Modular Responsiveness Verification of Rust Async Runtimes

Yanze Li, Ivan Beschastnikh, and Alexander J. Summers

University of British Columbia, Canada

Abstract. Asynchronous (async) programming is a popular paradigm for managing concurrency. Languages that provide async support typically have a runtime to manage asynchronous executions. These runtimes are critical infrastructure, yet there has been no prior work on verifying the correctness of async runtimes. One reason is that a property that users care most about from an async runtime is a liveness property: tasks submitted to the runtime eventually make progress. Verifying liveness is challenging for libraries that are both concurrent and highly optimized. We present a lightweight and modular proof technique for verifying eventual progression guarantees for Rust async runtimes. We describe this technique in the context of a simple language based on Rust and Rust's async model. We then realize this proof technique as a set of static analyses for Rust and use these to verify eventual progression of several key components of multiple Rust async runtime implementations.

Keywords: Verification · Static Analysis · Asynchronous Programming

1 Introduction

To take advantage of multi-core CPU resources, modern programs make extensive use of asynchronous programming, which relieves developers from needing to manage concurrency themselves. Asynchronous programming is usually supported by an *async runtime* and such runtimes exist for most languages, including Java [21,10], JavaScript [20], Python [23], and Rust [25,28].

An async runtime takes care of managing a graph of inter-dependent user-defined asynchronous tasks, some of which may be waiting on external events, and others that may be waiting on the result of other tasks. Async runtimes vary in complexity, but even a simple runtime must keep track of tasks, wait and observe relevant events, and notify tasks to continue executing. These runtimes and their correctness from a user's perspective are the focus of this paper.

A critical *responsiveness property* of async runtimes that users depend on is *eventual progression*: assuming correct user code and a correct underlying operating system, every user-level async task managed by the runtime should eventually get to make progress (see Section 3).

The eventual progression property is the cornerstone of any async runtime, but is also challenging to verify. This is because a typical async runtime manages a collection of async tasks, and these tasks may depend on a variety of

external events. Async runtimes are also usually concurrent and contain multiple components that communicate with one another. In addition to this inherent complexity of async runtimes, languages such as Rust and OCaml do *not* ship an async runtime as part of the languages but only provide an interface for implementing async runtimes; users can use different runtime implementations simply by importing different async libraries. This means that any formal techniques aimed at async runtimes for such languages must be able to handle a diversity of alternative implementations in practice.

In this paper, we present a lightweight verification technique that allows programmers to verify eventual progression for practical and diverse async runtime implementations. To the best of our knowledge, this is the first work to study the verification of liveness guarantees of async runtimes.

Rust async runtimes consist of three major parts: 1. async tasks, which include primitive async tasks provided by async runtimes (called *pollables*) and user-defined async functions using pollables, 2. Rust's async/await syntax and compiler support, which compile async functions to state-machine representations, and 3. *reactors*, which manage running async tasks by monitoring signals, waking up tasks when signals are observed, and repeating until all tasks complete. Assuming correctness of the Rust compiler, eventual progression boils down to the behaviors of the sets of runtime-provided pollables and reactors.

Our work relies on three key insights. First, we adopt a *task-centric view* and find that it is sufficient to consider verification of eventual progression for a single arbitrary task, instead of reasoning about a collection of tasks. Second, we use runtime component boundaries for a natural and modular decomposition of our verification goal. Finally, we observe that runtime components make use of implicit and, most importantly, *monotonic* events to determine how to deal with a task at any particular point in time. These events are usually internal to the runtime and are key to establishing eventual progression.

We formalize these three insights in a core language $\mathcal{L}_{\mathsf{ART}}$ with an operational semantics for the implementation of pollables and reactors. We then define static analyses for $\mathcal{L}_{\mathsf{ART}}$ whose verification conditions soundly imply eventual progression of the runtime. Grounded in our formalism, we design a lightweight specification language and verification technique for Rust called Jackdaw. Developers of async runtimes can specify the necessary conditions for pollables to be responsive, and annotate their implementation in terms of observations on events. We use Jackdaw to verify eventual progression of pollable and reactor components in smol [25], a commonly used Rust async runtime library, and tokio [28], the most widely used production runtime. We show that our model and analyses handle a variety of real-world components with two modest extensions.

Our evaluation shows that pollable implementations can be verified with an average specification-to-code ratio of less than 30%, and can detect manually-seeded responsiveness bugs of various kinds.

In summary, the main contributions of our work are as follows:

1. We develop an abstract model of Rust async runtimes, via our study of real-world implementations. Using our model, we observe that eventual progres-

sion for async runtimes fundamentally relies on the implementations of two kinds of critical components: *pollables* and *reactors*, which are programmed in a highly-stylized way. We formalize a core language $\mathcal{L}_{\mathsf{ART}}$ capturing this programming paradigm for modeling pollable and reactor implementations.
2. We formally define the key responsiveness property of interest, *eventual progression*, in terms of an operational semantics for $\mathcal{L}_{\mathsf{ART}}$ that uses a novel notion of *obligations*. We develop modular static analysis techniques that tracks how each obligation is eventually discharged, and show their soundness and that this implies eventual progression.
3. We implement a prototype tool Jackdaw in Rust, which allows annotating Rust pollable and reactor implementations to obtain models in $\mathcal{L}_{\mathsf{ART}}$, and analyzes these models. We evaluate Jackdaw on two popular Rust async runtimes and successfully verify eventual progression of various pollable/reactor implementations with low specification effort. We also show that Jackdaw rejects responsiveness bugs that can be arise in this challenging code.

2 Background

Async programming, as a concurrency programming model, provides a user-level abstraction (called *async tasks*) for writing concurrent programs. Compared to directly using OS threads, async tasks are entirely managed by a runtime system in user space; this lets programs run a larger number of computations concurrently on a small number of OS threads (even a single thread).

2.1 Async Rust

Rust provides `async`/`await` syntax for writing async programs in a style similar to sequential ones. One can define an *async function* by adding the `async` keyword before a normal function definition, such as the identity function below:

```
async fn id<T>(x: T) -> T { return x; }
```

When the async function `id` is called, instead of returning a value of declared type `T`, a *future* of type `Future<T>` is returned. This future represents an *async computation* that, once run, should eventually resolve to a value of type `T`.

Within async functions, one can attempt to resolve async computations by *awaiting* on them. The `await` expression in the example below executes the `id` function and immediately resolves it to the string literal.

```
async fn hello_world() {
    let s = id("hello␣world").await;
    println!(s);
}
```

Await-points define where an async computation might need to be paused. This does not happen in our `hello_world` example for two reasons:

1. The function only awaits async computations, which then run immediately.
2. There are no other computations that can be run concurrently.

These missing pieces are provided by combining `async`/`await` with async runtime-provided *pollables*: a collection of primitive async computations such as async I/O handlers, timers, and channels. Their implementations are provided by the runtime directly (and not as user-level async functions); each call to a pollable will return directly rather than be decomposed into smaller sub-computations.

The common usage of `await` reflects the design that all runtime-provided pollables implement the same `Future` trait; this syntax is compiled into calls to the `poll` method declared as follows:

```
pub trait Future<T> { fn poll(&mut self, w: Waker) -> Poll<T>; }
```

The `poll` method starts or resumes an async computation, executes until the computation yields, and returns the state of the async computation: `Pending` if the computation yields, and `Ready` if the computation completes[1]:

```
pub enum Poll<T> { Ready(T), Pending }
```

One might assume that user-defined async computations are expected to busy-poll their dependent pollables. Instead, an async computation that awaits another which is pending will itself be considered pending, and the *runtime* is given the responsibility to track pending tasks and resume them when appropriate.

The Rust compiler converts each async function into a state-machine: each await-point in an async function, along with its start and end, makes up the states. The state machine is implemented as the (generated) `poll` method for the `Future` object returned from the async function call. Correspondingly, all `await` expressions are translated to calls on the `poll` method of the awaited-on task. When such calls return with a `Ready` result, the state machine for the `await`ing task will advance; otherwise, a `Pending` result will be propagated upwards. This enables programmers to compose larger async programs from smaller ones, starting from pollables provided by the chosen runtime.

To enable concurrency when a user task is paused, a runtime must have alternative tasks to schedule. Runtime implementations provide an interface, typically named `spawn`, for registering potentially-concurrent tasks:

```
async fn concurrent_prints() {
    let t1 = spawn(print_after_3s());
    let t2 = spawn(print_after_3s());
    t1.await;
    t2.await;
}
```

[1] The method name `poll` may be somewhat misleading since `poll` does not simply query the state of async computations but also executes them as far as possible.

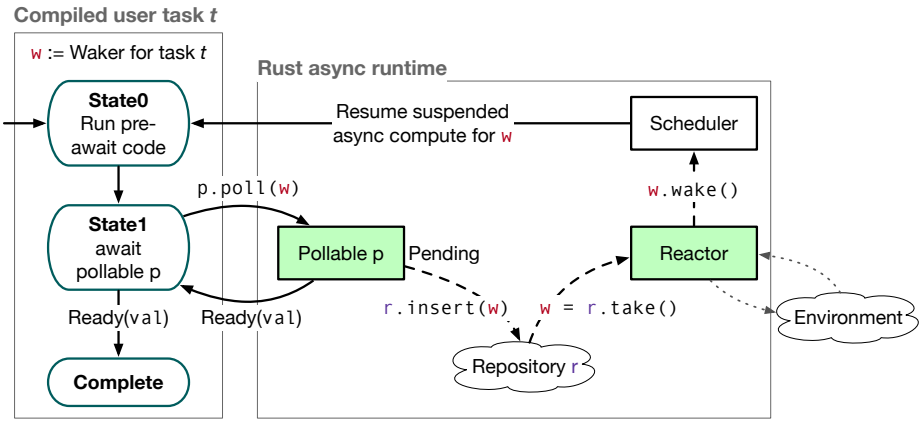

Fig. 1: Control flow between Pollable/Reactor/Waker/Scheduler. Each Pollable returns `Ready` or `Pending`; the Reactor reacts to the environment. Repositories are a concept we introduce in this paper to abstract over how wakers are passed to and stored by reactors.

When task `t1` starts waiting for its timer, the runtime can concurrently execute task `t2`. It is the runtime's responsibility to guarantee that, sometime after the waited-for time expires, the `poll` method of the timer will be called again, *and as a result*, the corresponding user task will resume executing.

In summary, async programming in Rust relies on the built-in `async`/`await` support of the language, but more importantly on an async runtime providing correctly implemented interfaces such as `Timer` and `spawn`. Since Rust does not ship with a dedicated async runtime, different async programs can use different runtime implementations. It is therefore important that different async runtimes live up to the same set of specifications.

2.2 Rust Async Runtimes

The progression of user-defined async tasks relies on an async runtime (Figure 1) whenever those tasks are paused. A critical responsibility of the runtime is that any paused task must eventually have its `poll` method called, attempting to resume its execution.

Wakers. Rust async runtimes encapsulate a pending-resumption task in a `Waker` type. Each `Waker` is a handle on the top-level user task whose poll chain currently (transitively) awaits a pollable. Calling `wake` on it notifies a *scheduler* to re-invoke that top-level task's `poll`. The compiler-generated state machine records the await-point at which the task was suspended, so the re-entered `poll` resumes there and re-invokes `poll` on the same awaited pollable that previously returned `Pending`. The only possibility for these tasks to be paused is if a runtime pollable returns `Pending`. It is the responsibility of the runtime implementation

to register the corresponding `Waker` with the runtime, along with information to allow the runtime to determine what event the task is waiting for.

Reactors. A dedicated runtime component (typically called a *Reactor*) is used to manage all currently-paused tasks, storing references to their corresponding `Waker` instances. The Reactor is run continuously: its responsibility is to periodically check the *conditions* that each paused task is waiting on (e.g., whether timers have expired, or I/O events occurred). Rather than directly calling `poll` on these tasks, which would interrupt the Reactor's execution, the Reactor calls `wake` to register the task with a scheduler. Assuming a fair scheduling algorithm, the scheduler guarantees that each of these tasks will ultimately have their `poll` function called again.

In this paper, we leverage the abstractions provided by this idiomatic design (which is shared across all Rust async runtimes we are aware of), and assume in particular a high-level specification for wakers (and their scheduling) without verifying their implementations or the fairness of the scheduling algorithm. Instead, we focus on (1) the implementations of pollables, which form the key building blocks and interfaces for building user-level programs, and (2) the implementations of Reactors, which are responsible for ensuring that these pollables, and therefore the dependent user tasks, get called again as needed.

