jid* waiting to be helped (Line 9), it executes the requested job in the form of an ITree, *job*(*jid*), (Line 10) and updates the request list at *hid* with the return value (Line 11). If the job has already been completed (Line 7), it simply returns the stored return value (Line 8). We note that the last case (Line 12) is a dummy case that we can ignore during verification.

Our key idea is as follows: by *linking our LAIS with* $\texttt{Help}_{\text{on}}$ and *inserting method calls in LAIS to* $\texttt{Help}_{\text{on}}$, we can *actually execute* the ghost update of other threads, registered in *req*. More specifically, instead of directly establishing $\texttt{Bin}_\texttt{I} \sqsubseteq_{ctx} \texttt{Bin}_\texttt{A}$, we prove an *intermediate refinement* between $\texttt{Bin}_\texttt{I}$ and $\texttt{Bin}_\texttt{M}$ (Fig. 10) linked with $\texttt{Help}_{\text{on}}$—an intermediate LAIS (*i.e.,* $\texttt{Bin}_\texttt{I} \sqsubseteq_{ctx} \texttt{Bin}_\texttt{M} \circ \texttt{Help}_{\text{on}}$).

Specification side: explanation of $\texttt{Help}_{\text{off}}$. However, this is not the end of the story. $\texttt{Bin}_\texttt{M}$ is not a feasible specification for a stack—for it to be usable, we must be able to show that our implementation, $\texttt{Bin}_\texttt{I}$, actually contextually refines $\texttt{Bin}_\texttt{A}$ (Fig. 8)! What we have shown is that $\texttt{Bin}_\texttt{I}$ only refines $\texttt{Bin}_\texttt{M} \circ \texttt{Help}_{\text{on}}$, a strange module that non-deterministically performs the operations.

This is what exactly Theorem 4.1 provides: an ability to *turn off helping* via contextual refinement, *i.e.,* $\texttt{Help}_{\text{on}} \sqsubseteq_{ctx} \texttt{Help}_{\text{off}}$. Having turned off the ability, what $\texttt{Help}_{\text{off}}$ does is trivial: `Help.Run`(*jid*) just runs the job, *i.e., job*(*jid*), while `Help.Help`() does nothing but $\mathcal{Y}$.

Note that there is essentially no difference between $\texttt{Bin}_\texttt{M}$ and $\texttt{Bin}_\texttt{A}$. The code of `Bin.Add` in $\texttt{Bin}_\texttt{M}$ is identical to that in $\texttt{Bin}_\texttt{A}$ except that the code at the ghost update (Line 17) in $\texttt{Bin}_\texttt{M}$ is replaced with `Help.Run`(γ, v). The code of `Bin.Remove` in $\texttt{Bin}_\texttt{M}$ is identical to that in $\texttt{Bin}_\texttt{A}$ except that a call to `Help.Help` is added at the linearization point (Line 23) in $\texttt{Bin}_\texttt{M}$. In other words, if we replace

[6] Detailed explanation of $\texttt{NDSch}_\texttt{I}$ follows in §6.

$$\textsc{help-run}\quad \frac{\forall \mathit{hid},\ \mathsf{Pend}(\mathit{hid}) \twoheadrightarrow t \lesssim (\mathcal{Y};\ r \leftarrow \mathtt{Help.TryRun}(\mathit{hid});\ \mathcal{Y};\ s(r))}{t \lesssim (r \leftarrow \mathtt{Help.Run}(\mathit{jid});\ s(r))}$$

$$\textsc{tryrun-pend}\quad \frac{\mathsf{Pend}(\mathit{hid}) \qquad \mathbf{return}\ () \lesssim \mathit{job}(\mathit{jid}) \qquad \mathsf{Done}(\mathit{hid},\ \mathit{ret}) \twoheadrightarrow t \lesssim s(\mathit{ret})}{t \lesssim (\mathtt{Help.TryRun}(\mathit{hid});\ s(r))}$$

$$\textsc{tryrun-done}\quad \frac{\mathsf{Done}(\mathit{hid},\ \mathit{ret}) \qquad t \lesssim s(\mathit{ret})}{t \lesssim (r \leftarrow \mathtt{Help.TryRun}(\mathit{hid});\ s(r))}$$

$$\textsc{pend-excl}\quad \mathsf{Pend}(\mathit{hid}) * \mathsf{Pend}(\mathit{hid}) \vdash \mathsf{False}$$

Fig. 11. Reasoning rules for $\mathtt{Help}_{\mathrm{on}}$.

$\mathtt{Help}_{\mathrm{on}}$ with $\mathtt{Help}_{\mathrm{off}}$ in $\mathtt{Bin}_{\mathrm{M}}$ using Theorem 4.1, we obtain a module that becomes identical to $\mathtt{Bin}_{\mathrm{A}}$ after inlining the functions in $\mathtt{Help}_{\mathrm{off}}$.

Summary. The refinement chain of the overall proof is given as:

$$\mathtt{Bin}_{\mathrm{I}} \sqsubseteq_{ctx} \mathtt{Bin}_{\mathrm{M}} \circ \mathtt{Help}_{\mathrm{on}} \sqsubseteq_{ctx} \mathtt{Bin}_{\mathrm{M}} \circ \mathtt{Help}_{\mathrm{off}} \sqsubseteq_{ctx} \mathtt{Bin}_{\mathrm{A}}$$

where the second refinement is provided by Theorem 4.1 and the third trivial. [7]

4.3 The verification of the elimination stack

We consider the proof of $\mathtt{Bin.Add}_{\mathrm{I}} \lesssim \mathtt{Bin.Add}_{\mathrm{M}}$, which is a part of showing $\mathtt{Bin}_{\mathrm{I}} \sqsubseteq_{ctx} \mathtt{Bin}_{\mathrm{M}} \circ \mathtt{Help}_{\mathrm{on}}$ (the first part of the refinement chain), where we use $\mathtt{Help}_{\mathrm{on}}$ for helping. As we saw in §3, verification in ConCRIS largely resembles that of Iris: in this way, we achieve portability of complex proofs and resource designs already developed. Thus, we refer readers interested in details of the proof to the well-established literature [18] and focus here on the role of $\mathtt{Help}_{\mathrm{on}}$ at a more abstract level.

