Synchronizing the Asynchronous

Bernhard Kragl
IST Austria
https://orcid.org/0000-0001-7745-9117

Shaz Qadeer
Microsoft

Thomas A. Henzinger
IST Austria

Abstract

Synchronous programs are easy to specify because the side effects of an operation are finished by the time the invocation of the operation returns to the caller. Asynchronous programs, on the other hand, are difficult to specify because there are side effects due to pending computation scheduled as a result of the invocation of an operation. They are also difficult to verify because of the large number of possible interleavings of concurrent computation threads. We present *synchronization*, a new proof rule that simplifies the verification of asynchronous programs by introducing the fiction, for proof purposes, that asynchronous operations complete synchronously. Synchronization summarizes an asynchronous computation as immediate atomic effect. Modular verification is enabled via *pending asynchronous calls* in atomic summaries, and a complementary proof rule that eliminates pending asynchronous calls when components and their specifications are composed. We evaluate synchronization in the context of a multi-layer refinement verification methodology on a collection of benchmark programs.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases concurrent programs, asynchronous programs, deductive verification, refinement, synchronization, mover types, atomic action, commutativity, Lipton reduction

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2018.21

Funding This research was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

1 Introduction

This paper focuses on the deductive verification of *asynchronous concurrent programs*, an important class that includes distributed fault-tolerant protocols, message-passing programs, client-server applications, event-driven mobile applications, workflows, device drivers, and many embedded and cyber-physical systems. A key aspect of such programs is that (long-running) operations complete asynchronously. A process that invokes an operation does not block for the operation to finish. Instead, the result from the completion of the operation is communicated later, e.g., via a callback message. Asynchronous completion not only introduces concurrency and nondeterminism into the program semantics, but also makes the task of specifying the correct behavior of operations difficult. The behavior of a *synchronous* operation can be specified with a precondition and a postcondition because there is no ambiguity about the state just before and just after the operation executes. The behavior of an *asynchronous* operation is harder to specify because multiple operations can be in flight at the same time and partial results from other operations may have already affected the state before the operation finishes.

In this paper, we propose that reasoning about asynchronous computation can be simplified via *synchronization*, a program transformation that generalizes reduction [15, 6]. While reduction allows the creation of a coarse-grained atomic action from a sequence of fine-grained atomic actions performed by a single thread, synchronization allows the creation of a coarse-grained atomic action from an asynchronous computation executed by a potentially unbounded number of concurrent threads. Synchronization reduces the number of interleavings; it allows us to pretend, for the purposes of proof, that asynchronous calls complete synchronously and atomically, which leads to significantly simpler invariants.

Synchronization, similar to reduction, relies on commutativity properties of low-level atomic actions. Establishing commutativity may be difficult if these atomic actions access shared state that is also accessed by other, interfering concurrent computations. To enable synchronization in the presence of interference, we leverage the observation that commutativity properties among a set of atomic actions can be established by abstracting these actions [5]. In particular, we incorporate synchronization as a program transformation in the verification methodology of *program layers* [12], which allows the programmer to chain together a sequence of increasingly abstract concurrent programs containing atomic actions that are increasingly coarse-grained. Since program layers allow history variables to be introduced, history variables are sufficient for converting an arbitrary safety property into assertions, and the synchronization transformation preserves all assertion failures, our technique is applicable to the proof of arbitrary safety properties of asynchronous programs.

Synchronization, if used naively, leads to summaries that are not modular and hence not reusable. Consider a scenario where a client invokes an operation S of a service, upon whose completion a callback function C is invoked asynchronously. If the code of C is synchronized into S, the summary of S will be cluttered by the effects of C, making reuse across a different client impossible. To solve this problem, we generalize atomic summaries to support *pending asynchronous calls* (*pending asyncs* in short). Using pending asyncs, we can synchronize asynchrony internal to the service, while leaving the asynchronous callback to C as pending in the summary of S, thus enabling the reuse across different clients. Once the summary of S has been absorbed into the client, we need a mechanism to replace the pending async with the effect of the concrete implementation of C. For that we provide a second proof rule to eliminate pending asyncs from specifications.

We integrated our proof rules in the CIVL verifier [9] which provided a baseline framework of program layers. We report on our experience verifying a collection of benchmark programs, showing that our technique enables elegant specifications and proofs of asynchronous programs.

2 Overview

We start with an overview of our new verification technique based on the two concepts *synchronization* and *pending asyncs*. In our examples we follow the convention of writing procedure names capitalized (e.g., `Acquire`), and atomic action names in all caps (e.g., `ACQUIRE`). We use the notation `[...]` to denote unnamed atomic actions, i.e., the statements inside square brackets are considered to execute indivisibly.