3 Overview

We introduce our approach to verifying eventual progression of Rust async runtimes through a representative example: a `Timer` pollable paired with a Reactor that manages a collection of timers. We then state the eventual progression property and the trusted components on which it relies, and informally describe the per-component conditions our verification technique enforces. The model and the analyses are made formal in Sections 4 and 5.

3.1 The Timer Example

A `Timer` is a runtime-provided pollable that pauses the calling async task until a specified time has elapsed. Figure 2 shows the relevant data definitions and the `poll` method of `Timer`. A timer carries the target time `instant` and a reference to the Reactor responsible for waking it up. The `poll` method first compares `instant` against the current time `now()`. If the time has already passed, `poll` returns `Ready` and the awaiting task proceeds. Otherwise, `poll` sends the waker and `instant` to the Reactor via `transfer`, and returns `Pending`. The awaiting task is paused, and the Reactor takes over the responsibility to call `poll` again when the target time has passed.

A typical Reactor that manages a collection of timers is shown in Figure 3. The Reactor maintains all currently-paused timers in the `timers` vector; `transfer` (elided) inserts a fresh `TimerData` into this vector under appropriate synchronization. The `react` method runs an infinite loop: at each iteration, it partitions the vector based on whether each timer's target time has passed, wakes every timer

```
struct TimerData { instant: Instant, waker: Waker }
struct Timer { instant: Instant, reactor: &Reactor }

// w is a Waker for this pollable (guaranteed by Rust compiler)
fn poll(self, w: Waker) -> Poll<()> {
    if self.instant <= now() {
        return Poll::Ready(());
    } else {
        self.reactor.transfer(
            TimerData { instant: self.instant, waker: w });
        return Poll::Pending;
    }
}
```

Fig. 2: A `Timer` pollable. `transfer` inserts a `TimerData` (the waker together with the target time) into the Reactor's repository. We elide Rust details concerning references and the `Future` trait.

whose time has passed, and retains the others for the next iteration. The Reactor cannot terminate, because new responsibilities to wake timers may be transferred to it at any future point.

This pair illustrates the three reasoning concepts our verification builds on:

Monotonic completion conditions. A timer's completion condition `self.instant <= now()` is a predicate on the external environment that is *monotonic*: once true, it remains true. Monotonicity is what allows `poll` (and the Reactor) to reliably decide completion based on an observation of the condition. Pollables other than timers (e.g., I/O handlers, channels) condition on different monotonic predicates, but the principle is the same.

Responsibility transfer between components. Initially the called pollable is responsible for its own completion: if its completion condition holds when `poll` is called, `poll` returns `Ready` directly. If not, the pollable hands the responsibility to its Reactor via `transfer`, then returns `Pending`. The Reactor periodically checks the completion condition and, once it holds, calls `wake` on the corresponding waker; the wakened task is then re-scheduled, and the responsibility is finally discharged by a `poll` call returning `Ready`.

3.2 Eventual Progression: the Core Responsiveness Property

Informally, the property we want of an async runtime is: *whenever a user task is paused waiting on a pollable whose completion condition will eventually hold, that task is eventually resumed.* We refer to this property as *eventual progression*. We make this precise in Section 5.4.

Several components participate in delivering eventual progression: the user task itself, the Rust compiler, the external environment, the scheduler, and the

```
struct Reactor { timers: Vec<TimerData> }

fn react(&self) {
    loop {
        let (ready, pending) = self.timers.partition(
            |t: TimerData| t.instant < now()
        );
        for t in ready { t.waker.wake(); }
        self.timers.append(pending);
    }
}
```

Fig. 3: A Reactor that manages a collection of timers. The `react` method is spawned as a background thread.

runtime-provided pollables and Reactors. A bug anywhere along this chain can violate the property; we cannot ask any single component to bear the entire burden. Our verification focuses on the pollable and Reactor implementations, treating the other components as *trusted*.

Trusted Component 1 (The Rust Compiler). *The Rust compiler preserves the semantics of async functions during the compilation of these functions to their state machine representations (Section 2.1).*

Trusted Component 2 (Observations of the External Environment). *All APIs for polling and extracting data from the environment external to the Rust program (OS, devices, etc.) behave correctly: signaling events and providing data when expected.*

Trusted Component 3 (Fair Scheduling of Rust Async Tasks). *The scheduler components of the Rust async runtime schedule tasks passed to them fairly.*

The first two trusted components reflect well-established compilation and OS infrastructure; the third reflects a well-studied problem largely independent of the async focus of our work. Under these assumptions, the responsibility for eventual progression reduces to the interaction between pollables and Reactors: it is their collaboration that translates "the completion condition holds" into "the awaiting task's `poll` is called and returns `Ready`."

3.3 Per-Component Conditions

We decompose eventual progression into two component-level conditions, one for pollables and one for Reactors. Both are stated here informally; we make them precise as verification conditions in Section 5.

A pollable's responsibility is captured by three conditions on its `poll` method:

(P1) *Completion faithfulness.* If `poll` returns `Ready`, the completion condition has held.
(P2) *Completion convergence.* If the completion condition holds when `poll` is called, `poll` returns `Ready`.
(P3) *Responsibility transfer.* If `poll` returns `Pending`, the pollable has transferred its waker to a Reactor along with its completion condition.

A Reactor's responsibility is captured by two iteration-level conditions on the body of its main loop:

(R1) *Iteration progress.* If a transferred waker's completion condition holds at the start of an iteration, the Reactor calls `wake` on it within that iteration.
(R2) *Iteration preservation.* A transferred waker that is not woken in this iteration is retained in the Reactor's repository for a future iteration.

We illustrate these conditions using the `Timer`/Reactor pair from Section 3.1. The completion condition is `self.instant <= now()`. `Timer`'s `poll` returns `Ready` only on the path where `self.instant <= now()` is observed, so (P1) holds. If `self.instant <= now()` holds when `poll` is called, the `if`-test takes the `Ready` branch, so (P2) holds. The `Pending` path is preceded by `transfer`, which inserts `TimerData` into the Reactor's repository; this is the responsibility transfer (P3). For the Reactor: `partition` separates timers whose target time has passed from those that have not; `wake` is then called on every timer in the first group, satisfying (R1); `append` retains the rest, satisfying (R2).

Crucially, (P1)–(P3) and (R1)–(R2) can be checked per-component, in isolation. This modularity is provided by our task-centric view: even though a Reactor concretely manages a collection of timers, the per-iteration conditions can be discharged by reasoning about a single arbitrary timer. Sections 4 and 5 formalize this insight by defining a core language $\mathcal{L}_{\mathsf{ART}}$ for pollable and Reactor implementations, an abstract domain that tracks observations of monotonic completion conditions, and verification conditions that imply (P1)–(P3) and (R1)–(R2).

4 A Model of Rust Async Runtimes

We model Rust async runtime components in a core language $\mathcal{L}_{\mathsf{ART}}$ (Language for Async Runtime). $\mathcal{L}_{\mathsf{ART}}$ is designed to capture the high-level *architecture* of Rust async runtimes and the programming paradigm for pollable and reactor reasoning rather than modeling the programming language Rust.

Importantly, $\mathcal{L}_{\mathsf{ART}}$ builds in two primitive notions that are central to verifying eventual progression: *monotonic predicates* (typically to model completion predicates for pollables) and *obligations* (used to abstractly represent suspended user tasks waiting on a runtime-provided pollable).

To reason about a collection of obligations in a reactor, we build in the notion of a *repository* as an abstraction over concrete ways a reactor may store wakers. A repository abstracts over a concrete data type such as a Rust `Vec`,

and $\mathcal{L}_{\mathsf{ART}}$ statements abstract over concrete functions how wakers are passed to (e.g., `transfer` in Fig. 2), stored in, and removed from these data types.

$\mathcal{L}_{\mathsf{ART}}$ is intentionally minimal, but as we will show in Sections 5 and 6, $\mathcal{L}_{\mathsf{ART}}$ is sufficient to capture the design idioms shared by real Rust async runtime components with few extensions.

4.1 The Language $\mathcal{L}_{\mathsf{ART}}$

$\mathcal{L}_{\mathsf{ART}}$ is parametrized over an infinite set of *monotonic predicate symbols* $\mathcal{P}$ (hereafter, simply *predicates*), partitioned into two disjoint (infinite) sets of *internal* and *external* predicates: $\mathcal{P} = \mathcal{P}_{int} \uplus \mathcal{P}_{ext}$. A monotonic predicate $p \in \mathcal{P}$ represents a *stable-once-established* condition: once p holds, it holds permanently in the future. *Internal* predicates represent conditions that can be satisfied by program actions (e.g., setting a flag from `false` to `true`); *external* predicates model conditions which can only be updated by the environment, e.g., external signals such as a timer expiring or an I/O event completing.

In real code, the truth of monotonic predicates can typically be tested via suitable boolean Rust expressions whose value changes monotonically; for example, a timer's completion condition can be checked via `self.instant <= now()`. We use the predicate symbols in our language to abstract over these concrete conditions, which are required only to satisfy monotonicity.

We assume an infinite set of wakers, WAKER, and a lookup function τ : WAKER $\rightharpoonup$ STMT $\times$ *Env*; for a waker w, $\tau(w)$ denote the (compiled, suspended) user task the waker is able to reschedule (Sec. 2.2). As will be explained in Sec. 4.2, such suspended tasks are represented concretely by pairs of *a pollable implementation* that the user task is currently waiting on and its local environment; in the real Rust code this is handled with compiler-generated closures.

Figure 4 defines the syntax of $\mathcal{L}_{\mathsf{ART}}$. Values include the unit, the pollable return values `Ready`(v) and `Pending`, and wakers. Statement `trigger`(p) sets the value of an internal predicate p to true (it is a no-op if p was already true). `insert`(r, w, p) sends the waker w to the public repository r together with the completion predicate p that must be waited for. Conceptually, this transfers the obligation for *progress* of $\tau(w)$ to the reactor r (which must now monitor p and wake w when satisfied). `take`(r, w) removes the entry for w from r (e.g., because it will be woken up). `wake`(w) invokes waker w, signaling the scheduler to re-schedule the user task $\tau(w)$, which will in turn re-poll *the corresponding pollable*. `for` (w, p) `in` r $\{\, s \,\}$ iterates the body s over every (w, p) pair in r. `loop` s repeatedly runs s (indefinitely). `return` v terminates a pollable with result v (typically `Ready`(()) or `Pending`).

We impose *well-formedness conditions* (formally defined in Appendix A) on the $\mathcal{L}_{\mathsf{ART}}$ statements representing pollable and reactor implementations. A pollable π is a statement that satisfies its well-formedness predicate *wfp* when π contains no `loop` or reactor-only constructs, and every execution path through π contains a `return` statement.

Reactors $\mathcal{R}$ are first-class in $\mathcal{L}_{\mathsf{ART}}$, and each owns (i.e., no overlap with other reactors) a non-empty list of *public repositories*. Each public repository r is de-

$$
\begin{array}{rl}
\text{Values} & v ::= () \mid \texttt{Ready}(v) \mid \texttt{Pending} \mid w \quad (w \in \textsc{Waker}) \\
\text{Statements} & s ::= s_1 \,;\, s_2 \mid \texttt{skip} \mid \texttt{return}\, v \mid \texttt{wake}(w) \\
& \quad \mid \texttt{trigger}(p) \;\; (p \in \mathcal{P}_{int}) \\
& \quad \mid \texttt{if}\, p\, \texttt{then}\, s_1\, \texttt{else}\, s_2 \quad (p \in \mathcal{P}) \\
& \quad \mid \texttt{insert}(r, w, p) \;\; (r \in \textsc{Repo},\; p \in \mathcal{P}) \\
& \quad \mid \texttt{take}(r, w) \;\; (r \in \textsc{Repo}) \mid \texttt{for}\, (w, p)\, \texttt{in}\, r\, \{\, s \,\} \\
& \quad \mid \texttt{loop}\, s \\
\text{Pollables} & \pi ::= \lambda(w, p).\, s \quad (\mathit{wfp}(s)) \\
\text{Reactors} & \mathcal{R} ::= \texttt{reactor}\, \{\, \texttt{shared}\, \overrightarrow{r_s : (T_{r_s}, cp_{r_s})}\, \texttt{in}\, (\texttt{loop}\, s) \,\} \quad (\mathit{wfr}(s))
\end{array}
$$

Fig. 4: Syntax of $\mathcal{L}_{\mathsf{ART}}$, parametrized over monotonic predicates $\mathcal{P} = \mathcal{P}_{int} \uplus \mathcal{P}_{ext}$. A pollable is a statement that returns on every path; a reactor declares a set of public repositories and wraps its body in an infinite loop. We highlight statements that are reactor-only.

clared along with a pair (T_r, cp_r), in which T_r is the type of pollable whose wakers may be stored in r and $cp_r : T_r \to \mathcal{P}$ is the *completion predicate scheme* mapping each pollable instance to its completion predicate. A reactor body needs to satisfy its well-formedness predicate *wfr*, which requires that the body never `return`s and contains no `loop` other than the (required) outermost one; this reflects the fact that reactors are intended to monitor the environment indefinitely and never return.