Prelude (Line 5–8). After achieving the ownership of $\mathsf{isBin}_{\gamma}(l)$ on the source-side (Fig. 10, Line 5–6) by **Assume** (ASSUME-SRC), we can execute multiple target-side operations such as `load`, `alloc`, and `store`. These instructions, appearing in Lines 5–9 in Fig. 7, are attempts to push onto the *main channel*: if the `cas` operation in Line 8 succeeds, this means we have committed our add operation and must update the ghost state accordingly.

Main channel: success. In case of success, we do not need any help of other threads, and should execute **Assume** and **Guarantee** on our own. Is such ghost update possible when there is nothing but `Help.Run` on the source-side? The answer is yes—the simulation rules in Fig. 11 provide corresponding reasoning principles. Specifically, given the goal simulation:

$$r \leftarrow \mathtt{cas}(l,\ h_{\mathrm{old}},\ h_{\mathrm{new}});\ \cdots \lesssim_{\mathcal{E}} \mathtt{Help.Run}(\mathit{jid});\ \cdots$$

we execute the `cas` operation on the target-side and check that it succeeds. Then we apply HELP-RUN to acquire ownership of $\mathsf{Pend}(\mathit{hid})$, an exclusive right to execute the job allocated for *hid*. After executing the initial $\mathcal{Y}_{\mathcal{E}}$ appearing on the source-side without any proof obligation (YIELD-SRC), we are able to apply the rule TRYRUN-PEND. The rest is straightforward: we are able to *update* the ghost resources through the execution of $\mathit{job}(\mathit{jid})$, and we proceed to the continuation with the receipt that the job is done, *i.e.,* $\mathsf{Done}(\mathit{hid},\ \mathit{ret}) \twoheadrightarrow t \lesssim_{\mathcal{E}} s(\mathit{ret})$.

[7] The memory $\mathtt{Mem}_{A}$ and scheduler $\mathtt{NDSch}_{\mathrm{I}}$ modules are omitted.

Main channel: failure (Line 10–12). What is interesting is when the `cas` at Line 8 fails and `Bin.Add` attempts to use the side channel. After allocating the offer node through Lines 10–11, as explained in §4.1, it exposes the offer and waits for some time so that threads attempting to `Bin.Remove` can take it (Line 12).

Side channel: offer taken. In a case where such helping occurs, our thread executing `Bin.Add` discovers that its offer has been taken in Line 13, when `cas` fails, but remember, the helper should also have executed our own ghost update. This is made possible by *sharing our* `Pend`(*hid*) *token through invariants.*

Note the rule TRYRUN-PEND. If one owns `Pend` right before executing `Help.TryRun` on the source-side, one is able to execute the job! Thus, the generous thread that acquired the `Pend` token through the invariant may actually perform the job and put `Done` in the invariant to inform other threads that the offer has been taken, completing the update of the helpee *at the right moment.*

After observing that someone has taken its offer and the state of the offer is set to 1, the remaining proof is straightforward. The `cas` at Line 13 is destined to fail, which allows us to break from the loop at Line 14 and finish the process. Then on the source-side, since we have acquired `Done` from the invariant, we apply TRYRUN-DONE to skip the execution of `Help.TryRun` on the source-side and terminate the simulation proof. In this way, we are able to show that $\texttt{Bin}_\texttt{I}$ contextually refines $\texttt{Bin}_\texttt{M} \circ \texttt{Help}_\texttt{on}$.

Side channel: offer revoked. If the `cas` operations in Line 13 succeeds, this means that unfortunately no thread has succeeded to take the offer we made. We end our verification by coinductive reasoning. We note here that our simulation relation supports FreeSim [4], technique to ensure the soundness of stuttering simulations, although related parameters are omitted throughout the paper.

Summary. We sketched the verification process of $\texttt{Bin.Add}_\texttt{I} \lesssim \texttt{Bin.Add}_\texttt{M}$ and skip the proof of $\texttt{Bin.Remove}_\texttt{I} \lesssim \texttt{Bin.Remove}_\texttt{M}$, since the most interesting part of the proof is the interaction with $\texttt{Help}_\texttt{on}$, but we explained it in previous paragraphs.

A brief proof sketch of Theorem 4.1: to prove $\texttt{Help}_\texttt{on} \circ \texttt{NDSch}_\texttt{I} \sqsubseteq_{ctx} \texttt{Help}_\texttt{off} \circ \texttt{NDSch}_\texttt{I}$, we reorder the source-side sequence of scheduled threads. Specifically, when a job is being executed in $\texttt{Help}_\texttt{on}$ by the helper thread, we schedule the helped thread in $\texttt{Help}_\texttt{off}$ to execute the job at the same moment. We refer interested readers to our artifact [2] for further details.

We note that to the best of our knowledge, ConCRIS is the first separation logic based refinement framework to support helping of I/O operations.

5 Prophecy Variables in ConCRIS

In §4, we showed how the module system enables a global reasoning in a thread-local fashion through the helping module $\texttt{Help}_\texttt{on}$. In this section, we present another module for a *temporally* global reasoning, namely the *prophecy module.* We first briefly review the motivations of prophecy variables and point out shortcomings of previous works (§5.1). We proceed to present our solution reusable across languages (§5.3) and an interesting countexample of why a naive support of prophecy variables in ConCRIS is impossible (§5.3).