2.1 Asynchronous Increments and Decrements

Consider the program in Figure 1 (a). The program comprises a single procedure `Main` that uses a global variable `x` and a local variable `i`. Every iteration of the while loop in `Main` creates two new threads, one executing an atomic increment `[x := x + 1]`, and one

(a)

```
global var x

proc Main(n):
  i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    i := i + 1
```

(b)

```
proc Main:
  async Foo
  assert false

proc Foo:
  call Foo
```

(c)

```
global var x

proc Main(n):
  i := 0
  while i < n:
    async [x := x + 1]
    async [x := x - 1]
    if *: i := i + 1
```

Figure 1 Asynchronous increments and decrements.

executing an atomic decrement `[x := x - 1]`. Due to asynchronous thread creation, the execution of individual increments and decrements can be interleaved arbitrarily. However, once all threads finish, the value in variable `x` is equal to its initial value. Thus, `Main` *refines* the atomic action `[skip]`, which does nothing.

A standard noninterference-based proof of this program requires an invariant that states that "`x` is equal to its original value, plus the number of finished increment threads, minus the number of finished decrement threads". Stating this invariant requires ghost code that tracks the progress of each thread. In contrast, our *synchronization* proof rule (Section 4) allows us to consider both asynchronous calls in `Main` as regular synchronous calls. Then sequential reasoning suffices to prove that the procedure leaves the variable `x` unchanged. Synchronization is justified by the commutativity of atomic actions on shared state. Specifically, both increment and decrement are *left movers* in the context of our program. Thus the asynchronous computation steps in an interleaved execution can be rearranged to obtain a corresponding synchronous execution that preserves final states.

However, commutativity alone is not sufficient! We also need to ensure that synchronization preserves failing behaviors. Consider the program in Figure 1 (b) where `Main` asynchronously calls a procedure `Foo` (which calls itself recursively) followed by a failing assertion. The program has failing executions; the assertion can be scheduled any time between steps of `Foo`. If we synchronize the call to `Foo`, however, the nontermination of `Foo` makes the assertion unreachable and thus synchronization must not be allowed. We could require termination of the synchronized program, but this would be unnecessarily restrictive. We propose a weaker condition called *cooperation*, which only requires the possibility to terminate. In other words, it must be impossible for the synchronized program to reach a state where nontermination is inevitable. To illustrate cooperation, consider Figure 1 (c), a modification of (a) which nondeterministically increments the loop counter `i`. The program does not terminate because it *may* loop forever, but it cooperates because it *can* always increment `i`. By synchronization we can show analogously to (a) that (c) also refines `[skip]`.

2.2 Lock Service

Figure 2 (a) shows a simple lock service implementation. A client requests the lock by asynchronously invoking `Acquire`, which is implemented as spinlock using the atomic compare-and-swap (CAS) operation on the global variable `l`. Once successful, the client of the lock service is notified via an asynchronous callback. Summarizing `Acquire` as atomic action via synchronization of the callback is not desirable, because it would drag in the effect of the client into the specification of `Acquire`. Instead, we propose the modular, reusable, and client-independent atomic action specifications `ACQUIRE` and `RELEASE` shown in (b). Notice how we represent guarded atomic transitions as program code. But more importantly, observe that the atomic action specification `ACQUIRE` contains a *pending async* to `Callback`. That is, we allow the effect of asynchronous thread creation as part of atomic actions. Now, to

(a)
```
global var l

proc Acquire (tid):
  b := false
  while !b:
    call b := CAS(l, nil, tid)
  async Callback(tid)

proc Release (tid):
  call [l := nil]
```

(b)
```
global var l

action ACQUIRE (tid):
  assert tid != nil
  assume l == nil
  l := tid
  async Callback(tid)

action RELEASE (tid):
  assert tid != nil && l == tid
  l := nil
```

(c)
```
global var x

proc Callback(tid):
  call [t := x]
  call [x := t + 1]
  async Release(tid)
```

(d)
```
global var x, l

action CALLBACK(tid):
  assert tid != nil
  assert l == tid
  x := x + 1
  l := nil
```

(e)
```
global var x, l

action ACQUIRE' (tid):
  assert tid != nil
  assume l == nil
  l := tid
  x := x + 1
  l := nil
```

(f)
```
global var x

action ACQUIRE'' (tid):
  x := x + 1
```

Figure 2 Lock service.