For simplicity, we assume each reactor has at most one public repository per type of pollable and that well-formed programs respect the repository type-checking constraints. We also assume the convention that `insert`, `take`, and `for` in a reactor body access only repositories declared by that reactor, and pollables always communicate with a dedicated, running reactor.

In real code, rather than being associated with completely independent predicates, two pollables of the same type will always depend on *similar* conditions, but may differ in instance-specific parameters: for example, two timers both wait on $t \leq$ `now()`, but with different deadlines t. In our language, we simply represent these as different predicates under the same scheme, as we abstract over the form of these concrete boolean expressions.

4.2 Operational Semantics

We define a small-step operational semantics for $\mathcal{L}_{\mathsf{ART}}$; aside from the language itself the main design novelty is that our runtime states explicitly track the known status of monotonic predicates, as well the status of wakers for suspended user tasks. We begin by defining rules for statement execution (only), and then

show how to lift these to accommodate rules that model interleavings with e.g., events from the external environment.

A runtime state is a pair $(\rho, G) \in \textsc{Config} \triangleq \mathit{Env} \times \textsc{Global}$, separating local states from shared global states:

- $\rho \in \mathit{Env}$ is the *local environment* for pollables or reactors, mapping local variables to values;
- $G = (\Sigma, \mathcal{Q}, R) \in \textsc{Global} \triangleq \textsc{Hist} \times \wp(\textsc{Waker}) \times \textsc{Repos}$ is the *global state*, where
  - $\Sigma \in \textsc{Hist} \triangleq \wp(\mathcal{P})$ is the *monotonic history*, recording the set of monotonic predicates true so far;
  - $\mathcal{Q} \in \wp(\textsc{Waker})$ is the *rescheduling queue*, containing the wakers that have been woken up but waiting for the scheduler to reinvoke the `poll` method of the corresponding pollable;
  - $R \in \textsc{Repos} \triangleq \textsc{Repo} \to \wp(\textsc{Waker} \times \mathcal{P})$ is the *repository map*, mapping each repository to the set of waker-predicate paired stored in it. We assume all waker-predicate pairs in a repository r respect the repository typing, i.e., every $(w, p) \in R(r)$ has $p \in \mathit{img}(cp_r)$, where img denotes the image of the predicate scheme cp_r.

Figure 5 defines the operational semantics for $\mathcal{L}_{\mathsf{ART}}$. `trigger`, `insert`, `take`, and `wake` are atomic effectful statements that update the global state and reduce to `skip`. `if` statements branch on whether a predicate is in the monotonic history Σ[2]. `for` statements iterate over every (w, p) in $R(r)$ present *at the start of the for-loop*. We omit other conventional rules for `loop`, `skip`, etc. The full definition is in App. B.

As explained in Sec. 3, a Rust async runtime in general has many pollables and reactors currently executing, and additionally interacts with other trusted components, user tasks and the external environment, none of which are directly modeled by $\mathcal{L}_{\mathsf{ART}}$ programs. We model them via additional non-deterministic transitions interleaved with the execution of statements defined in Fig. 5.

We define *runtime configurations* as a pair $(\mathcal{T}, (\Sigma, \mathcal{Q}, R))$ where $\mathcal{T}$ is a multiset of pairs $\textsc{Stmt} \times \mathit{Env}$ representing all currently running pollables and reactors. The runtime-level judgment $\mathcal{T}, (\Sigma, \mathcal{Q}, R) \longmapsto_r \mathcal{T}', (\Sigma', \mathcal{Q}', R')$ is the reflexive transitive closure of the four rules at the bottom of Fig. 5. Rule (exec) models normal execution of a single-step of a currently-running pollable (or reactor). Rule (user-new-await) models a new pollable instance being started due to the compilation of an `await` usage in a user-task. We abstract over the details of initializing a new instance of some pollable π (e.g., initializing the environment) using the init_π predicate on the new pair of its implementation s and environment ρ. For simplicity, we do not model dynamic creation of reactors but assume these to be present in an initial runtime configuration.

Rule (env-trigger) adds an external predicate to the history at arbitrary time; this models that the environment may make an external condition true

[2] $\mathcal{L}_{\mathsf{ART}}$ only allows branching on single monotonic predicates; extending this to combinations as well as more-standard program expressions would be straightforward.

$\boxed{s \mid \kappa \longrightarrow_p s' \mid \kappa'}$

$$\frac{p \in \mathcal{P}_{int}}{\texttt{trigger}(p) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma \cup \{p\}, \mathcal{Q}, R))} \text{ (trigger)}$$

$$\frac{}{\texttt{wake}(w) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q} \cup \{w\}, R))} \text{ (wake)}$$

$$\frac{R' = R[r \mapsto R(r) \cup \{(w,p)\}]}{\texttt{insert}(r,w,p) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q}, R'))} \text{ (insert)}$$

$$\frac{\begin{array}{c}(w,p) \in R(r) \\ R' = R[r \mapsto R(r) \setminus \{(w,p)\}]\end{array}}{\texttt{take}(r,w) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q}, R'))} \text{ (take)}$$

$$\frac{p \in \Sigma}{\texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2 \mid \kappa \longrightarrow_p s_1 \mid \kappa} \text{ (if-t)}$$

$$\frac{p \notin \Sigma}{\texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2 \mid \kappa \longrightarrow_p s_2 \mid \kappa} \text{ (if-f)}$$

$$\frac{\begin{array}{c}R(r) = \{(w_1,p_1), \ldots, (w_k,p_k)\} \\ s_i \triangleq s[w_i, p_i / w, p] \quad k \geq 1\end{array}}{\texttt{for } (w,p) \texttt{ in } r \{ s \} \mid (\rho, G) \longrightarrow_p s_1 ; \cdots ; s_k \mid (\rho, G)} \text{ (for)}$$

$$\frac{R(r) = \varnothing}{\texttt{for } (w,p) \texttt{ in } r \{ s \} \mid (\rho, G) \longrightarrow_p \texttt{skip} \mid (\rho, G)} \text{ (for-emp)}$$

$\boxed{\mathcal{T}, G \longmapsto_r \mathcal{T}', G'}$

$$\frac{s \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p s' \mid (\rho', (\Sigma', \mathcal{Q}', R'))}{\mathcal{T} \cup \{(s,\rho)\}, (\Sigma, \mathcal{Q}, R) \longmapsto_r \mathcal{T} \cup \{(s',\rho')\}, (\Sigma', \mathcal{Q}', R')} \text{ (exec)}$$

$$\frac{init_\pi(s,\rho)}{\mathcal{T}, (\Sigma, \mathcal{Q}, R) \longmapsto_r \mathcal{T} \cup \{(s,\rho)\}, (\Sigma, \mathcal{Q}, R)} \text{ (user-new-await)}$$

$$\frac{p \in \mathcal{P}_{ext} \quad p \notin \Sigma}{\mathcal{T}, (\Sigma, \mathcal{Q}, R) \longmapsto_r \mathcal{T}, (\Sigma \cup \{p\},\ \mathcal{Q},\ R)} \text{ (env-trigger)}$$

$$\frac{w \in \mathcal{Q} \quad \tau(w) = (s,\rho)}{\mathcal{T}, (\Sigma, \mathcal{Q}, R) \longmapsto_r \mathcal{T} \cup \{(s,\rho)\},\ (\Sigma, \mathcal{Q} \setminus \{w\}, R)} \text{ (sched)}$$

Fig. 5: Operational semantics for $\mathcal{L}_{\mathsf{ART}}$: statement small-step rules (top) and runtime configuration rules (bottom). Standard rules such as sequencing or `skip` are omitted; the full definition is in Appendix B.

at arbitrary time. Rule (sched) schedules a waker w in the rescheduling queue $\mathcal{Q}$ by reinvoking the corresponding pollable. Since our model abstracts over the user task code, which (due to the Rust compiler's transformations) is guarantee to re-call the awaited runtime pollable, our τ function formally maps the waker *directly* to this runtime pollable's representation, which is made available for rescheduling by our rules. In the real runtime system, it is the user task that originally awaited the pollable that would be rescheduled by the waker.

Obligations. A key intuition for what is needed for an async runtime to be responsive is that the runtime *never forgets* async tasks that may be able to make further progress (for example, it doesn't forget a task waiting on a timer). To formalize this idea, we introduce the concept of *obligations*: responsibilities that runtime components (especially reactors) are conceptually assigned that must not be forgotten until they are discharged.

Specifically, we use pairs $(w, p) \in \textsc{Waker} \times \mathcal{P}$ in our novel operational semantics to represent the obligation: *if the monotonic condition p ever holds, the runtime must eventually resume the task associated with w.* An obligation is created every time a user task awaits on a pollable, represented by the waker used for resumption of the task and the condition being waited on. If the pollable waited on returns `Ready`, the obligation is discharged as the user code will be returned to directly; on the other hand, if a reactor executes $\texttt{wake}(w)$ this discharges the obligation based on our assumptions on the fair scheduler. These are the only ways these obligations can be discharged.

Given a repository map, R, the *obligations of a reactor* $\mathcal{R}$, written $Ob(\mathcal{R})$ are defined as the union of the obligations in $R(r)$ for every repository r owned by $\mathcal{R}$. For a given global state G we overload this syntax to define the *total obligations* of a global state $Ob(G)$ to be the union of $Ob(\mathcal{R})$ in terms for all reactors $\mathcal{R}$. Note that $Ob(\mathcal{R})$ grows when a pollable transfers a waker via `insert` to $\mathcal{R}$ and shrinks one is removed via `take`.

4.3 Eventual Progression and Component Local Conditions

We now state an overall guarantee in terms of the interaction between user code and an async runtime; we use this high-level notion to derive decomposed verification conditions specific to pollable and reactor implementations.

Definition 1 (Eventual Progression). *An async runtime satisfies Eventual Progression iff, for any runtime configuration $(\mathcal{T}, G)$ reached during its execution, and for all user tasks awaiting on a pollable s with predicate p, either (a) s is running with $(s, \rho) \in \mathcal{T}$ or (b) for which a waker w exists with $(w, p) \in Ob(G)$, it is guaranteed that if, on further execution p eventually becomes true, then eventually the user task will be resumed.*

Case (a) corresponds to a user task awaiting a pollable which is already running; this can potentially either result in delegation to a reactor (and case (b)) or returning a $\texttt{Ready}(v)$ value directly causing the user task to resume. This allows us to define conditions specific to the implementation of a single pollable:

Definition 2 (Pollable Responsiveness). *A pollable π is* responsive *if every invocation with obligation (w, p) satisfies:*

(P1) if the invocation returns `Ready`(v)*, then $p \in \Sigma$ at the return point;*
(P2) if $p \in \Sigma$ at the entry of the invocation, then the invocation returns `Ready`(v) *for some v.*
(P3) if the invocation returns `Pending`*, then either $(w, p) \in R(r)$ for some public repository r, or $w \in \mathcal{Q}$ at the return point;*

Definition 2 captures two key concerns. (P1) and (P2) form the soundness and completeness for completion: (P1) ensures `Ready`-returns reflect actual completion; (P2) ensures that an already-completed computation is eventually reported. (P3) ensures obligation-preservation: when a pollable yields, the obligation it carries must not be forgotten by the runtime. It may be transferred to a public repository (via `insert`) or discharged immediately by waking w (via `wake`).