5.1 What are prophecy variables?

As Abadi and Lamport [1] showed that we need *prophecy variables* for *future-dependent* reasoning for certain type of programs, Jung et al. [17] add support of prophecy in the realm of separation logic, *i.e.* Iris, and prove the linearizability of certain FCDs such as RDCSS [10]. In this section, we employ the introductory example, *lazy coins*, for a brief overview of prophecy variables in Iris and demonstration of the usage of our prophecy variables.

```
1  newCoin() ≜
2    let v = ref(None);
3    let p = Proph.New;
4    (val = v; p = p)
5  readCoin(c) ≜
6    match !c.val with
7    | Some(b) ⇒ b
8    | None    ⇒ let r = choose(𝔹);
9                  c.val = Some(r);
10                 Resolve c.p to r; r
11   end
```

Fig. 12. HeapLang Implementation $\texttt{Coin}_\texttt{I}$ of lazy coins (excerpt from Jung et al. [17])

Lazy coins. Fig. 12 shows an implementation of a 'coin' library, $\texttt{Coin}_\texttt{I}$. We first note that it is written in HeapLang, an example language of Iris, and not ITrees. This is intentional to show the benefits regarding language generality we achieve via ITrees, but please ignore this aspect for now.

The library is quite simple: `newCoin()` allocates a pair of values, and `readCoin`(c) with coin c reads a boolean value from the coin. What is interesting is that the coin is *lazily* tossed: the coin value is not determined in `newCoin`, but in the first `readCoin` that takes place after `newCoin`. That is, if `readCoin` observes that the toin has not yet been tossed (Line 8), it tosses the coin then. With this in mind, observe the triples we wish to prove in Iris:

$$\{\mathsf{True}\}\ \texttt{newCoin}()\ \{v.\ \mathsf{Coin}(v)\} \qquad \text{(NEWCOIN-HT)}$$

$$\{\mathsf{Coin}(v)\}\ \texttt{readCoin}()\ \{v.\ \mathsf{Coin}(v)\} \qquad \text{(READCOIN-HT)}$$

We run into a problem if we try to verify the Hoare triples naively. In verifying `newCoin`, we need a mechanism to somehow *predict the future* and see what the value of the coin will be in the first toss of `readCoin`! We will then be able to give the user *up front* the corresponding $\mathsf{Coin}(v)$ with the value v we saw in the future, and when the program actually tosses the coin, the user can be guaranteed that the tossed value equals the value of the Coin resource.

This is the exact reason of the creation of prophecy variables—if there is an auxilliary variable that records *future nondeterminism* of the implementation, we are able to show specific refinements or Hoare triples that required future-dependent reasoning. Including the verification of linearizability of RDCSS [10] and Herlihy-Wing queues [13], prophecy variables were deployed for other SL projects [3, 7, 24] too, proving its usefulness.

$$\{\mathsf{True}\}\ \texttt{Proph.New}\ \{p.\ \exists vs.\ \mathsf{Proph}(p,\ vs)\} \qquad \text{(NEWPROPH)}$$

$$\{\mathsf{Proph}(p,\ vs)\}\ \texttt{Resolve}\ c.p\ \texttt{to}\ r\ \{\exists vs'.\ vs = ((), w) :: vs' \ *\ \mathsf{Proph}(p,\ vs')\} \qquad \text{(RESOLVE)}$$

We end this section by proving NEWCOIN-HT with related Hoare triples for prophecy. In proving NEWCOIN-HT, we take the ownership of $v \mapsto \mathsf{None}$ and $\mathsf{Proph}(p,\ vs)$ by NEWPROPH. It is enough to define Coin to incorporate two cases: either the coin has not been tossed and it is prophecied to be tossed to v, or the coin has been tossed to v. The prophecy variable p is essential when switching from the first case to the second in `readCoin`: we can exclude the case where the actual value returned from the coin toss is not v, safely guaranteeing that the return value is v.

5.2 Decoupling prophecy variables from the programming language

Let us assume a situation where one wants to reason with Iris, about programs written in an assembly language, say $\mathsf{L}_{\mathtt{Asm}}$ that requires the use of prophecy variables.

What should be done for it? First of all, note that since Iris is a language-agnostic program logic, we have to instantiate Iris with $\mathsf{L}_{\mathtt{Asm}}$. However, we cannot instantiate Iris with naive $\mathsf{L}_{\mathtt{Asm}}$—we first have to come up with a new language for prophecy! Indeed, HeapLang has augmented language

$Pro : Type$ $Obs : Type$ Consistent : $Pro \times List\, Obs \longrightarrow Prop$

```
1 module Proph_A.
2 def Proph.New(pid : String × Any) ≜
3   Assume(Free({pid}));
4   Guarantee(∃ p. Proph(p, []));
5   return null;
```

```
1 def Proph.Resolve(pid : String × Any, o : Obs) ≜
2   (p, l) ← take(Pro × List Obs);
3   Assume(Proph(p, l));
4   Guarantee(Consistent(p, o :: l) ∗ Proph(p, o :: l));
5   return null;
```

Fig. 13. Prophecy module $\mathsf{Proph}_\mathsf{A}$.

constructs, **Proph.New** and **Resolve** $c.p$ **to** r for the support of prophecy. Note that the set of values should be extended for prophecy variables p and poison values ☣ (a dummy value made for *erasing* prophecy operations, which will be explained soon), implying every operations have to be redefined. There should be an additional mechanism to store prophecy variables in the heap, too. Finally, after augmenting L_Asm for accounting prophecy variables, namely $\mathsf{L}_{\mathsf{Asm}_\mathsf{Prp}}$, we have to define the *weakest precondition* connective, a core logical construct of Iris, and prove appropriate reasoning rules for each operations of $\mathsf{L}_{\mathsf{Asm}_\mathsf{Prp}}$.