make use of such specifications, our technique is complemented with a proof rule to eliminate pending asyncs (Section 6), once an atomic action specification for the target is available. For example, consider the callback implementation in (c) that reads and writes a shared variable `x`, and then releases the lock. Since the callback is only supposed to be invoked with the lock held, we strengthen `[t := x]` and `[x := t + 1]` with the gate `assert tid != nil && l == tid`, which makes the operations commutative. Together with `RELEASE` being a left mover, we use synchronization to show that `Callback` refines the atomic action `CALLBACK` in (d). Now that we have an atomic action specification for `Callback`, we use it to eliminate the pending async in `ACQUIRE` and obtain the atomic action `ACQUIRE'` in (e). Notice how the gates of `CALLBACK` are discharged by the code preceding the pending async in `ACQUIRE`. Finally, we can abstract away the lock acquire and release, such that the client of the lock service only sees the atomic action `ACQUIRE''` in (f).

2.3 Layered Refinement Proofs

Our proof rules connect a lower-level, more fine-grained program with a higher-level, more coarse-grained program (both a bottom-up and top-down interpretation is possible), and repeated applications lead to a hierarchy of connected programs. However, due to the structure-preserving nature of our rules, in practice (Section 7) the programmer only writes a single program with *layer annotations* [12] that encode the program on multiple layers of abstraction. Our verifier automatically extracts the hierarchy of programs and generates the necessary verification conditions to justify their connection.

3 An Asynchronous Programming Language

In this section we define a core asynchronous programming language on which we formalize our verification technique, and recall the notion of mover types and reduction.

Variables and stores. Let $\mathcal{V}$ be a set of *variables* partitioned into *global variables* $\mathcal{V}_G$ and *local variables* $\mathcal{V}_L$, and $\mathcal{V}_R \subseteq \mathcal{V}_L$ is a set of *return variables*. A *store* is a mapping $\sigma : \mathcal{V} \to \mathcal{D}$ that assigns a *value* from a domain $\mathcal{D}$ to every variable. Similarly, $g : \mathcal{V}_G \to \mathcal{D}$ is a *global store* and $\ell : \mathcal{V}_L \to \mathcal{D}$ is a *local store*. Let $g{\cdot}\ell$ denote the combination of g and ℓ into a store. To model return values from a procedure with local store ℓ_1 to a caller procedure with local store ℓ_2, we define the resulting store at the caller as $\ell_1 \rhd \ell_2 = \lambda v.\,\text{if } v \in \mathcal{V}_R \text{ then } \ell_1(v) \text{ else } \ell_2(v)$.

$$(g, TC[\ell][\texttt{skip}; s] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][s] \uplus \mathcal{T}) \quad \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \alpha \quad \mathcal{T}' = \{(\ell'', \texttt{call } P) \mid (\ell'', P) \in \Omega\}}{(g, TC[\ell][\texttt{call } A] \uplus \mathcal{T}) \Rightarrow (g', TC[\ell'][\texttt{skip}] \uplus \mathcal{T}' \uplus \mathcal{T})} \quad \textsc{ActionStep}$$

$$\frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \notin \rho}{(g, TC[\ell][\texttt{call } A] \uplus \mathcal{T}) \Rightarrow \lightning} \quad \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g, TC[\ell][\texttt{if } le \texttt{ then } s_1 \texttt{ else } s_2] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][s'] \uplus \mathcal{T})} \quad \textsc{If}$$

$$(g, TC[\ell][\texttt{call } P] \uplus \mathcal{T}) \Rightarrow (g, (\ell, \mathcal{P}.P){\cdot}TC[\ell][\texttt{skip}] \uplus \mathcal{T}) \quad \textsc{Call}$$

$$(g, (\ell_1, \texttt{skip}){\cdot}TC[\ell_2][s] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell_1 \triangleright \ell_2][s] \uplus \mathcal{T}) \quad \textsc{Return}$$

$$(g, TC[\ell][\texttt{async } P] \uplus \mathcal{T}) \Rightarrow (g, TC[\ell][\texttt{skip}] \uplus (\ell, \texttt{call } P) \uplus \mathcal{T}) \quad \textsc{Async} \qquad (g, (\ell, \texttt{skip}) \uplus \mathcal{T}) \Rightarrow (g, \mathcal{T}) \quad \textsc{End}$$

Figure 3 Small-step operational semantics.