Considering Case (b) in Def. 1, we must require enough of a reactor to also ensure that obligations are preserved and eventually discharged when ready. We express these conditions per iteration i of the `loop` s construct defining the reactor, and write G^i_{start}, G^i_{end} for the global states at iteration start and end. In the following, Ob^i_{con} denotes the obligations inserted into public repositories *during* iteration i (e.g., by concurrently-running pollables). We write $Wake_i \triangleq \{\, w \mid \texttt{wake}(w) \text{ fires during iteration } i \,\}$ for the set of all obligations discharged via `wake` during the iteration:

Definition 3 (Reactor Responsiveness). *A reactor* `loop` s *is* responsive *if every iteration i satisfies:*

$$\textit{(R1)} \quad \forall (w,p) \in Ob(G^i_{start}).\ p \in \Sigma^i_{start} \ \Rightarrow\ w \in Wake_i$$
$$\textit{(R2)} \quad Ob(G^i_{end}) \ \supseteq\ \big(Ob(G^i_{start}) \cup Ob^i_{con}\big) \setminus \big\{(w,p) \mid w \in Wake_i\big\}$$

(R1) guarantees progress at each iteration; the obligations whose predicate already holds must be woken in the *same* iteration. In principle this condition could be weakened, but in practice this strong guarantee is common. (R2) guarantees that no obligation is lost; obligations not discharged during the iteration must be retained in the public repositories by the end of the iteration. Obligations arriving mid-iteration via Ob^i_{con} are not required to be woken in the same iteration even if their predicate already holds by iteration end; (R2) carries them into the next iteration.

The relationship between these component local conditions and the global Eventual Progression property is summarized as follows; the corresponding soundness theorem and its proof are in Section 5.4.

Theorem 1 (Componentwise Decomposition). *Assume the three trusted components of Section 3.2. If every pollable in the runtime preserves obligations when yielding (Definition 2, (P3)) and every reactor is responsive (Definition 3), then the runtime satisfies Eventual Progression (Definition 1).*

The focus of Section 5 is to define new and efficient *modular* static analyses whose success implies these required properties. We design abstract domains and tailored verification conditions VC_π (for pollables) and $VC_\mathcal{R}$ (for reactors) that, when discharged on the syntactic body of a component, imply that the component is responsive.

5 Static Analysis of Eventual Progression

This section defines novel static analyses presented in an abstract-interpretation style, that verify the componentwise properties defined and motivated in the previous section. Our abstract domains are novel in that they focus on the notions of monotonic predicates and undischarged obligations. Section 5.1 introduces a novel unified abstract domain shared by both analyses; Sections 5.2 and 5.3 define analysis-refined pollable and reactor verification conditions; Section 5.4 states the soundness theorem and lifts the componentwise guarantees to a user-facing runtime-responsiveness theorem.

5.1 Abstract Domain

We design an abstract domain which tracks *two* kinds of information at each program point: observations on monotonic predicates, and *symbolic obligations* abstracting over sets of concrete obligations held by reactors (their repositories).

We define abstract states $\sigma = (S, \widehat{R}, \widehat{L}) \in \textsc{State}$, where S is a map tracking *predicate observations*, $\widehat{R}$ is a map symbolically tracking obligations stored in public repositories, and $\widehat{L}$ a set tracking locally-held symbolic obligations. We explain each component formally below.

States are ordered pointwise. The join $\sqcup$ takes the pointwise lub of S, pointwise union of $\widehat{R}$, and union of $\widehat{L}$. The bottom element $\bot_{\textsc{State}}$ represents an *unreachable* program state. $\bot_{\textsc{State}} \sqcup \sigma = \sigma$ for any abstract state.

Predicate observations. We use the set $\textsc{Obs} = \{\bot, \mathsf{tt}, \mathsf{ff}, \top\}$ for *predicate observations*. tt means *p has been observed* to hold; ff means p was observed not to hold at an earlier moment; $\top$ is unknown; $\bot$ means unreachable.

Two operators are defined over $\textsc{Obs}$: the refinement operator $\triangleright$ updates an observation along a single execution path (defined below); the join $\sqcup$ merges observations from multiple paths, taken as the least upper bound on the ordering $\bot \sqsubseteq \mathsf{tt}, \mathsf{ff} \sqsubseteq \top$ (where tt and ff are incomparable and $\mathsf{tt} \sqcup \mathsf{ff} = \top$).

The monotonic refinement operator $\triangleright : \textsc{Obs} \times \{\mathsf{tt}, \mathsf{ff}\} \to \textsc{Obs}$ updates a prior observation by a fresh observation result (e.g., a branch of an if-statement) along a single path, encoding the monotonicity of predicates:

$$\begin{array}{lll} \mathsf{tt} \triangleright \mathsf{tt} = \mathsf{tt} & \mathsf{ff} \triangleright \mathsf{tt} = \mathsf{tt} & \top \triangleright v = v \\ \mathsf{tt} \triangleright \mathsf{ff} = \bot & \mathsf{ff} \triangleright \mathsf{ff} = \mathsf{ff} & \bot \triangleright v = \bot \end{array}$$

If the value of a predicate is unknown $\top$, it can be refined into a concrete observation (either tt or ff). A ff observation can be refined to tt, while refining tt into ff violates monotonicity, indicating the current path is unreachable.

$\boxed{\mathcal{T}[s] : \textsc{State} \to \textsc{State}}$

$$
\begin{aligned}
\mathcal{T}[\texttt{skip}](\sigma) &= \sigma \\
\mathcal{T}[\texttt{trigger}(p)](\sigma) &= \sigma\big[\, S := S[p \mapsto S(p) \triangleright \mathsf{tt}] \,\big] \\
\mathcal{T}[\texttt{insert}(r, \hat{w}, \hat{p})](\sigma) &= \sigma\big[\, \widehat{R}(r) := \widehat{R}(r) \cup \{(\hat{w}, \hat{p}, \tau)\},\ \ \widehat{L} := \widehat{L} \setminus \{(\hat{w}, _, _)\} \,\big] \\
&\qquad \text{if } (\hat{w}, \hat{p}, \tau) \in \widehat{L} \ \wedge\ \hat{p} \in \mathit{img}(cp_r);\ \text{otherwise } \bot_{\textsc{State}} \\
\mathcal{T}[\texttt{wake}(\hat{w})](\sigma) &= \sigma\big[\, \widehat{L} := \widehat{L} \setminus \{(\hat{w}, _, _)\} \,\big] \\
&\qquad \text{if } (\hat{w}, _, _) \in \widehat{L};\ \text{otherwise } \bot_{\textsc{State}} \\
\mathcal{T}[\texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2](\sigma) &= \sigma_{s_1} \sqcup \sigma_{s_2} \\
\text{where } \sigma_{s_1} &= \mathcal{T}[s_1]\big(\sigma\big[\, S := S[p \mapsto S(p) \triangleright \mathsf{tt}] \,\big]\big) \\
\sigma_{s_2} &= \mathcal{T}[s_2]\big(\sigma\big[\, S := S[p \mapsto S(p) \triangleright \mathsf{ff}] \,\big]\big) \\
\mathcal{T}[s_1 \,;\, s_2](\sigma) &= \mathcal{T}[s_2](\mathcal{T}[s_1](\sigma)) \\
\mathcal{T}[\texttt{return } v](\sigma) &= \bot_{\textsc{State}}
\end{aligned}
$$

Fig. 6: Transfer functions for the shared statements of $\mathcal{L}_{\mathsf{ART}}$. All transfer functions preserve $\bot_{\textsc{State}}$, i.e., $\mathcal{T}[s](\bot_{\textsc{State}}) = \bot_{\textsc{State}}$ for every s.

$$
\begin{aligned}
\mathcal{T}[\texttt{take}(r, \hat{w})](\sigma) &= \sigma\big[\, \widehat{R}(r) := \widehat{R}(r) \setminus \{(\hat{w}, \hat{p}, \tau)\},\ \ \widehat{L} := \widehat{L} \cup \{(\hat{w}, \hat{p}, \tau)\} \,\big] \\
&\qquad \text{if } (\hat{w}, \hat{p}, \tau) \in \widehat{R}(r);\ \text{otherwise } \bot_{\textsc{State}} \\
\mathcal{T}[\texttt{for } (w, p) \texttt{ in } r \,\{\, s \,\}](\sigma) &= \mathcal{T}[s_k](\cdots \mathcal{T}[s_1](\sigma)) \\
&\qquad \text{where } \widehat{R}(r) = \{(\hat{w}_1, \hat{p}_1, \tau_1), \ldots, (\hat{w}_k, \hat{p}_k, \tau_k)\} \\
&\qquad \text{and } s_i \triangleq s[\hat{w}_i, \hat{p}_i / w, p].
\end{aligned}
$$

Fig. 7: Transfer functions for reactor-only $\mathcal{L}_{\mathsf{ART}}$ statements.

$\bot$ cannot be refined to any other values. We adopt a smash-product convention for abstract states: if $S(p) = \bot$ for any p, the entire state σ is identified as the bottom $\bot_{\textsc{State}}$. Intuitively, this means any observation that violates monotonicity reveals an infeasible path; our analyses rely on this insight to prune infeasible paths.

Symbolic obligations. Let $\widehat{\textsc{Waker}}$ be a set of *symbolic wakers*. A *symbolic obligation* is a triple $\hat{o} = (\hat{w}, \hat{p}, \tau) \in \widehat{\textsc{Ob}} \triangleq \widehat{\textsc{Waker}} \times \mathcal{P} \times \{\mathsf{tt}, \top\}$. We use tag τ to distinguish different assumptions about predicate $\hat{p}$'s state at the start of analysis: tag tt indicates $\hat{p}$ is assumed to hold; $\top$ indicates $\hat{p}$'s state is unknown. $\widehat{R} : \textsc{Repo} \to \wp(\widehat{\textsc{Ob}})$ maps each public repository to the symbolic obligations currently stored in it; $\widehat{L} \in \wp(\widehat{\textsc{Ob}})$ collects the symbolic obligations held locally (taken out of a repository, or initialized at the start of the analysis).

Transfer functions. Figure 6 defines the transfer function $\mathcal{T}[s]$ for $\mathcal{L}_{\mathsf{ART}}$ statements that are shared between pollables and reactors. $\texttt{trigger}(p)$ refines

$S(p)$ to `tt`. `insert` moves an obligation from $\widehat{L}$ into $\widehat{R}(r)$, requiring the obligation to be held locally and matching the repository's typing. `wake` consumes a locally-held obligation. `if` p refines $S(p)$ on each branch via $\triangleright$ and joins the two results after the statement. `return` v yields $\bot_{\text{STATE}}$, as the execution is terminated.

`insert`, `take`, and `wake` yield $\bot_{\text{STATE}}$ when side-conditions fail; this models infeasible paths due to e.g., type mismatches or exceptions.

Figure 7 extends the transfer functions to reactor-only statements `take` and `for`. `take` moves an obligation out of a public repository and holds it locally; `for` (w, p) `in` r is unrolled for every symbolic obligation in $\widehat{R}(r)$. The w and p mentioned in the loop body are substituted with the $\hat{w}_i$ and $\hat{p}_i$ of each obligation.

5.2 Pollable Analysis Verification Conditions

We analyze each pollable implementation π *twice*, running our transfer function with two different initial states, defined as follows (in which $\hat{w}$ and $\hat{p}$ are symbolic waker and completion predicates):

- $\sigma^{\top}_{start}$: $S_{start}(\hat{p}) = \top$, $\widehat{L}_{start} = \{(\hat{w}, \hat{p}, \top)\}$, $\widehat{R}_{start}(r) = \varnothing$ for all r;
- $\sigma^{\mathtt{tt}}_{start}$: $S_{start}(\hat{p}) = \mathtt{tt}$, $\widehat{L}_{start} = \{(\hat{w}, \hat{p}, \mathtt{tt})\}$, otherwise identical.

For a well-formed pollable body, every path terminates with a `return`, so the abstract state immediately before each `return` summarizes what is known about Σ and the entry obligation at that exit point.

Verification conditions. $VC_{\pi}(\pi)$ checks the abstract state immediately before each return site of π:

- (P1_{vc}): under $\sigma^{\top}_{start}$, at every `return Ready`(v): $S(\hat{p}) = \mathtt{tt}$;
- (P2_{vc}): under $\sigma^{\mathtt{tt}}_{start}$, every reachable return site is a `Ready`-return;
- (P3_{vc}): under $\sigma^{\top}_{start}$, at every `return Pending`: $\widehat{L}_{end} = \varnothing$.

The three checks discharge (P1), (P2), and (P3) of Definition 2, respectively. (P3_{vc}) witnesses that the locally-held obligation has been transferred to a public repository (via `insert`) or discharged (via `wake`) by the time the pollable yields.