As Vistrup et al. [32] pointed out and showed, *reusability* of the logic across languages is a useful factor of a verification framework. However, we see that all the tedious proofs and language design one has to go through for a reasoning with prophecy variables make the current approach in Iris less scalable. Although the need for this has been identified by Vistrup et al., they left the support of prophecy variables as a future work, leaving the reusability aspect unsatisfactory.

Prophecy modules. Fig. 13 presents the *prophecy module*, $\mathsf{Proph}_\mathsf{A}$ of ConCRIS. Recall that in our treatment of helping, we provided the ability to execute source-side specifications of other threads, which was enabled by *linking helping modules* with our own specification and inlining them.

$\mathsf{Proph}_\mathsf{A}$ follows a similar pattern: the user *links* $\mathsf{Proph}_\mathsf{A}$ with the implementation and inlines the functions of $\mathsf{Proph}_\mathsf{A}$ to reason with prophecy variables. Note that this is the exact pattern our LAIS is exploited. The user inlines the spec, gives or takes the ownership of resources via **Assume** and **Guarantee** operators.

For a language-generic support, we parameterize the prophecy module with types of prophecy variables and observations, *Pro* and *Obs*. In this way, we can freely instantiate *Obs* for any type of values depending on the language we use, and design *Pro* accordingly. What connects *Pro* and *Obs* is Consistent. `Proph.Resolve` guarantees at Line 4 that the observation o is consistent with the prophecy variable p we instantiated. For example, in verifying the $\mathtt{Coin}_\mathtt{I}$ example, we could instantiate both *Pro* and *Obs* with boolean type $\mathbb{B}$, and define:

$$\mathsf{Consistent}(p,\, l) \triangleq l = [] \;\vee\; \exists\, rest.\; l = rest \mathbin{+\!\!+} [p]$$

After taking the ownership of the prophecy variable $\mathsf{Proph}(p,\, [])$ from `Proph.New` with empty history of observations, we are able to predict the future observation by case analysis on p: the observation made in the initial `readCoin` should match the predicted value by the second case of Consistent. That is, given $\mathsf{Consistent}(p,\, [o])$, we are able to conclude $p = o$.

Besides the parameterization of types *Pro* and *Obs*, one notable difference from Iris prophecy variables is the existence of *prophecy identifiers pid*. Our `Proph.New` takes *pid* of type $String \times Any$ as a unique identifier for the sequence of future observations. Since the user will be inlining

```
1 module Proph_I.
2 def Proph.New(pid) ≜
3   return null;
4 def Proph.Resolve(pid, o) ≜
5   return null;
```

```
1 module Coin_I.
2 def newCoin() ≜
3   l ← alloc(1);
4   Proph.New(('coin', l));
5   return l
6 def readCoin(c) ≜ ···
```

```
1 module Coin_0.
2 def newCoin() ≜
3   l ← alloc(1);
4   return l
5 def readCoin(c) ≜ ···
```

Fig. 14. The erased prophecy module $\mathtt{Proph}_\mathtt{I}$ and our ConCRIS implementation $\mathtt{Coin}_\mathtt{I}$ of lazy coins.

`Proph.New` in the target-side, it is the obligation of the user to show that v is an indeed unique identifier by proving $\mathsf{Free}(\{p\})$. [8]

The reason we let the user designate the identifier of the prophecy variable is to ease the *burden of program annotation for prophecy*. Taking a look back to $\mathtt{Coin}_\mathtt{I}$, we find out that it is not truely the program we wished to verify—it is annotated with *ghost codes* that store and read prophecy variables to associate them with actual program values! In this way, we can resolve the prophecy variable to the observation made in `readCoin`, but make the program have different semantics with the original one, say $\mathtt{Coin}_0$. For example, `newCoin` in $\mathtt{Coin}_\mathtt{I}$ is a program that allocates a *pair*, which should have been a singleton in $\mathtt{Coin}_0$. If the end goal, namely the safety property of $\mathtt{Coin}_0$ is desired, we have to manually prove that the behavior of $\mathtt{Coin}_0$, for example, refines $\mathtt{Coin}_\mathtt{I}$.

Rather than letting the prophecy module generate identifiers for the variable, we let the user identify prophecies with values that appear in the program, *e.g.* pointers to data structures. In this way, we do not have to treat prophecy variables as actual variables and modify the code, but rather only call `Proph.Resolve` with appropriate annotations. As we will see with the adequacy theorem of prophecy modules, this will ease the burden of showing that $\mathtt{Coin}_0$ is related with $\mathtt{Coin}_\mathtt{I}$.

5.3 The adequacy of the prophecy module

Suppose we showed that $\mathtt{Coin}_\mathtt{I} \circ \mathtt{Proph}_\mathtt{A} \sqsubseteq_{ctx} \mathtt{Coin}_\mathtt{A}$ in ConCRIS with the help of $\mathtt{Proph}_\mathtt{A}$. [9] This, of course, is not the end—what we actually want is: $\mathtt{Coin}_0 \sqsubseteq_{ctx} \mathtt{Coin}_\mathtt{A}$. The missing link here is a chain of refinement going from $\mathtt{Coin}_0$ to $\mathtt{Coin}_\mathtt{I} \circ \mathtt{Proph}_\mathtt{A}$. Can we fill in the gap?

The answer is yes. Fig. 14 presents the *erased prophecy module* $\mathtt{Proph}_\mathtt{I}$. Consider the following chain of refinements:

$$\mathtt{Coin}_0 \sqsubseteq_{ctx} \mathtt{Coin}_\mathtt{I} \circ \mathtt{Proph}_\mathtt{I} \sqsubseteq_{ctx} \mathtt{Coin}_\mathtt{I} \circ \mathtt{Proph}_\mathtt{A} \sqsubseteq_{ctx} \mathtt{Coin}_\mathtt{A}$$

We already have the last refinement through reasoning with prophecy. The first refinement is trivial: observe that $\mathtt{Coin}_\mathtt{I}$ is basically $\mathtt{Coin}_0$ with appropriate ghost prophecy codes (*i.e.,* calls to $\mathtt{Proph}_\mathtt{I}$, which do nothing) inserted. It is in this sense what we meant our prophecy modules *ease the burden* of program annotation: insertion of prophecy codes do not change the behavior.