Atomic actions. We generalize gated actions introduced in [5] with the idea of pending asyncs. An *atomic action* is a pair (ρ, α), where the *gate* ρ is a set of stores and the *update* α is a set of *transitions* $(\sigma, \sigma', \Omega)$ where σ,σ' are stores and Ω is a finite multiset of *pending asyncs* (ℓ, P) consisting of a local store and a procedure name. If an atomic action is executed in a store σ with $\sigma \notin \rho$, the program "fails"; otherwise, if $\sigma \in \rho$, a transition $(\sigma, \sigma', \Omega) \in \alpha$ atomically updates the store to σ' and creates new threads according to Ω.

Atomic actions subsume many standard programming language statements. In particular, (nondeterministic) assignments, assertions, and assumptions. The table on the right shows some examples ranging over variables x and y.

command	gate	update
$x := x + y$	*true*	$x' = x + y \wedge y' = y$
havoc x	*true*	$y' = y$
assert $x < y$	$x < y$	$x' = x \wedge y' = y$
assume $x < y$	*true*	$x < y \wedge x' = x \wedge y' = y$

Syntax. A program $\mathcal{P}$ is a finite mapping from *atomic action names* A to atomic actions, and *procedure names* P to *statements* s of the form

$$s ::= \texttt{skip} \mid s; s \mid \texttt{if } le \texttt{ then } s \texttt{ else } s \mid \texttt{call } A \mid \texttt{call } P \mid \texttt{async } P.$$

A program contains a dedicated procedure *Main* that serves as an entry point for executions, and every atomic action name respectively procedure name appearing in a call statement must be properly mapped to an atomic action respectively statement. We will write $\mathcal{P}.A$ and $\mathcal{P}.P$ for $\mathcal{P}(A)$ and $\mathcal{P}(P)$, and $A, P \in \mathcal{P}$ for $A, P \in dom(\mathcal{P})$. We identify the conditional expression le with the set of local stores that satisfy it.

Semantics. A *frame* f is a pair (ℓ, s) of local store ℓ and statement s. A *thread* t is a sequence of frames $\vec{f}$, denoting a call stack. A *state* $(g, \mathcal{T})$ is a pair of global store g and a finite multiset of threads $\mathcal{T}$. By slight abuse of notation we will identify a thread t with the singleton multiset $\{t\}$, and thus write $\mathcal{T} \uplus t$ for adding t to $\mathcal{T}$. Let *statement contexts* SC, *frame contexts* FC, and *thread contexts* TC be defined as follows:

$$SC ::= \bullet_{Stmt} \mid SC; s \qquad FC ::= (\bullet_{LStore}, SC) \qquad TC ::= FC{\cdot}\vec{f}$$

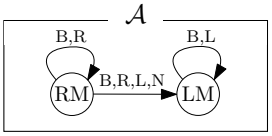
$TC[\ell][s]$ denotes the thread obtained by filling the two unique holes $\bullet_{Stmt}$ and $\bullet_{LStore}$ in TC with statement s and local store ℓ, respectively. Thus, $TC[\ell][s]$ executes s from ℓ as next step. The operational semantics is formalized in Figure 3 as a transition relation $\Rightarrow$ between states. An *execution* π is a sequence of states $x_0 \Rightarrow x_1 \Rightarrow \ldots$, and we write $\pi : x_0 \Rightarrow^* x_n$ to denote that π is an execution that starts in x_0 and ends in x_n.

Refinement. Given a program $\mathcal{P}$, we are interested in executions that start with a single thread executing *Main* from some initial store $\sigma = g{\cdot}\ell$, i.e., executions that start in a state $(g, (\ell, \texttt{call}\ \mathit{Main}))$. In particular, we are interested in executions that either fail or terminate. We define $\mathit{Bad}(\mathcal{P})$ to be the set of initial stores associated with *failing executions*, and $\mathit{Good}(\mathcal{P})$ to be the relation between initial and final stores associated with *terminating executions*:

$$\mathit{Bad}(\mathcal{P}) = \left\{g{\cdot}\ell \mid \left(g, (\ell, \texttt{call}\ \mathit{Main})\right) \Rightarrow^* \lightning\right\};\quad \mathit{Good}(\mathcal{P}) = \left\{(g{\cdot}\ell, g') \mid \left(g, (\ell, \texttt{call}\ \mathit{Main})\right) \Rightarrow^* (g', \varnothing)\right\}.$$

A program $\mathcal{P}_1$ *refines* a program $\mathcal{P}_2$, denoted $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, if (1) $\mathit{Bad}(\mathcal{P}_1) \subseteq \mathit{Bad}(\mathcal{P}_2)$ and (2) $\overline{\mathit{Bad}(\mathcal{P}_2)} \circ \mathit{Good}(\mathcal{P}_1) \subseteq \mathit{Good}(\mathcal{P}_2)$; $\overline{\cdot}$ is set complement, $\circ$ is relation composition. The first condition states that $\mathcal{P}_2$ has to preserve failing executions of $\mathcal{P}_1$. The second condition states that $\mathcal{P}_2$ has to preserve terminating executions of $\mathcal{P}_1$ for initial states that cannot fail. That is, $\mathcal{P}_2$ can fail more often than $\mathcal{P}_1$.