Example: The Timer pollable in Section 3.1 can be modeled in $\mathcal{L}_{\mathsf{ART}}$ as

$$\begin{aligned}\pi_{timer} \triangleq\ & \lambda(w, p_c).\\ & \texttt{insert}(\texttt{A}, w, p_c)\,;\\ & \texttt{if } p_c \texttt{ then } \{\ \texttt{return Ready}(())\ \} \texttt{ else } \{\ \texttt{return Pending}\ \}\end{aligned}$$

where $p_c \in \mathcal{P}_{ext}$ is the timer's completion predicate (i.e., `self.instant <= now()`) and `A` is the Reactor's public repository.

When running the analysis with $\sigma^{\top}_{start}$, `insert`$(\texttt{A}, w, p_c)$ moves the entry obligation $(\hat{w}, p_c, \top)$ from $\widehat{L}$ to $\widehat{R}(\texttt{A})$, leaving $\widehat{L} = \varnothing$. The `if` statement refines the state of p_c to $S(p_c) = \mathtt{tt}$ on the then-branch and $S(p_c) = \mathtt{ff}$ on the else-branch. On the then-branch we reach `return Ready`$(())$ with $S(p_c) = \mathtt{tt}$, discharging (P1_{vc}). On the else-branch we reach `return Pending` with $\widehat{L}_{end} = \varnothing$, discharging (P3).

When running the analysis with $\sigma^{\mathtt{tt}}_{start}$: the else-branch refinement gives $S(p_c) = \mathtt{tt} \triangleright \mathtt{ff} = \bot$, so only the `Ready`-return is reachable, discharging (P2). $VC_\pi(\pi_{timer})$ holds.

5.3 Reactor Analysis Verification Conditions

For each public repository r of type (T_r, cp_r), we analyze its implementation from an initial $\widehat{R}_{start}(r)$ with *two* symbolic obligations:

$$(\hat{w}^{\mathtt{tt}}_r,\, \hat{p}^{\mathtt{tt}}_r,\, \mathtt{tt}), \qquad (\hat{w}^{\top}_r,\, \hat{p}^{\top}_r,\, \top),$$

where $\hat{p}^{\mathtt{tt}}_r, \hat{p}^{\top}_r \in img(cp_r)$. The initial observation function is $S_{start}(\hat{p}^{\mathtt{tt}}_r) = \mathtt{tt}$ and $S_{start}(\hat{p}^{\top}_r) = \top$; $\widehat{L}_{start} = \varnothing$. The $\mathtt{tt}$-tagged obligation models the case where the completion predicate already holds before the iteration (for checking (R1)); the $\top$-tagged obligation models the case where the predicate's state is unknown (for checking (R2)). Given a reactor body `loop` s, the analysis runs *once* on s with the initial state σ_{start}, it computes the end-of-iteration state $\sigma_{end} = \mathcal{T}[\![s]\!](\sigma_{start})$; no fixpoint over `loop` is needed.

Verification conditions. $VC_{\mathcal{R}}(s)$ inspects σ_{end} and quantifies over the symbolic obligations seeded at iteration start.

- (R1$_{vc}$): for every $\hat{o} = (\hat{w}, \hat{p}, \mathtt{tt}) \in \widehat{R}_{start}(r)$, $\hat{o} \notin \widehat{L}_{end}$ and $\hat{o} \notin \bigcup_{r'} \widehat{R}_{end}(r')$.
- (R2$_{vc}$): for every $\hat{o} = (\hat{w}, \hat{p}, \top) \in \widehat{R}_{start}(r)$, $\hat{o} \notin \widehat{L}_{end}$.

$VC_{\mathcal{R}}(s) \triangleq \mathrm{R1}_{vc} \wedge \mathrm{R2}_{vc}$. R1$_{vc}$ witnesses that all obligations whose predicates hold at iteration start have been consumed by `wake` and removed from public repositories. R2$_{vc}$ witnesses that obligations with unknown predicates are either discharged by `wake` (hence absent from $\widehat{L}$ and every $\widehat{R}$) or preserved in some public repository for future iterations.

Example: The Timer Reactor of Sec. 3.1 can be encoded in $\mathcal{L}_{\mathsf{ART}}$ as

$$\begin{aligned}\mathcal{R}_{timer} \triangleq\ & \mathtt{reactor}\,\{\ \mathtt{shared}\ \mathtt{A} : (Timer, cp_{\mathtt{A}})\ \mathtt{in} \\ & \quad \mathtt{loop\ for}\ (w, p)\ \mathtt{in}\ \mathtt{A}\ \{ \\ & \quad\ \ \mathtt{if}\ p\ \mathtt{then}\ \{\ \mathtt{take}(\mathtt{A}, w)\,;\ \mathtt{wake}(w)\ \}\ \mathtt{else}\ \mathtt{skip}\ \}\}\end{aligned}$$

We initialize the repository map $\hat{R}_{start}$ with the two symbolic obligations introduced above, i.e., $\widehat{R}_{start}(\mathtt{A}) = \{\hat{o}^{\mathtt{tt}}, \hat{o}^{\top}\}$ and set the local obligation set to empty $\widehat{L}_{start} = \varnothing$.

The `for`-rule instantiates the body once per symbolic obligation. For the instantiation for $\hat{o}^{\mathtt{tt}}$, since we assumed $S(\hat{p}^{\mathtt{tt}}) = \mathtt{tt}$ at iteration start, the else-branch refines to $\bot_{\textsc{State}}$; only the then-branch survives. $\mathtt{take}(\mathtt{A}, \hat{w}^{\mathtt{tt}})$ moves $\hat{o}^{\mathtt{tt}}$ into $\widehat{L}$ and subsequently $\hat{o}^{\mathtt{tt}}$ is consumed by $\mathtt{wake}(\hat{w}^{\mathtt{tt}})$. After the if-statement, $\widehat{R}(\mathtt{A}) = \{\hat{o}^{\top}\}$ and $\widehat{L} = \varnothing$.

For the instantiation with $\hat{o}^{\top}$, both branches survive; the join unions $\widehat{R}$ and $\widehat{L}$. After the for-loop, we have $\widehat{R}_{end}(\mathtt{A}) = \{\hat{o}^{\top}\}$ (contributed by the else-branch) and $\widehat{L}_{end} = \varnothing$.

We see that the symbolic obligation $\hat{o}^{\mathsf{tt}}$ is not in $\widehat{L}_{end}$ or $\widehat{R}_{end}(\mathtt{A})$ by the end of the iteration, hence $\mathrm{R1}_{vc}$ holds. Meanwhile, obligation $\hat{o}^{\top}$ only exists in $\widehat{R}_{end}(\mathtt{A})$, hence $\mathrm{R2}_{vc}$ holds and the reactor is verified.

Example (buggy): Now consider a buggy implementation of the same reactor obtained by replacing the then-branch's `wake` with `skip` (the reactor takes the obligation out of `A` but forgets to wake it). The iteration with obligation $\hat{o}^{\mathsf{tt}}$ never discharges the obligation in the then-branch, leaving $\hat{o}^{\mathsf{tt}} \in \widehat{L}_{end}$, hence failing $\mathrm{R1}_{vc}$.

5.4 Soundness

In the subsection, we state the key soundness theorems and provide proof sketches for them. We first show that the verification conditions soundly approximate the responsiveness properties for pollables and reactors as we defined in Defintion 2 and Definition 3.

Theorem 2 (Pollable Soundness). *Let π be a pollable in $\mathcal{L}_{\mathsf{ART}}$. If $VC_{\pi}(\pi)$ holds, then π is responsive (Definition 2).*

Proof sketch. The concrete semantics in Fig. 5 *simulates* the transfer functions defined in Fig. 6; the abstract state at every program point over-approximates the concrete state.

(P1) follows because VC_{π} enforces $S(\hat{p}) = \mathtt{tt}$ at every `Ready`-return site, which by simulation gives $p \in \Sigma$ in every concrete trace. (P2) follows because, when $S(\hat{p})$ is initialized to $\mathtt{tt}$, VC_{π} rules out reaching any `Pending`-return site, so concrete executions starting with $p \in \Sigma$ at entry must return `Ready`. (P3) follows because VC_{π} enforces $\widehat{L}_{end} = \varnothing$ at every `Pending`-return site; an obligation can only leave $\widehat{L}$ either via `insert` (placing it in $\widehat{R}$) or `wake` (firing w), so by simulation the concrete trace either transfers (w, p) to a public repository or fires w before returning `Pending`.

Theorem 3 (Reactor Soundness). *Let $\mathcal{R}$ be a reactor in $\mathcal{L}_{\mathsf{ART}}$ whose body is `loop` s. If $VC_{\mathcal{R}}(s)$ holds, then $\mathcal{R}$ is responsive (Definition 3).*

Proof sketch. By simulation, $\sigma_{end} = \mathcal{T}[s](\sigma_{start})$ over-approximates every concrete iteration-end configuration. The symbolic obligation $\hat{o}^{\mathsf{tt}}$ over-approximates all obligations whose monotonic predicates hold at the start of an iteration. $\mathrm{R1}_{vc}$ implies that the iteration both removes all corresponding obligations from every public repository (via `take`) and consumes them locally (via `wake`), so the concrete wakers are in $Wake_i$ (the set of wakers being woken up in iteration i); therefore (R1) holds.

$\hat{o}^{\top}$ over-approximates both obligations present at iteration start (preserved from previous iterations) and those inserted concurrently during the iteration. $\mathrm{R2}_{vc}$ implies that, on every path, such concrete obligations are either discharged via `wake` or preserved in some public repository, satisfying (R2).

Theorem 4 (Reactor Eventual Progression). *Let* `loop` s *be a verified reactor and* (w, p) *an obligation ever placed in one of its public repositories; the following holds*

$$\Box\big(p \in \Sigma \;\Rightarrow\; \Diamond\, w \in \mathcal{Q}\big).$$

Proof sketch. Theorem 3 gives that verified reactors satisfy Definition 3. By (R2) of Def. 3, (w, p) is preserved at every iteration end, so it will remain in some public repository from when it was first inserted until `wake`(w) fires. By monotonicity of the predicate p, once $p \in \Sigma$ it remains so; since the reactor iterates infinitely often, there will be an iteration j at whose start $p \in \Sigma$, and (unless `wake`(w) has already been called) (w, p) will be still present in some repository. By (R1) of Def. 3, w will be woken up at iteration j, i.e., $w \in \mathit{Wake}_j$, and by the (wake) rule in Fig. 5, eventually $w \in \mathcal{Q}$. This discharges Definition 1 case (b).

Theorems 2 and 4 together establish the runtime side of Eventual Progression: every obligation introduced into the runtime, whose monotonic predicate eventually holds, eventually has its waker fired. Combined with the three trusted components of Section 3.2 (a fair scheduler, a correct compiler, and a responsive environment), the runtime delivers the end-to-end responsiveness property below.

Theorem 5 (Runtime Responsiveness). *Suppose every reactor and every pollable in the runtime satisfies its verification condition, async-task compilation is faithful, the scheduler is fair, and the external predicates a task depends on eventually hold. Then every async task running in this runtime eventually makes progress.*

Proof sketch. Faithful compilation transforms every async function into a state-machine pollable whose `poll` method delegates to the awaitees' `poll` method.

By induction on the depth of nested `await`s, the base case (a task awaiting a runtime pollable) follows from Theorems 2 and 4.

For the inductive case (a task t awaiting a sub-task t'), π_t's `poll` invokes $\pi_{t'}$'s `poll`. The inductive hypothesis gives that $\pi_{t'}$ eventually returns `Ready`; faithful compilation propagates the result to π_t, which continues past the await.

Corollary 1 (Responsiveness Attribution). *If an async task is observed to hang at a program point, at least one of the trusted components must have failed, or the user's code itself has unintended non-termination or deadlocks independently of the runtime's pollables. The verified pollable and reactor implementations provided by the async runtime* cannot *be the cause.*

The corollary above is the contrapositive of Theorem 5. In practice, our analyses rule out these async runtime's implementations from causing responsiveness issues.

$$\text{Reactors} \quad \mathcal{R} ::= \ldots \mid \texttt{t-reactor}\ \{\ \texttt{shared}\ \overrightarrow{r_s : (T_{r_s}, cp_{r_s})}\ \texttt{in}\ s\ \} \quad (wfr(s))$$

$$\mathcal{T}[\texttt{insert}(r, w, p)](\sigma) = \sigma\big[\, S(tfr) := \mathsf{tt},\ \ \widehat{R}(r) := \widehat{R}(r) \cup \{(\hat{w}, p)\},\ \ \widehat{L} := \widehat{L} \setminus \{(\hat{w}, p)\}\,\big]$$
$$\sigma|_{p \mapsto \mathsf{ff}} = \sigma\big[\, S(p) := S(p) \triangleright \mathsf{ff},\ \ \prec := (\prec \cup \{(q, p) : q \neq p,\ S(q) = \mathsf{tt}\})^{+}\,\big]$$

Fig. 8: Extensions for terminating reactors. *Syntax*: the new `t-reactor` construct. *Transfer function*: the two transfer rules for recording strict-obligation-transfer extension: `insert` records the transfer event *tfr*, and refining a predicate to ff extends $\prec$ (X^+ denotes transitive closure). The state smashes to $\bot$ if $S(p) \triangleright \mathsf{ff} = \bot$; all other forms are unchanged.