The last piece of the puzzle is left: namely $\mathtt{Proph}_\mathtt{I} \sqsubseteq_{ctx} \mathtt{Proph}_\mathtt{A}$. This is a similar situation we already saw in §4.2 with helping modules. That is, a local reasoning principle was provided to its users, and the actual global reasoning was hidden in the refinement proof of each helping modules. Unfortunately, a naive statement, namely $\mathtt{Proph}_\mathtt{I} \sqsubseteq_{ctx} \mathtt{Proph}_\mathtt{A}$, *does not hold in general in ConCRIS*.

[8] Free is a resource designed to represent the unique ownership of the name p. We do not present the formal definition here for space reasons, and refer an interested reader to our Rocq development.

[9] We omit the specification $\mathtt{Coin}_\mathtt{A}$ module of $\mathtt{Coin}_\mathtt{I}$, but it will be evident now how $\mathtt{Coin}_\mathtt{A}$ would look like. We refer the readers to the Rocq development.

Intersection and union do not commute. We present the semantic model of **take** and **choose** operators, key operators for the theory of CRIS, to illustrate the problem.

$$beh(x \leftarrow \textbf{choose}(X);\ K\ x) \triangleq \bigcup_{x \in X} beh(K\ x) \qquad beh(x \leftarrow \textbf{take}(X);\ K\ x) \triangleq \bigcap_{x \in X} beh(K\ x)$$

The **choose** operator is a standard nondeterministic operator. The behavior of the program is the set union of the continuations with all possible values $x \in X$. The **take** operator is a mathematical dual to **choose**, known as *angelic nondeterminism* in the literature. The behavior of the program is the set intersection of all $K\ x$.

Essentially, the proof of $\texttt{Proph}_\texttt{I} \sqsubseteq_{ctx} \texttt{Proph}_\texttt{A}$, or any proofs of the adequacy of prophecy variables, reduce to *pulling nondeterminisms forward from the future.* All possible sequences of resolutions can be thought as if determined with the creation of prophecy variables, *i.e.* `Proph.New`, while the actual resolutions happen far after the creation. However, pulling forward the resolutions in the presence of **take** is in general unsound—one may call this phenomena, a *prophet's dilemma.*

Consider the following example:

$$beh\left(\begin{array}{l} b_1 \leftarrow \textbf{take}(\mathbb{B}); \\ b_2 \leftarrow \textbf{choose}(\mathbb{B}); \\ \textbf{if}(b_1 = b_2)\ \textbf{IO}(42); \end{array}\right) = \{\textbf{IO}(42)\} \qquad beh\left(\begin{array}{l} b_1 \leftarrow \textbf{choose}(\mathbb{B}); \\ b_2 \leftarrow \textbf{take}(\mathbb{B}); \\ \textbf{if}(b_1 = b_2)\ \textbf{IO}(42); \end{array}\right) = \emptyset$$

Suppose we wish to predict the nondeterminism of the first function (*i.e.*,**choose**), and thus pulled forward the **choose** as in the second function which provides prophecy reasoning to the user early. The problem here is that the second program has less behavior than the first, which means that properties proven of the behavior of the second program may not apply to the behavior of the first, making a future-dependent reasoning unsound.

Our solution. We are not at a dead-end yet: it is still possible to use prophecy variables for programs that do not have angelic nondeterminism. Indeed, since **take**s only arise in LAIS and actual program values in the implementation are what we wish to prophesy, we restore the power of propehcy variables by an auxilliary compilation function, $\downarrow \in itree\, E_{mod}\ Any \to itree\, E_{mod}\ Any$, where $\downarrow\ i = i$ except for $\downarrow\ (x \leftarrow \textbf{take}(X);\ K\ x) = \textbf{take}(\emptyset)$. Note that $\textbf{take}(\emptyset)$ exhibits *undefined behavior* (*i.e.,* all behaviors including *Error*), since an indexed intersection with empty set of indices is the whole set.

Thus our adequacy theorem is as follows:

Lemma 5.1. $\forall ctx,\ (\downarrow ctx) \circ \texttt{Proph}_\texttt{I} \sqsubseteq (\downarrow ctx) \circ \texttt{Proph}_\texttt{A}$

Proof. It is possible to extract all resolutions from the implementation given its trace: we associate the prophecy identifier and resolutions. We refer interested readers to our artifact [2]. □

6 Hybrid Schedulers and Heterogeneous Memory Model

This section presents our scheduler modules that capture multi-node concurrency (§6.1). We also allow users to define and exploit custom scheduling policies (like round-robin) beyond the standard nondeterministic approach, which is vital for modeling specialized systems such as embedded kernels. We provide an example with hybrid schedulers and memory models (§6.2).