Reduction. Let M be a mapping from atomic action names to *mover types* [6]: B (*both mover*), L (*left mover*), R (*right mover*), N (*non-mover*). Intuitively, an atomic action is a right mover, if it commutes to the right (i.e., later in time) with respect to all other atomic actions in $\mathcal{P}$. A left mover is symmetric, and an atomic action can be both a left and right mover. Reduction has traditionally been applied to multithreaded programs to convert a sequence of atomic actions performed by a single thread into an atomic block. The sequence of mover types of the atomic actions in this block must be a valid run of the nondeterministic *atomicity automaton* $\mathcal{A}$ on the right. In this paper, we exploit and extend this work to synchronize asynchronous computation spanning multiple threads.



We define the predicate $\mathit{MoverValid}(\mathcal{P}, M)$ which holds whenever the atomic actions in $\mathcal{P}$ satisfy the mover types indicated by M. Formally, $\mathit{MoverValid}(\mathcal{P}, M)$ holds if for all $A_1, A_2 \in \mathcal{P}$ with $\mathcal{P}.A_1 = (\rho_1, \alpha_1)$ and $\mathcal{P}.A_2 = (\rho_2, \alpha_2)$, the following conditions hold (generalizing [10] to support pending asyncs).

- **Commutativity:** If $M(A_1) \in \{R, B\}$ or $M(A_2) \in \{L, B\}$, then the effect of executing A_1 followed by A_2 in two different threads can also be achieved by A_2 followed by A_1.

$$\begin{array}{l}\forall g, \bar{g}, g', \ell_1, \ell_1',\\ \quad \ell_2, \ell_2', \Omega_1, \Omega_2\\ \exists \hat{g}, \Omega_1', \Omega_2'\end{array} : \left(\begin{array}{ll} \wedge & g{\cdot}\ell_1 \in \rho_1\\ \wedge & g{\cdot}\ell_2 \in \rho_2\\ \wedge & (g{\cdot}\ell_1, \bar{g}{\cdot}\ell_1', \Omega_1) \in \alpha_1\\ \wedge & (\bar{g}{\cdot}\ell_2, g'{\cdot}\ell_2', \Omega_2) \in \alpha_2\end{array}\right) \implies \left(\begin{array}{ll} \wedge & (g{\cdot}\ell_2, \hat{g}{\cdot}\ell_2', \Omega_2') \in \alpha_2\\ \wedge & (\hat{g}{\cdot}\ell_1, g'{\cdot}\ell_1', \Omega_1') \in \alpha_1\\ \wedge & \Omega_1 \uplus \Omega_2 = \Omega_1' \uplus \Omega_2'\end{array}\right)$$

- **Forward preservation:** If $M(A_1) \in \{R, B\}$ or $M(A_2) \in \{L, B\}$, then the failure of A_2 after the execution of A_1 implies that A_2 must also fail before the execution of A_1.
  $\forall g, g', \ell_1, \ell_1', \ell_2, \Omega_1 : (g{\cdot}\ell_1 \in \rho_1 \wedge g{\cdot}\ell_2 \in \rho_2 \wedge (g{\cdot}\ell_1, g'{\cdot}\ell_1', \Omega_1) \in \alpha_1) \implies g'{\cdot}\ell_2 \in \rho_2$
- **Backward preservation:** If $M(A_2) \in \{L, B\}$, then the failure of A_1 before the execution of A_2 implies that A_1 must also fail after the execution of A_2. $\forall g, g', \ell_1, \ell_2, \ell_2', \Omega_2 :$ $(g{\cdot}\ell_2 \in \rho_2 \wedge (g{\cdot}\ell_2, g'{\cdot}\ell_2', \Omega_2) \in \alpha_2 \wedge g'{\cdot}\ell_1 \in \rho_1) \implies g{\cdot}\ell_1 \in \rho_1$
- **Nonblocking:** If $M(A_2) \in \{L, B\}$, then A_2 must be nonblocking. $\forall \sigma \in \rho_2\ \exists \sigma', \Omega :$ $(\sigma, \sigma', \Omega) \in \alpha_2$
- **Async freedom:** If $M(A_1) \in \{R, B\}$, then A_1 cannot have pending asynchronous calls.
  $\forall \sigma, \sigma', \Omega : \sigma \in \rho_1 \wedge (\sigma, \sigma', \Omega) \in \alpha_1 \implies \Omega = \varnothing$