5.5 Extension 1: Terminating Reactors and Precedes Relations

Our model so far requires a reactor to run an infinite loop over its repositories, which is the most-common idiom. In this subsection we show an adaptation of our techniques to support reactor-like components which handle only *finitely many* wakers, and can thus justify termination.

This idiom of *terminating reactors* arises in practice e.g., in the implementation of the receiving end of a one-shot channel. The receiver is an async future awaiting a value, while its peer code `send` is an ordinary *synchronous* function. Calling `send` delivers the value and wakes the suspended receiver, which is exactly the obligation a reactor discharges. We can see `send` as playing a reactor-like role, despite being a single function call rather than an event loop.

Two things set such a reactor apart from those of Section 4. First, it discharges a finite, bounded collection of obligations rather than an unbounded stream: a one-shot channel has at most one receiver to wake. Second, it runs a statically bounded number of times (one, in this example) and then terminates. Rust's ownership discipline consumes the sender, so `send` fires at most once per channel; after that single invocation the reactor is dead, and any obligation registered afterwards can never be discharged. We model this as a *terminating reactor* `t-reactor` (Fig. 8), whose body s runs to completion once per invocation, with no enclosing `loop`. Its well-formedness predicate *wfr* is that of a regular reactor (Section 4.1), minus the required outermost `loop`.

Verifying a `t-reactor` reuses most of our existing design with only modest adaptation. $VC_{\mathcal{R}}$ and the abstract domain are unchanged, with (R1) and (R2) of Definition 3 read *per invocation* rather than per iteration; Reactor Soundness (Theorem 3) transports directly, since its proof already reasons about a single execution of the body.

Reactor Eventual Progression (Theorem 4), by contrast, does *not* translate as directly: recall that its proof relies on the reactor iterating infinitely often: an obligation is preserved across iterations by Definition 3 (R2) and woken by (R1) at some later iteration when its predicate holds. A terminating reactor runs only

once, so an obligation inserted after that single invocation is preserved by (R2) but never discharged.

Progression therefore holds only for obligations whose transfer *strictly precedes* termination of the reactor. The observation function S in the current abstract state cannot witness this alone: a receiver that re-checks completion after transferring its obligation and one that does not both reach `Pending` with the same observation $S(p_c) = \mathsf{ff}$.

Building on our first-class notion of monotonic predicates, we extend the abstract domain of Section 5.1 with a *precedes relation* $\prec \subseteq \mathcal{P} \times \mathcal{P}$: a strict partial order representing observed temporal order between two monotonic predicates. In particular, $p \prec q$ means there *exists* a moment with $p \in \Sigma$ and $q \notin \Sigma$. This relation (and its transitive closure) can be efficiently maintained by our analysis.

We extend abstract states with an additional set $\prec$ to capture all the observed precedes relations, i.e., $\sigma = (S, \prec, \widehat{R}, \widehat{L})$. The set $\prec$ joins by intersection, keeping only orderings established on all incoming paths.

We additionally initialize S with a dedicated monotonic predicate *tfr* for "the obligation has been transferred to the reactor". The *strict obligation transfer (SOT)* condition is $\mathit{tfr} \prec p_c$: the obligation must be transferred *before* the completion predicate holds.

The precedes relations can be tracked via two updated transfer rules, presented in Fig. 8 with no new annotation. First, $\texttt{insert}(r, w, p)$ means the obligation has been transferred to a reactor; therefore, besides moving the obligation to $\widehat{R}(r)$, we also set $S(\mathit{tfr}) = \mathsf{tt}$. Second, refining $S(p)$ to ff (e.g., by entering the else-branch of an if-statement) extends $\prec$ with $q \prec p$ for every q already observed to be true ($S(q) = \mathsf{tt}$).

Let $\mathit{Term} \subseteq \textsc{Repo}$ denote the set of repositories owned by terminating reactors. The pollable verification condition VC_π is strengthened by a check on *strict-obligation-transfer* that fires only when the pollable transfers its entry obligation into some $r \in \mathit{Term}$:

- (SOT_{vc}): under $\sigma^{\top}_{\mathit{start}}$, at every reachable `return Pending` such that $(\hat{w}, \hat{p}, _) \in \widehat{R}(r)$ for some $r \in \mathit{Term}$: $\mathit{tfr} \prec \hat{p}$.

A re-checking receiver that performs `insert` (refining $S(\mathit{tfr})$ to tt) followed by a re-check (refining $S(p_c)$ to ff) now records the precedes relation $\mathit{tfr} \prec p_c$, hence satisfying (SOT_{vc}); a non-re-checking receiver will fail (SOT_{vc}).

Theorem 6 (Terminating Reactor Eventual Progression). *Let r be the repository of a terminating reactor whose body s satisfies $VC_{\mathcal{R}}(s)$, and let (w, p) be an obligation inserted into r by a pollable satisfying* $\mathrm{SOT}_{\mathrm{vc}}$ *for r. If the host invokes the reactor, then*

$$\Box\big(\, p \in \Sigma \;\Rightarrow\; \Diamond\, w \in \mathcal{Q}\,\big).$$

Proof sketch. $\mathrm{SOT}_{\mathrm{vc}}$ gives $\mathit{tfr} \prec p$: the registration of w into r strictly precedes p entering Σ. The host's invocation establishes p (it `triggers` p, or observes it already tt) and, by $VC_{\mathcal{R}}(s)$/(R1), `takes` and `wakes` every obligation in r whose

predicate holds. Since w was registered strictly before p, (w, p) is present in r at invocation time; (R1) fires `wake`(w), and by the (wake) rule eventually $w \in \mathcal{Q}$. A single invocation thus replaces the "iterates infinitely often" step of Theorem 4.

The proof leaves one assumption uncovered, mirroring the regular reactor's reliance on its external predicates eventually holding: the host must eventually invoke the terminating reactor (e.g., call `send`). If a one-shot channel is never sent on, its receiver legitimately waits forever, a user-level failed obligation rather than a runtime fault.

5.6 Extension 2: Fixpoint for Pollable Retry-loops

The base language requires pollable bodies to be loop-free (*wfp*, Section 4.1). Sometimes, however, pollables themselves contend for bounded shared resources (a channel slot, a mutex permit), following a *try-wait-retry* cycle: attempt to acquire the resource; on failure, `insert` a waker and yield; on wake, retry. The reinvocation of pollables on wake only implies that the contended resource *may* now be available, but success on the second call is not guaranteed.

Our modular analysis techniques do not support this pattern as described, which would require some kind of analysis around the implicit loop illustrated in Fig. 1. However, a variant of this pattern which we have encountered in practice is to use an explicit *try-wait-retry* loop in the pollable implementation itself; we describe here an extension which supports this case.

To support such retry-loops, we can admit `loop` in pollable bodies and discharge it with a fixpoint; VC_π, both initial states, and the other transfer rules of Section 5 are unchanged. The well-formedness condition *wfp* is relaxed accordingly: `loop` s is admitted in any statement position, and the body s must contain *at least* one `return`, so that the loop exit is reachable. Our analysis does *not* attempt to check termination of this loop, which ultimately depends on an external predicate eventually holding; instead, this extension to pollable loops checks that *provided* the loop eventually terminates, the necessary obligations will be satisfied.

Our fixpoint analysis converges because the abstract domain is a finite complete lattice and the transfer function is monotone. For any pollable, the monotonic predicates, symbolic wakers, and repository names appearing in the pollable body are finite. Therefore, the symbolic obligations $\widehat{\textsc{Ob}} \subseteq \widehat{\textsc{Waker}} \times \mathcal{P} \times \{\mathtt{tt}, \top\}$ and the space of observation maps S are also finite. The state space $(\textsc{State}, \sqsubseteq, \sqcup, \bot_{\textsc{State}})$ is therefore a finite, complete lattice, with $\sqcup$ monotone in both arguments. Monotonicity of $\mathcal{T}[\![s]\!]$ follows by structural induction on s: every rule of Fig. 6 is built from monotone operations on $\textsc{Obs}$ and $\wp(\widehat{\textsc{Ob}})$.

The transfer function definitions extend with one new case for loops:

$$\begin{aligned} \mathcal{T}[\![\texttt{loop}\ s]\!](\sigma) \;&=\; \bot_{\textsc{State}}, \\ h^* \;&\triangleq\; \mathrm{lfp}\big(\, h \mapsto \sigma \sqcup \mathcal{T}[\![s]\!](h) \,\big). \end{aligned}$$

Here h^* is the abstract state computed as the fixpoint above. A `loop` exits only through a `return`, so its result state is $\bot_{\textsc{State}}$; the return-site states that VC_π

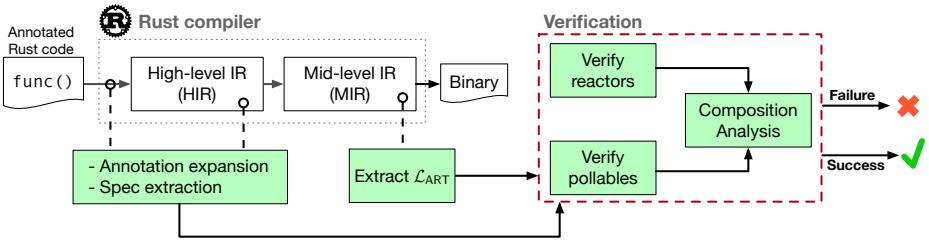

Fig. 9: JACKDAW design. The green boxes are part of JACKDAW.

inspects are gathered from inside s during the evaluation of $\mathcal{T}[s](h^*)$, exactly as the base analysis collects them for non-loop bodies, just with h^* in place of the pollable's entry state. The body's transfer $\mathcal{T}[s](h)$ collects exactly the loop-back paths because any `return` inside s yields $\bot_{\text{STATE}}$, the identity of $\sqcup$. The chain $h_0 = \bot_{\text{STATE}}$, $h_{n+1} = \sigma \sqcup \mathcal{T}[s](h_n)$ ascends and stabilises in finitely many steps. This generalizes to nested loops because monotonicity of the body's transfer is preserved by structural induction.

The proof of Pollable Soundness (Theorem 2) has a new case for the loop. By induction on the iteration number, every concrete state at the loop head is abstracted by h^*: the entry state by $\sigma \sqsubseteq h^*$, and any future-iteration state, obtained by running the body once from a previous state, by $\mathcal{T}[s](h^*) \sqsubseteq h^*$. Return-site observations collected under h^* therefore over-approximate those of every concrete iteration, so VC_π remains sound.

6 The JACKDAW Tool and Evaluation

We implement our technique as a prototype, JACKDAW, in roughly 5k lines of Rust. JACKDAW is designed as a Rust compiler plugin [5]; see Figure 9. JACKDAW enables modeling pollables/reactors of Rust async runtimes in $\mathcal{L}_{\text{ART}}$ via annotations and extracts an explicit $\mathcal{L}_{\text{ART}}$ model out of Rust's intermediate representation (Middle-level IR, MIR). The extracted model can be manually inspected, and also automatically verified against VC_π and $VC_\mathcal{R}$ as described in Section 5. In Sec. 6.1 we use JACKDAW to evaluate (1) the expressiveness of $\mathcal{L}_{\text{ART}}$ by modeling real-world pollables and reactors, and (2) the effectiveness of our static analyses.

Users model a pollable/reactor as an instance of $\mathcal{L}_{\text{ART}}$ through a macro library that overlays $\mathcal{L}_{\text{ART}}$ constructs on Rust code. Macros expand to no-op markers in MIR for model extraction. The annotated Rust code compiles and behaves identically to the original. The macro syntax mirrors $\mathcal{L}_{\text{ART}}$: `mono_pred!` declares a monotonic predicate; `obs!`, `obs_tt!`, and `obs_ff!` record observations on predicates; `trigger!`, `wake!`, `insert!`, and `take!` mark the corresponding effects; and `repo_for!` marks iteration over a repository. The entry points of pollables/reactors are tagged with `#[jackdaw::pollable]` and `#[jackdaw::reactor]`,

Table 1: Verification statistics for pollables.