6.1 Scheduler as a Module

Metatheory extension. Fig. 15 presents the extended metatheory of ConCRIS (Fig. 2). ITrees are enriched with two primitive concurrency events: **Spawn** for thread creation and **Yield** for explicit control transfer. The **Spawn** event returns an identifier for the newly created thread, while

$$E_{ctrl}(X) \triangleq \cdots \uplus \{\textbf{Spawn}\ fn\ arg \mid fn \in String,\ arg \in Any\} \uplus \{\textbf{Yield}\ tid \mid tid \in \mathbb{N}\}$$

$$\textsc{spawn}\ \frac{\forall tid \in \mathbb{N}.\ K_t\ tid \lesssim K_s\ tid}{(\textbf{Spawn}\ fn\ arg \gg\!= K_t) \lesssim (\textbf{Spawn}\ fn\ arg \gg\!= K_s)} \qquad \textsc{yield}\ \frac{\mathcal{I}(st_s,\ st_t) \qquad K_t\ () \lesssim K_s\ ()}{(\textbf{Yield}\ tid \gg\!= K_t) \lesssim (\textbf{Yield}\ tid \gg\!= K_s)}$$

Fig. 15. Extended definitions and simulation rules of ConCRIS from CRIS.

```
1  module NDSch_I .
2  var pool : list(ℕ × option Any)
3  var tid_cur : ℕ
4  def spawn(fn, arg) ≜
5    stid_new ← Spawn (doSpawn) (fn, arg);
6    let mtid_new := len(pool) in
7    put (pool , pool{mtid_new ↦ (stid_new, None)});
8    return mtid_new ;
9  def doSpawn(fn, arg) ≜ ···
10 def yield() ≜
11   mtid_nxt ← choose([0..len(pool)]);
12   let stid_nxt := fst(pool[mtid_nxt]) in
13   put (tid_cur, mtid_nxt );
14   Yield (stid_nxt );
15   return null;
```

```
1  module RRSch_I .
2  var pool : list of (ℕ × option Any)
3  var tid_cur : ℕ
4  def spawn(fn, arg) ≜
5    stid_new ← Spawn (doSpawn) (fn, arg);
6    let mtid_new := len(pool) in
7    put (pool, pool{mtid_new ↦ (stid_new, None)});
8    return mtid_new;
9  def doSpawn(fn, arg) ≜ ···
10 def yield() ≜
11   let mtid_nxt := (tid_cur + 1) % len(pool) in
12   let stid_nxt := fst(pool[mtid_nxt]) in
13   put (tid_cur, mtid_nxt);
14   Yield (stid_nxt);
15   return null;
```

Fig. 16. Implementation of Scheduler Modules, $\mathtt{NDSch}_\mathtt{I}$ and $\mathtt{RRSch}_\mathtt{I}$ (simplified version).

Yield accepts a natural number identifying the next thread to be scheduled. Crucially, **Yield** is deterministic: the next thread is explicitly specified rather than chosen nondeterministically.

This design choice requires justification, as cooperative concurrency typically leaves the scheduling decision implicit. The key insight is *separation of mechanism from policy*. By making the next thread explicit, the metatheory provides a minimal primitive upon which scheduler modules can implement arbitrary thread selection policies.

For example, consider what a round-robin scheduler must express: "schedule the thread whose identifier follows the current thread in the queue". With nondeterministic **Yield**, there would be no way to program a round-robin scheduling policy and give the threads corresponding reasoning rules, with the intended policy lost in the semantics. The same limitation would apply to priority-based, fair, or any policy that needs to be distinguished from plain nondeterministic schedulers. Deterministic **Yield** thus enables definitions of diverse scheduling strategies.

Scheduler modules. Fig. 16 present two concrete scheduler modules that build on our primitve events. The NDSch module implements nondeterministic scheduling. It maintains two variables: *pool* records system thread identifiers (returned from **Spawn**) and their return values, while tid_{cur} stores the currently executing thread's module-level identifier.

The spawn function creates a thread by invoking a helper doSpawn that executes the given function and stores its result, then records the thread in the pool and returns its module-level identifier. The yield function selects an arbitrary thread from the pool and transfers control via the deterministic **Yield** primitive.

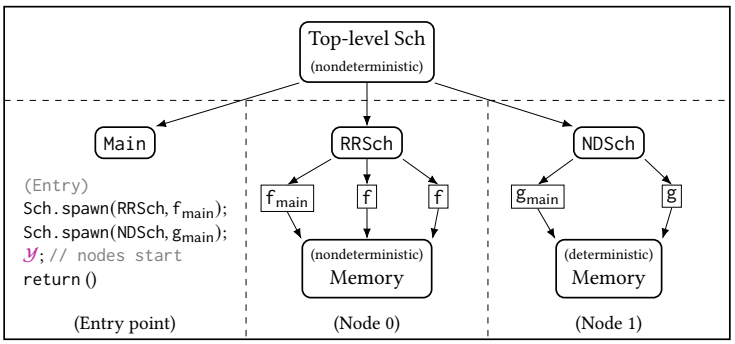

Fig. 17. Structure of the example illustrating the hierarchical scheduler.

The RRSch module implements round-robin scheduling. Its structure mirrors NDSch, but `yield` computes the next thread deterministically based on the current thread's identifier. This difference—hidden from the users of the scheduler module—demonstrates how the deterministic **`Yield`** primitive supports multiple scheduling policies.

Reasoning Principles. We provide separation logic style specifications for schedulers: [10]

$$\{\{\texttt{Tid} * \texttt{PRE}\}\ \mathit{fn}(\mathit{arg})\ \{v.\texttt{Tid}\} * \texttt{PRE}\}\ \textbf{NDS.spawn}\ (\mathit{fn}, \mathit{arg})\ \{v.\exists \mathit{mtid}_{\text{new}}.\, v = \mathit{mtid}_{\text{new}}\} \qquad \text{(NDS-SPAWN)}$$

$$\{\texttt{Tid}\}\ \textbf{NDS.yield}\ ()\ \{v.\, v = () * \texttt{Tid}\} \qquad \text{(NDS-YIELD)}$$

$$\left\{\left\{\texttt{Tid} * \texttt{INV}_{\texttt{Invs++[I]}}\right\}\ \mathit{fn}(\mathit{arg})\ \{\texttt{Tid}\} * \texttt{INV}_{\texttt{Invs}}\right\}\ \textbf{RRS.spawn}\ (\mathit{fn}, \mathit{arg})\ \left\{v.\, \texttt{INV}_{\texttt{Invs++[I]}}\right\} \qquad \text{(RRS-SPAWN)}$$

$$\{\texttt{INV}_{\texttt{I}} * \texttt{I}_{\text{ctid}}\}\ \textbf{RRS.yield}\ ()\ \left\{v.\, v = () * \texttt{INV}_{\texttt{I}} * \texttt{I}_{\text{prev of ctid}}\right\} \qquad \text{(RRS-YIELD)}$$

To invoke spawn or `yield`, threads must own `Tid`, which represents exclusive control of execution. This can be understood as a resource: a thread relinquishes control by giving up the ownership when yielding, and the scheduled thread acquires it when resumed. The RRSch additionally uses ownership `INV` to associate thread-specific invariants with each managed thread, enabling finer-grained reasoning about scheduler guarantees.