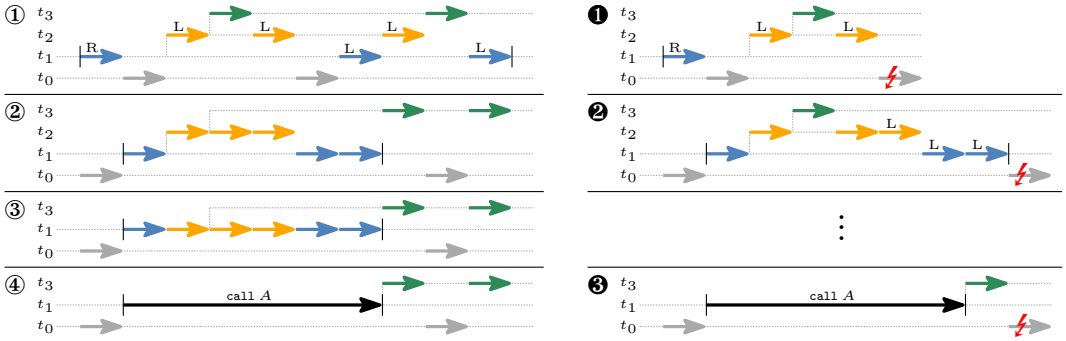


Figure 4 Synchronizing asynchronous executions.

4 Synchronizing Asynchrony

In this section, we formalize the synchronization proof rule which allows us to transform a procedure into an atomic action that summarizes asynchronous effects, either directly or via pending asyncs. Synchronization requires two technical innovations. First, we extend the *commutativity* conditions required for reduction to account for asynchronous thread creation. Second, we impose a new *cooperation* condition necessary for the soundness of our transformation.

Given a procedure Q, a mover typing M, and a set of procedures Σ to synchronize in Q (asynchronous calls to procedures not in Σ are treated as pending asyncs), the SYNCHRONIZE rule transforms procedure Q into an atomic action (ρ, α) with fresh name A:

SYNCHRONIZE

$$\frac{\mathit{MoverValid}(\mathcal{P}, M) \quad \mathit{Sync}(\mathcal{P}, M, Q, \Sigma) \quad \mathit{Refinement}(\mathcal{P}, Q, \Sigma, \rho, \alpha)}{\mathcal{P} \rightsquigarrow \mathcal{P}[Q \mapsto \texttt{call } A] \cup [A \mapsto (\rho, \alpha)]} \quad \begin{array}{l} Q \in \mathcal{P} \\ A \notin \mathcal{P} \end{array}$$

We already defined *MoverValid* in the previous section. Now we informally discuss the soundness of SYNCHRONIZE, and formally defined the other two premises *Sync* and *Refinement*. In the next section we show how all premises can be efficiently checked in practice.

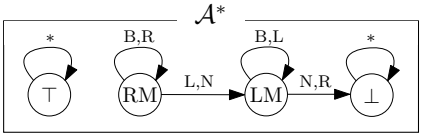
▶ **Theorem 1.** *If* $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ *using the* SYNCHRONIZE *rule, then* $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.

Intuition. The core idea of Theorem 1 is the rewriting of a $\mathcal{P}_1$-execution π_1 into an equivalent $\mathcal{P}_2$-execution π_2. Concretely, (1) if π_1 fails then π_2 must fail, and (2) if π_1 terminates then π_2 must either terminate with the same final state or fail. We illustrate this transformation in Figure 4. On the left, ① shows part of an asynchronous execution, initially comprising two threads t_0 and t_1. Thread t_1 executes the transformed procedure Q (the call and return are indicated with black bars), which makes an asynchronous call to spawn t_2, and t_2 asynchronously spawns t_3. Notice that t_2 terminates after three steps. We consider the procedure of t_2 to be in Σ (i.e., to be synchronized), while the procedure of t_3 is not in Σ (i.e., to be treated as pending async). Our goal is to transform execution ① into execution ②, which has the following properties: (1) Q executes without interruption from t_0, (2) t_2 terminates without interruption before t_1 continues, and (3) t_3 only starts after Q returns. To permit this transformation, *Sync* requires that Q, *including* asynchronous calls to procedures in Σ, executes a sequence of right movers, followed by at most one non-mover, followed by a sequence of left movers. Furthermore, the asynchronous calls to procedures in Σ must only execute left movers. The steps of t_1 and t_2 in ① are labeled with mover types that satisfy

this conditions. When moving the right mover to the right and the left movers to the left to obtain ②, the commutativity, forward preservation, and backward preservation properties of *MoverValid* guarantee that the executions stay equivalent. Now, as shown in ③, the steps of t_2 can be considered to execute synchronously in its parent t_1. Finally, *Refinement* ensures that the synchronized behavior of Q is summarized by the atomic action A in ④, which captures the creation of t_3 as pending async.