Pollable	Runtime	LoC	Anno LoC	Ratio	Analysis time (μs)
EventListener	smol	15	5	33%	24.6
MPSC-send	smol	28	6	21%	68.0
MPSC-recv	smol	27	6	22%	63.2
MPSC-close	smol	21	4	19%	39.0
AsyncIO-Read/Write	smol	29	6	21%	99.5
AsyncIO-Ready	smol	38	5	13%	86.2
OneShot-recv	futures	23	8	35%	82.8
OneShot-recv	tokio	59	11	19%	152.2
OneShot-closed	tokio	45	10	22%	111.1

and `bind!` binds the $\mathcal{L}_{\mathsf{ART}}$ parameters of a component (its waker, completion predicate, and public repositories) explicitly rather than having them inferred.

Jackdaw traverses the MIR of annotated functions, captures the control-/data-flows and user annotations, and lifts the pollable/reactor implementations into $\mathcal{L}_{\mathsf{ART}}$. The lifting from MIR to $\mathcal{L}_{\mathsf{ART}}$ is not formalized and contains heuristics (e.g., recovering nested branches and loops from a CFG).

The $\mathcal{L}_{\mathsf{ART}}$ formalized in Sec. 4 and the static analyses described in Sec. 5 are fully implemented (including the extensions). We add an assume statement $\texttt{assume}(p, b)$ that refines a predicate p to a specific value b without branching, and a nondeterministic choice `nd-match` for more straightforward translation from Rust.

6.1 Evaluation

We evaluate Jackdaw on a representative subset of components drawn from three Rust async runtimes: smol [25], futures [24], and tokio [28]. These span the three building blocks of async programs: scheduling (timers), I/O, and communication (one-shot and MPSC channels). Tables 1 and 2 list every verified component with its source size (LoC, excluding blank lines and comments), total annotation cost (Anno LoC; Ratio = Anno / LoC), and per-component *analysis time*, the cost of lifting to $\mathcal{L}_{\mathsf{ART}}$ and discharging the verification conditions, measured as the mean of 300 runs. Experiments ran on a 2.30 GHz Intel Core i9-9880H with 32 GB RAM under `rustc 1.91.0-nightly`.

Effectiveness. For each verification condition (P1/P2/P3 of VC_π, R1/R2 of $VC_\mathcal{R}$, and the strict-obligation-transfer condition from Section 5.5), we seed a violation in a real component and confirm Jackdaw rejects it; no false negatives are observed across the suite.

Specification burden. Each pollable verifies with 4 to 11 lines of annotation, giving an annotation-to-code ratio of 23% on average. Every pollable in the suite is specified with a *single* monotone predicate. Terminating reactors show a higher ratio because of their short implementations (≤ 13 lines).

Table 2: Verification statistics for reactors.

Reactor	Runtime	Kind	LoC	Anno LoC	Ratio	Analysis time (μs)
Timer / AsyncIO	smol	looping	59	11	19%	38.5
OneShot-recv	futures	terminating	12	7	58%	10.9
OneShot-recv	tokio	terminating	12	7	58%	9.9
OneShot-closed	tokio	terminating	13	7	54%	10.6

Imprecision/Code modification. We encountered three cases where additional annotations or source code modifications are needed to overcome JACKDAW's imprecision or limitations. 1. Post-join re-assertion is needed when a completion decision is carried in a boolean flag across control-flow merges. This is a precision limitation of our abstract domain. 2. Our model cannot handle tokio's fairness mechanism, which requires pollables to cooperatively yield. We use a special annotation in JACKDAW, `ignore_path!`, to prune such unsupported code paths. 3. Tokio's OneShot-recv pollable contains paths requiring cross-invocation invariants to justify their returns; this requires additional ghost annotations to establish such invariants.

Overhead. Per-component analysis is sub-millisecond throughout (10 to 152 μs; Tables 1 and 2). Lifting dominates and scales with control-flow complexity; the heaviest case is tokio's one-shot receiver with its coop/`will_wake`/state branches at ≈152 μs, while reactors are 10 to 40 μs. These costs are negligible relative to Rust compilation, making JACKDAW suitable for integrating more expensive analysis/verification techniques.

7 Limitations and Future Work

We discuss some of the current limitations and potential future work for our technique and its prototype implementation. Known gaps in the existing model of $\mathcal{L}_{\mathsf{ART}}$ include: a *wake-one* primitive (non-deterministically discharge one obligation in a repository) for single-permit synchronisation idioms (e.g., tokio's `Mutex`, `Semaphore`), and cross-poll reasoning that would let invariants established in a prior poll discharge later checks.

Sound extraction of $\mathcal{L}_{\mathsf{ART}}$ models. Our soundness theorems are stated with respect to $\mathcal{L}_{\mathsf{ART}}$, but real programs are written in Rust. JACKDAW bridges the two by lifting MIR to $\mathcal{L}_{\mathsf{ART}}$ using a (currently trusted but fairly straightforward) translation of control flow, and, crucially, trusted user-supplied annotations. In particular, JACKDAW does not check for mistakes in these annotations, such as triggering a monotonic event when the corresponding update has not been performed in the real memory. We plan to investigate scalable approaches to close this gap or reduce the chance of errors in the user's modelling, by integrating other complementary static or dynamic analysis techniques.

Fine-grained concurrency. All operations in $\mathcal{L}_{\mathsf{ART}}$, e.g., `insert`, `take`, and `wake` are atomic operations under a sequential consistency model; our techniques, and especially the precedes relation extension defined in Sec. 5.5 rely on sequential consistency. Real-world async runtime implementations such as `tokio` use weaker atomic memory accesses for high performance; so far, we have not seen examples in which extra observable behaviors could result, but our technique is not checking this formally when such primitives are used; adding such checks directly to our analyses in a way that matches the idioms used in this kind of code might be possible in future.

Cancellation and task drop. Async tasks can typically be cancelled at any await point, at which time any wakers the task has registered with reactors must be reclaimed. Our model currently has no notion of obligation revocation and no mechanism for a reactor to learn that a transferred obligation becomes irrelevant; adding this to our operational semantics would be straightforward. Although challenging in terms of general logic of user code, from the pollable and reactor perspectives we believe this extension should be reasonably simple, especially since wakers typically include logic to handle unnecessary extra calls.

Safety and functional correctness. Our analyses currently only check that obligations arisen from pollables managed by an async runtime are not lost, and we assume properties such as panic-freedom (and type-correctness, in $\mathcal{L}_{\mathsf{ART}}$). It might be interesting in future to also be able to prove panic-freedom as well as to prove a formal basis for more-general verification of functional properties guaranteed by the runtime's mechanisms. We believe $\mathcal{L}_{\mathsf{ART}}$ could be used as a basis for exploration of such techniques in the future.

Bounded responsiveness. Finally, eventual progression is a specific responsiveness property that only asserts progress in the arbitrarily-far future. Users typically care about finer-grained responsiveness properties, e.g., bounded latency, tail latencies, and fairness across tasks of different priorities. Lifting our technique to stronger responsiveness properties (given some reasonable timing model) and/or trying to infer such guarantees would be interesting future work.

8 Related Work

Our work addresses the problem of verifying eventual progression, a liveness property ensuring that asynchronous runtimes make progress and do not indefinitely defer pending tasks. Some prior work has proposed approaches to reason about *user code* [9,16,8,19] that relies on an async runtime. However, the underlying async runtime is typically considered part of the trusted computing base. Instead, our work treats errors in *user code* (or other components such as the OS) as an orthogonal concern and focuses on critical runtime components that almost all user code implicitly relies upon.

Compositional Verification of Responsiveness. Our work takes inspiration from Actor Services [26], which introduced a modular logic for verifying responsiveness and functional correctness of purely-actor-based programs. Actor Services employed a notion of logical obligations (reminiscent of "causal obliga-

tions" of Helm et al. [13]), to structure progress proofs under fair scheduling. Again, they address verification of *user code*, while we focus on the more-specific but crucial concern of the responsiveness of Rust async runtime components.

We aim for similarly-modular reasoning to Actor Services, but our technique differs in several important aspects. We can verify implementation strategies that Actor Services cannot (in particular, reactor-like components which do not discharge obligations immediately, but which promise progress in a different fashion): reactors which handle multiple obligations in an interleaved fashion cannot be verified with this prior work. We also identify and build in monotonic events and their orderings as a first-class notion, which allows us to express subtle requirements that cannot be captured as responsiveness properties directly. We have also applied our technique to real-world code with JACKDAW, enabling experimentation and evidence of potential impact in practice.

Anvil [27] similarly tackles liveness reasoning in Rust systems but in a very different setting. It provides a framework for verifying liveness properties of Kubernetes controllers using TLA specifications embedded within Verus [17]. Anvil's "Eventually Stable Reconciliation" (ESR) property is conceptually related to our notion of Eventual Progression. Anvil requires users to encode their system logic manually in TLA and to prove ESR interactively within Verus. In contrast, we automatically verify liveness guarantees of async runtimes through lightweight specifications, rather than a user-driven proof framework.

Obligations have been used in other verification contexts: Leino et al. [18] proposed obligations as a first-class specification notion for enabling modular proofs of deadlock-freedom. This idea was extended by Boström and Müller [4] to enable verification of finite blocking for concurrent algorithms. These works are aimed at user code and do not directly apply to our setting; conversely, the simplifications afforded by our task-centric view or decomposition of responsibilities across pollables and reactors cannot be directly exploited for user code.

Verification for Rust Programs. Rust's ownership model has motivated a growing body of verification work. RustBelt [15] provides a foundational semantics and verification technique based on Rust's type system, and deductive verifiers such as Prusti [2,1], Creusot [7], and Verus [17] support semi-automated functional correctness proofs. These techniques primarily target improvements to sequential reasoning made possible by Rust's guarantees; our key reasoning concepts of monotonic events and their orderings are novel. To our knowledge, none of these tools or techniques apply to the Rust async setting.

Cutner and Yoshida [6] formalize session-based async coordination of Rust user code that *uses* async primitives such as channels. This helps prevent user code errors such as deadlocks; the responsibilities of the runtime are assumed in such work (as standard), while our work addresses these responsibilities directly.

Verification of Liveness / Async Runtimes. To our knowledge, the first usage of monotonic predicates as a specific notion was in Hailpern and Owicki's pioneering work on modular specification and verification for liveness properties [12]. This work presents a wide variety of specification idioms and shows why monotonic properties have useful properties at the boundaries between compo-

nents. We take this idea further and make one-shot monotonic events into first class specification notions, along with the ability to specify and observe orderings of these events. In addition, we show how to connect our reasoning to concrete program implementations.

Model checking has been used extensively to verify liveness properties [3], sometimes using variations of rely-guarantee style proof techniques [14] to support modular verification [11]. While these techniques allow abstracting over the concrete interleavings of actions by other components, they do not provide the abstraction or simplicity of our task-centric view.

Liveness verification has been considered at the protocol level in the context of session types by Padovani et al. [22]; this work does not connect to concrete implementations or support modular specification of separate components.

Emílio de Vilhena and Pottier develop a framework for separation logic reasoning for effect handlers [29]. As part of the evaluation, they verified the safety property of an async runtime built with effect handlers. They do not consider liveness of async runtimes.

9 Conclusion

This work takes a first step toward verifying the *liveness guarantees* of async runtimes. Whereas prior research has primarily focused on verifying the safety and correctness of async programs, we shift attention to the runtimes that are responsible for ensuring that tasks eventually complete.

We developed a detailed yet generic model of Rust async runtimes that captures how reactors, pollables, and schedulers cooperate to provide responsiveness. We formalized this model in a core language $\mathcal{L}_{\mathsf{ART}}$ and identified two component-level verification conditions whose conjunction soundly implies eventual progression of the runtime. We automated these checks in Jackdaw, a compiler extension for Rust that uses lightweight annotations and static analysis. Our evaluation on `smol` and `tokio` async runtimes demonstrate that, with the two extensions of the core formalism, our approach verifies real pollable and reactor components written with various common idioms, while catching common responsiveness bugs.

Ultimately, our work advances the goal of *verified asynchrony*: connecting the high-level liveness guarantees expected by developers with the concrete behavior of runtimes that implement them.