Defining Yield for User Code. One technical challenge remains. The simulation rule YIELD-TGT from Fig. 6 allows stepping the target side independently while leaving the source side unchanged. This rule is essential for verifying concurrent programs, but it cannot be derived from a single `yield` call. The issue is that primitive ConCRIS simulation rules for **`Yield`** maintain lockstep correspondence between source and target steps. A single `yield` call would require simultaneous progress on both sides, preventing the independent target-side steps that YIELD-TGT enables.

The solution is to introduce unbounded nondeterminism. We define the user-level $\mathcal{Y}$ as:

$$\mathcal{Y} \triangleq \texttt{while(*)\{ yield() \}}$$

The `while(*)` construct represents nondeterministic iteration (zero or more executions), allowing the target to take multiple scheduling steps while the source remains at the yield point. Notably, while the loop can iterate infinitely in principle, this poses no issue for simulation: ConCRIS's refinement relation permits such non-terminating behavior in the target as long as it refines the source, meaning unbounded yielding remains a valid implementation choice.

6.2 Proof with Hierarchical Structured Scheduler

Although Fig. 16 gives readers the intuition for how custom schedulers can be implemented, those examples do not yet demonstrate hierarchical composition. Fortunately, extending schedulers for it is straightforward with carefully designed ownership and an additional `init` function that manages the thread pool and communicates with the parent scheduler. The wrapped schedulers mostly maintain the same interface as their non-hierarchical counterparts. We have mechanized hierarchical extensions of both `NDSch` and `RRSch` in our Rocq artifact, where readers can examine the details. Here, we illustrate the hierarchical approach through a high-level explanation that demonstrates ConCRIS's reasoning power.

System structure. Fig. 17 shows the overall architecture of our example system. The entry point (`Main`) initializes two nodes: Node 0 uses a round-robin scheduler (`RRSch`), while Node 1 uses a nondeterministic scheduler (`NDSch`). Since nodes can execute in parallel, their interleaving is modeled by a top-level nondeterministic scheduler.

Round-robin reasoning. Consider Node 0. `RRSch` acts as the node's top-level scheduler, providing each thread with the guarantee that it holds a distinct *iProp* before yielding. This stronger assumption enables much finer-grained specifications for functions.

For example, suppose $\mathtt{f}_{\mathsf{main}}$ spawns two threads that share a pointer initially storing 0. Each spawned thread executes function `f`, which atomically: (1) reads value v from the pointer, (2) stores $v + 1$ back to the pointer, and (3) prints $(tid - v)$ via I/O, where tid is the thread identifier assigned by `RRSch`. Because `RRSch` assigns thread IDs sequentially starting from 0, the I/O is deterministic, unlike what would occur under `NDSch`.

Importantly, the top-level scheduler can still yield between any two lines of code, even within what we consider an "atomic sequence" at Node 0's level. This preserves parallel execution while maintaining the reasoning principles afforded by the round-robin policy within the node.

Heterogeneous memory models. The difference between Node 0 and Node 1 reveals another dimension of ConCRIS's generality: heterogeneous memory models. Node 0 employs a memory system that chooses block numbers nondeterministically, while Node 1 uses deterministic allocation. This demonstrates that ConCRIS can serve as a framework for verifying hierarchical programs with multiple schedulers, diverse scheduling policies, and even heterogeneous memory models—all within a unified reasoning system.

We also note that we implemented a variant of the vRC11 [22] memory model with relaxed memory consistency (RMC) and gave corresponding LAIS to them that largely resembles iRC11 [6] logic. It is not included in the hierarchical example yet, but incorporating it for multi-node environment will be straightforward. Moreover, to gain confidence, we verified a message-passing client on top of vRC11.

7 Related Works

Logical atomicity. Although a naive LAT presented in §3.1 is an underspecification, we note that it is possible to specify $I_{\mathtt{Queue}}$ by devising a notion of *nested LATs* and give corresponding reasoning rules in Iris. However, LAIS provides more benefits over logical atomicity of previous separation logics [5, 18].

First, LAIS inherits the strengths of CRIS in an *open verification*. In specifying programs with I/O or calls to functions that may possibly be unverified, LAIS can concisely specify the behavior of a concurrent library. For example, think of specifying a function f which takes an integer input n

[10]They are lockstep simulation rules to execute both function calls at the source and the target, in reality.

from the environment and pushes n number of elements to a stack. While the LAIS of f naturally specifies this behavior, one should come up with a mechanism such as prophecy variables to fix the value of n *up front* to correctly identify the number of ghost updates for the specification.

Similar argument applies to verification in programming languages with *nondeterministic operators* like our **choose**. LAIS works seamlessly even if the behaviors of a library depend on such nondeterminism determined dynamically. Last but not least: LAIS can be again verified to a more abstract, simplified LAIS, admitting an *incremental verification*.

Prophecy variables. Detailed explanation of prophecy variables in Iris by Jung et al. [17] is provided in §5.1. Especially, we support language-generic prophecy variables, which was left as a future work in Program Logics à la Carte [32].