On the right of Figure 4, ❶ shows an execution where Q started, but then t_0 failed. Notice that, if all steps of Q before the failure are right movers, these steps can be removed from the execution (by moving them to the right, "past" the failure), and the failure occurs before Q even starts. In ❶, however, Q already executed a left mover. Even if we move the steps of Q together, the partial execution of Q is not summarized by A. However, we know that only left movers can follow in t_1 and t_2. Since left movers are non-blocking and backward preserving, they can be inserted at the end of the execution, right before the failure. The *cooperation* condition (part of *Sync*) ensures that this can be done so that Q is completed, as shown in ❷. Then we can again arrive at an execution where Q is replaced by A (see ❸).

Concurrent tracking semantics. The execution in ① is labeled with mover types that allowed us to rearrange the steps of Q to obtain the execution in ②. To characterize the executions for which such a rearrangement is possible in general, we define the *concurrent tracking semantics* $\xrightarrow{M,Q,\Sigma}$ (Figure 5) that is similar to $\Rightarrow$, except that we additionally track a *mover phase* m in frames, which is one of the states of the *tracking automaton* $\mathcal{A}^*$ on the right: $\top$ (*no tracking*), RM (*right-mover phase*), LM (*left-mover phase*), $\bot$ (*violation*). Call transitions from $\top$ to RM on a top-level call to Q, or otherwise propagates the mover phase of the caller to the callee. Conversely, Return transitions back to $\top$ when returning from a top-level call to Q, or otherwise propagates the mover phase of the callee to the caller. ActionStep follows a transition in $\mathcal{A}^*$ according to the mover type of the invoked atomic action. In particular, if we are tracking ($m \neq \top$), we stay in RM until a non-right mover (L or N) causes a transition to LM. In LM only left movers should follow, and thus the occurrence of a non-left mover (N or R) causes a transition to the violation state $\bot$. Notice that the async freedom condition of *MoverValid* forces a thread that executes an atomic action with pending asyncs to LM. This is important to ensure that only left movers can follow, which can be moved before the steps of any pending async. Similarly, Async transitions the parent thread of an asynchronous call to LM. The child thread is set to LM if we want to synchronize the call, otherwise it is not tracked. In both ActionStep and Async, if an untracked child thread executes `call` Q, the subsequent application of Call will start to track the child tread separately.



Sequential synchronized semantics. In ③ we are concerned with the sequential execution of Q, with asynchronous calls to procedures in Σ being synchronized. We formally define the *sequential synchronized semantics* $\xrightarrow{\Sigma}$ (Figure 5) that executes a single thread and stores a multiset of pending asyncs. In ActionStep, the pending asyncs of an atomic action are added to the already existing pending asyncs. For an asynchronous call to P, Async records a pending thread creation if $P \notin \Sigma$, and synchronizes the call if $P \in \Sigma$. The synchronized stack frame is marked with $\sharp$ such that it is popped in AsyncReturn without writing return variables to the caller. This technicality is necessary in our formalization. In practice, asynchronously called procedures simply cannot have return parameters.

$$\xrightarrow{M,Q,\Sigma}$$

$$(g, TC[\ell][\texttt{skip}; s][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, TC[\ell][s][m] \uplus \mathcal{T}) \ \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \alpha \quad m' = \mathcal{A}^*(m, M(A)) \quad \mathcal{T}' = \{(\ell'', \texttt{call}\ P, \top) \mid (\ell'', P) \in \Omega\}}{(g, TC[\ell][\texttt{call}\ A][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g', TC[\ell'][\texttt{skip}][m'] \uplus \mathcal{T}' \uplus \mathcal{T})} \ \textsc{ActionStep}$$

$$\frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \notin \rho}{(g, TC[\ell][\texttt{call}\ A][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} \lightning} \ \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g, TC[\ell][\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, TC[\ell][s'][m] \uplus \mathcal{T})} \ \textsc{If}$$