References

1. Astrauskas, V., Bílỳ, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The prusti project: Formal verification for rust. In: NASA Formal Methods Symposium. pp. 88–108. Springer (2022)
2. Astrauskas, V., Müller, P., Poli, F., Summers, A.J.: Leveraging Rust types for modular specification and verification. Proc. ACM Program. Lang. **3**(OOPSLA), 147:1–147:30 (2019). https://doi.org/10.1145/3360573, https://doi.org/10.1145/3360573
3. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electronic Notes in Theoretical Computer Science **66**(2), 160–177 (2002)
4. Boström, P., Müller, P.: Modular Verification of Finite Blocking in Non-terminating Programs. In: Boyland, J.T. (ed.) European Conference on Object-Oriented Programming (ECOOP). LIPIcs, vol. 37, pp. 639–663. Schloss Dagstuhl (2015), https://doi.org/10.4230/LIPIcs.ECOOP.2015.639
5. Crichton, W., Adam, J., Gray, G., Coblenz, M.: rustc_plugin. https://github.com/cognitive-engineering-lab/rustc_plugin (2024)
6. Cutner, Z., Yoshida, N.: Safe Session-Based Asynchronous Coordination in Rust. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages. pp. 80–89. Springer International Publishing (2021)
7. Denis, X., Jourdan, J., Marché, C.: Creusot: A foundry for the deductive verification of Rust programs. In: Riesco, A., Zhang, M. (eds.) Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings. LNCS, vol. 13478, pp. 90–105. Springer (2022). https://doi.org/10.1007/978-3-031-17244-1_6, https://doi.org/10.1007/978-3-031-17244-1_6
8. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. **34**(1) (May 2012)
9. Ganty, P., Majumdar, R., Rybalchenko, A.: Verifying liveness for asynchronous programs. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 102–113 (2009)
10. Google: Listenablefuture (guava 21.0 api) (2017), https://guava.dev/releases/21.0/api/docs/com/google/common/util/concurrent/ListenableFuture.html, accessed: 2025-10-07
11. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 843–871 (1994)
12. Hailpern, B., Owicki, S.: Modular verification of concurrent programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 322–336 (1982)
13. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: Specifying behavioral compositions in object-oriented systems. In: OOPSLA/ECOOP 1990. pp. 169–180 (1990). https://doi.org/10.1145/97945.97967, http://doi.acm.org/10.1145/97945.97967
14. Jones, C.B.: Development methods for computer programs including a notion of interference. Oxford University Computing Laboratory (1981)
15. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–34 (2017)
16. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: Joint Meeting of the European Software

Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE). p. 13–22 (2009)

17. Lattuada, A., Hance, T., Bosamiya, J., Brun, M., Cho, C., LeBlanc, H., Srinivasan, P., Achermann, R., Chajed, T., Hawblitzel, C., Howell, J., Lorch, J.R., Padon, O., Parno, B.: Verus: A Practical Foundation for Systems Verification. In: Symposium on Operating Systems Principles (SOSP). p. 438–454 (2024)

18. Leino, K.R.M., Müller, P., Smans, J.: Deadlock-free channels and locks. In: Gordon, A.D. (ed.) European Symposium on Programming (ESOP). LNCS, vol. 6012, pp. 407–426. Springer-Verlag (2010)

19. Maiya, P., Kanade, A., Majumdar, R.: Race detection for Android applications. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). p. 316–325 (2014)

20. Mozilla Developer Network: Async functions — javascript | mdn (2025), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function, accessed: 2025-10-07

21. Oracle: Completablefuture (java platform se 8) (2014), https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html, accessed: 2025-10-07

22. Padovani, L., Vasconcelos, V.T., Vieira, H.T.: Typing liveness in multiparty communicating systems. In: COORDINATION 2014. LNCS, vol. 8459, pp. 147–162 (2014). https://doi.org/10.1007/978-3-662-43376-8_10, http://dx.doi.org/10.1007/978-3-662-43376-8_10

23. Python Software Foundation: asyncio — asynchronous i/o — python 3.12.6 documentation (2025), https://docs.python.org/3/library/asyncio.html, accessed: 2025-10-07

24. Rust Async Working Group: futures-rs — zero-cost asynchronous programming in rust (2025), https://docs.rs/futures/latest/futures/, accessed: 2026-05-26

25. Smol Contributors: smol — rust asynchronous runtime documentation (2025), https://docs.rs/smol/latest/smol/, accessed: 2025-10-07

26. Summers, A.J., Müller, P.: Actor services. In: European Symposium on Programming. pp. 699–726. Springer (2016)

27. Sun, X., Ma, W., Gu, J.T., Ma, Z., Chajed, T., Howell, J., Lattuada, A., Padon, O., Suresh, L., Szekeres, A., et al.: Anvil: Verifying liveness of cluster management controllers. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). pp. 649–666 (2024)

28. Tokio Contributors: Tokio — rust asynchronous runtime documentation (2025), https://docs.rs/tokio/latest/tokio/, accessed: 2025-10-07

29. de Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. Proceedings of the ACM on Programming Languages **5**(POPL), 1–28 (2021)

A Well-formedness Conditions

This appendix gives the well-formedness predicates *wfp* (for pollable bodies) and *wfr* (for reactor bodies) referenced in Section 4.1. They capture the programming paradigm not reflected in the bare syntax of $\mathcal{L}_{\mathsf{ART}}$ (Fig. 4).

Pollable bodies. A pollable π is a statement satisfying the well-formedness predicate *wfp*: every execution path contains a `return` v statement, and the statement contains no `loop` or reactor-only constructs, i.e., `take` and `for`. Formally, $\mathit{wfp}(s) \triangleq \mathit{returns}(s) \wedge \mathit{pollable\text{-}only}(s)$, where:

$$\begin{aligned}
\mathit{returns}(\texttt{return}\ v) &= \top \\
\mathit{returns}(s_1\ ;\ s_2) &= \mathit{returns}(s_1) \vee \mathit{returns}(s_2) \\
\mathit{returns}(\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &= \mathit{returns}(s_1) \wedge \mathit{returns}(s_2) \\
\mathit{returns}(s) &= \bot \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
\mathit{pollable\text{-}only}(\texttt{loop}\ s) = \mathit{pollable\text{-}only}(\texttt{take}(r, w)) &= \bot \\
\mathit{pollable\text{-}only}(\texttt{for}\ (w, p)\ \texttt{in}\ r\ \{\ s\ \}) &= \bot \\
\mathit{pollable\text{-}only}(s_1\ ;\ s_2) &= \mathit{pollable\text{-}only}(s_1) \wedge \mathit{pollable\text{-}only}(s_2) \\
\mathit{pollable\text{-}only}(\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &= \mathit{pollable\text{-}only}(s_1) \wedge \mathit{pollable\text{-}only}(s_2) \\
\mathit{pollable\text{-}only}(s) &= \top \quad \text{otherwise}
\end{aligned}$$

Relaxed pollable bodies (with loops). The loop extension admits `loop` s in pollable bodies. The relaxed predicate $\mathit{wfp}'(s) \triangleq \mathit{returns}'(s) \wedge \mathit{pollable\text{-}only}'(s)$ replaces the `loop` clauses of *returns* and *pollable-only* with

$$\begin{aligned}
\mathit{returns}'(\texttt{loop}\ s) &= \mathit{has\text{-}return}(s) \\
\mathit{pollable\text{-}only}'(\texttt{loop}\ s) &= \mathit{pollable\text{-}only}'(s)
\end{aligned}$$

where $\mathit{has\text{-}return}(s)$ asserts that s contains a `return` on at least one path:

$$\begin{aligned}
\mathit{has\text{-}return}(\texttt{return}\ v) &= \top \\
\mathit{has\text{-}return}(s_1\ ;\ s_2) &= \mathit{has\text{-}return}(s_1) \vee \mathit{has\text{-}return}(s_2) \\
\mathit{has\text{-}return}(\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &= \mathit{has\text{-}return}(s_1) \vee \mathit{has\text{-}return}(s_2) \\
\mathit{has\text{-}return}(\texttt{loop}\ s) &= \mathit{has\text{-}return}(s) \\
\mathit{has\text{-}return}(s) &= \bot \quad \text{otherwise.}
\end{aligned}$$

All other clauses of *returns* and *pollable-only* are unchanged. A `loop` s is admitted iff its body uses no `take` or `for` and contains a reachable `return`, so loop exit is structurally possible.

Reactor bodies. A reactor body satisfies *wfr*: it never `return`s (reactors are intended to monitor the environment indefinitely) and contains no `loop` outside the outermost `loop` that surrounds it. Formally:

$$\begin{aligned} \mathit{wfr}(\texttt{skip}) = \mathit{wfr}(\texttt{trigger}(p)) = \mathit{wfr}(\texttt{wake}(w)) &= \top \\ \mathit{wfr}(\texttt{insert}(r, w, p)) = \mathit{wfr}(\texttt{take}(r, w)) &= \top \\ \mathit{wfr}(\texttt{return}\ v) = \mathit{wfr}(\texttt{loop}\ s) &= \bot \\ \mathit{wfr}(s_1\ ;\ s_2) &= \mathit{wfr}(s_1) \wedge \mathit{wfr}(s_2) \\ \mathit{wfr}(\texttt{if}\ p\ \texttt{then}\ s_1\ \texttt{else}\ s_2) &= \mathit{wfr}(s_1) \wedge \mathit{wfr}(s_2) \\ \mathit{wfr}(\texttt{for}\ (w, p)\ \texttt{in}\ r\ \{\ s\ \}) &= \mathit{wfr}(s) \end{aligned}$$

B Full Operational Semantics for $\mathcal{L}_{\mathsf{ART}}$

Figure 10 gives the complete set of small-step program-semantics rules for $\mathcal{L}_{\mathsf{ART}}$. The main-paper version (Fig. 5) omits some conventional rules; the full set is reproduced here for reference.

$$\boxed{s \mid \kappa \longrightarrow_p s' \mid \kappa'}$$

$$\frac{p \in \mathcal{P}_{int}}{\texttt{trigger}(p) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma \cup \{p\}, \mathcal{Q}, R))} \text{ (trigger)}$$

$$\frac{}{\texttt{wake}(w) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q} \cup \{w\}, R))} \text{ (wake)}$$

$$\frac{R' = R[r \mapsto R(r) \cup \{(w,p)\}]}{\texttt{insert}(r,w,p) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q}, R'))} \text{ (insert)}$$

$$\frac{\begin{array}{c}(w,p) \in R(r) \\ R' = R[r \mapsto R(r) \setminus \{(w,p)\}]\end{array}}{\texttt{take}(r,w) \mid (\rho, (\Sigma, \mathcal{Q}, R)) \longrightarrow_p \texttt{skip} \mid (\rho, (\Sigma, \mathcal{Q}, R'))} \text{ (take)}$$

$$\frac{p \in \Sigma}{\texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2 \mid \kappa \longrightarrow_p s_1 \mid \kappa} \text{ (if-t)}$$

$$\frac{p \notin \Sigma}{\texttt{if } p \texttt{ then } s_1 \texttt{ else } s_2 \mid \kappa \longrightarrow_p s_2 \mid \kappa} \text{ (if-f)}$$

$$\frac{\begin{array}{c}R(r) = \{(w_1,p_1), \ldots, (w_k,p_k)\} \\ s_i \triangleq s[w_i, p_i / w, p] \quad k \geq 1\end{array}}{\texttt{for } (w,p) \texttt{ in } r \texttt{ \{ } s \texttt{ \} } \mid (\rho, G) \longrightarrow_p s_1 \,;\, \cdots \,;\, s_k \mid (\rho, G)} \text{ (for)}$$

$$\frac{R(r) = \varnothing}{\texttt{for } (w,p) \texttt{ in } r \texttt{ \{ } s \texttt{ \} } \mid (\rho, G) \longrightarrow_p \texttt{skip} \mid (\rho, G)} \text{ (for-empty)}$$

$$\frac{}{\texttt{loop } s \mid \kappa \longrightarrow_p s \,;\, \texttt{loop } s \mid \kappa} \text{ (loop)} \qquad \frac{s_1 \mid \kappa \longrightarrow_p s_1' \mid \kappa'}{s_1 \,;\, s_2 \mid \kappa \longrightarrow_p s_1' \,;\, s_2 \mid \kappa'} \text{ (seq)}$$

$$\frac{}{\texttt{skip} \,;\, s_2 \mid \kappa \longrightarrow_p s_2 \mid \kappa} \text{ (seq-skip)}$$

$$\frac{}{\texttt{return } v \,;\, s_2 \mid \kappa \longrightarrow_p \texttt{return } v \mid \kappa} \text{ (seq-return)}$$

Fig. 10: Full small-step operational semantics for $\mathcal{L}_{\mathsf{ART}}$.