Reasoning Principles for Concurrency. Iris requires global invariants to hold before physical atomic operations instantiated in language definitions. Consequently, VMSL [23], which verified hypervisors under cooperative multitasking, needs tweaking weakest precondition for multiple instruction reasoning. In contrast, ConCRIS naturally allows users to reason about multiple instructions between consecutive $\mathcal{Y}$s. Additionally, since physical atomicity is not fixed in the language, ConCRIS can prove properties when nodes use different languages.

Hierarchical Structure. Distributed systems are representative examples requiring hierarchical structure. Aneris [20] verified distributed systems by extending HeapLang to `AnerisLang` with well-structured semantics and ownership design. Trillium [29] verifies distributed systems based on *intensional refinements* using LTS specifications. ConCRIS distinguishes itself by enabling users to verify hierarchical structures through modular scheduler definitions. Unlike Trillium, ConCRIS provides specifications that can be linked with other programs, just as CRIS does. Moreover, users can freely extend both depth and width of application structure without additional modifications, whereas other approaches require manual extensions beyond the presented 2-depth structure in §6.

References

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

[2] Anonymous. 2025. Artifact for ConCRIS: Imaginary Specifications for Fine-grained Concurrency. Anonymous submission for double-blind review.

[3] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *USENIX Symposium on Operating Systems Design and Implementation*. https://api.semanticscholar.org/CorpusID:259265778

[4] Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. Stuttering for Free. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (Oct. 2023), 1677–1704. doi:10.1145/3622857

[5] Pedro Da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming*. Vol. 8586. Springer Berlin Heidelberg, Berlin, Heidelberg, 207–231. doi:10.1007/978-3-662-44202-9_9 Series Title: Lecture Notes in Computer Science.

[6] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt meets relaxed memory. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–29. doi:10.1145/3371102

[7] Paulo Emílio De Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. doi:10.1145/3371101

[8] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. 2014. Quiescent Consistency: Defining and Verifying Relaxed Linearizability. In *FM 2014: Formal Methods*. Vol. 8442. Springer International Publishing, Cham, 200–214. doi:10.1007/978-3-319-06410-9_15 Series Title: Lecture Notes in Computer Science.

[9] Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021), 6598. doi:10.46298/lmcs-17(3:9)2021

[10] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing*. Vol. 2508. Springer Berlin Heidelberg, Berlin, Heidelberg, 265–279. doi:10.1007/3-540-36108-1_18 Series Title: Lecture Notes in Computer Science.

[11] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2004. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, Barcelona Spain, 206–215. doi:10.1145/1007912.1007944

[12] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2021. *The art of multiprocessor programming* (second edition ed.). Esevier, Morgan Kaufmann Publishers, Cambridge, MA, United States.

[13] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. doi:10.1145/78969.78972

[14] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, Philadelphia PA USA, 59–72. doi:10.1145/2103656.2103666

[15] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, Nara Japan, 256–269. doi:10.1145/2951913.2951943

[16] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. doi:10.1017/S0956796818000151

[17] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The future is ours: prophecy variables in separation logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. doi:10.1145/3371113

[18] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. *ACM SIGPLAN Notices* 50, 1 (May 2015), 637–650. doi:10.1145/2775051.2676980

[19] Yonghee Kim, Taeyoung Yoon, Sanghyun Yi, Jaehyung Lee, Soonwon Moon, Yeji Han, Seonho Lee, Taeyoung Rhee, Yujin Im, Donghyun Nam, Jieung Kim, and Chung-Kil Hur. 2025. *CRIS: The Power of Imagination in Software Verification*. Technical Report SFLab-2025-001. SFLab. https://sf.snu.ac.kr/technical-reports/files/SFLab-2025-001.pdf

[20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A mechanised logic for modular reasoning about distributed systems. In *European Symposium on Programming*. Springer International Publishing Cham, 336–365.

[21] Dongjae Lee, Janggun Lee, Taeyoung Yoon, Minki Cho, Jeehoon Kang, and Chung-Kil Hur. 2025. Lilo: A Higher-Order, Relational Concurrent Separation Logic for Liveness. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (April 2025), 1267–1294. doi:10.1145/3720525 Publisher: Association for Computing Machinery (ACM).

[22] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. 2023. Putting Weak Memory in Order via a Promising Intermediate Representation. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 1872–1895. doi:10.1145/3591297

[23] Zongyuan Liu, Sergei Stepanenko, Jean Pichon-Pharabod, Amin Timany, Aslan Askarov, and Lars Birkedal. 2023. VMSL: A Separation Logic for Mechanised Robust Safety of Virtual Machines Communicating above FF-A. *Proc. ACM Program. Lang.* 7, PLDI, Article 165 (June 2023), 25 pages. doi:10.1145/3591279

[24] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, San Diego CA USA, 841–856. doi:10.1145/3519939.3523704

[25] Yusuke Matsushita and Takeshi Tsukada. 2025. Nola: Later-Free Ghost State for Verifying Termination in Iris. *Proceedings of the ACM on Programming Languages* 9, PLDI (June 2025), 98–124. doi:10.1145/3729250

[26] Youngju Song and Minki Cho. 2025. CCR 2.0: High-level Reasoning for Conditional Refinements. doi:10.48550/arXiv.2507.04298 arXiv:2507.04298 [cs].

[27] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 1121–1151. doi:10.1145/3571232 Publisher: Association for Computing Machinery (ACM).

[28] The Rocq Development Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.15149629

[29] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* 8, POPL, Article 9 (Jan. 2024), 32 pages. doi:10.1145/3632851

[30] Aaron J. Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming*

languages. ACM, Rome Italy, 343–356. doi:10.1145/2429069.2429111

[31] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Philadelphia PA USA, 100–115. doi:10.1145/3497775.3503689

[32] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 300–331. doi:10.1145/3704847

[33] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. doi:10.1145/3371119