$$\frac{m' = \text{if } (m = \top \wedge P = Q) \text{ then RM else } m}{(g, TC[\ell][\texttt{call}\ P][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, (\ell, \mathcal{P}.P, m'){\cdot}TC[\ell][\texttt{skip}][m] \uplus \mathcal{T})} \ \textsc{Call}$$

$$\frac{m' = \text{if } (m_2 = \top) \text{ then } \top \text{ else } m_1}{(g, (\ell_1, \texttt{skip}, m_1){\cdot}TC[\ell_2][s][m_2] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, TC[\ell_1 \rhd \ell_2][s][m'] \uplus \mathcal{T})} \ \textsc{Return}$$

$$\frac{m' = \text{if } (m \neq \top) \text{ then LM else } \top \quad m'' = \text{if } (m \neq \top \wedge P \in \Sigma) \text{ then LM else } \top}{(g, TC[\ell][\texttt{async}\ P][m] \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, TC[\ell][\texttt{skip}][m'] \uplus (\ell, \texttt{call}\ P, m'') \uplus \mathcal{T})} \ \textsc{Async}$$

$$(g, (\ell, \texttt{skip}, m) \uplus \mathcal{T}) \xrightarrow{M,Q,\Sigma} (g, \mathcal{T}) \ \textsc{End}$$

$$\xrightarrow{\Sigma}$$

$$(g, TC[\ell][\texttt{skip}; s], \Omega) \xrightarrow{\Sigma} (g, TC[\ell][s], \Omega) \ \textsc{Seq}$$

$$\frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \in \rho \quad (g{\cdot}\ell, g'{\cdot}\ell', \Omega) \in \alpha}{(g, TC[\ell][\texttt{call}\ A], \Omega') \xrightarrow{\Sigma} (g', TC[\ell'][\texttt{skip}], \Omega \uplus \Omega')} \ \textsc{ActionStep} \qquad \frac{\mathcal{P}.A = (\rho, \alpha) \quad g{\cdot}\ell \notin \rho}{(g, TC[\ell][\texttt{call}\ A], \Omega) \xrightarrow{\Sigma} \lightning} \ \textsc{ActionFail}$$

$$\frac{s' = \text{if } (\ell \in le) \text{ then } s_1 \text{ else } s_2}{(g, TC[\ell][\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2], \Omega) \xrightarrow{\Sigma} (g, TC[\ell][s'], \Omega)} \ \textsc{If}$$

$$(g, TC[\ell][\texttt{call}\ P], \Omega) \xrightarrow{\Sigma} (g, (\ell, \mathcal{P}.P){\cdot}TC[\ell][\texttt{skip}], \Omega) \ \textsc{Call}$$

$$(g, (\ell_1, \texttt{skip}){\cdot}TC[\ell_2][s], \Omega) \xrightarrow{\Sigma} (g, TC[\ell_1 \rhd \ell_2][s], \Omega) \ \textsc{Return}$$

$$(g, TC[\ell][\texttt{async}\ P], \Omega) \xrightarrow{\Sigma} \begin{cases} (g, TC[\ell][\texttt{skip}], (\ell, P) \uplus \Omega) & \text{if } P \notin \Sigma \\ (g, (\ell, \texttt{call}\ P)^\sharp{\cdot}TC[\ell][\texttt{skip}], \Omega) & \text{if } P \in \Sigma \end{cases} \ \textsc{Async}$$

$$(g, (\ell, \texttt{skip})^\sharp{\cdot}\vec{f}, \Omega) \xrightarrow{\Sigma} (g, \vec{f}, \Omega) \ \textsc{AsyncReturn}$$

Figure 5 Concurrent tracking semantics $\xrightarrow{M,Q,\Sigma}$ and sequential synchronized semantics $\xrightarrow{\Sigma}$.

With the concurrent tracking semantics and the sequential synchronized semantics we can now formally define *Sync* and *Refinement*.

Sync. $\mathit{Sync}(\mathcal{P}, M, Q, \Sigma)$ comprises the following two conditions:

S1 $(g, (\ell, \texttt{call}\ \mathit{Main}, \top)) \xrightarrow{M,Q,\Sigma}{}^* (g', TC[\ell'][s][m] \uplus \mathcal{T})$ implies $m \neq \bot$;

S2 $(g, (\ell, \texttt{call}\ \mathit{Main}, \top)) \xrightarrow{M,Q,\Sigma}{}^* (g', TC[\ell'][\texttt{call}\ P][\text{LM}] \uplus \mathcal{T})$ implies $(g', (\ell', \texttt{call}\ P), \varnothing) \xrightarrow{\Sigma}{}^* (g'', (\ell'', \texttt{skip}), \Omega'')$.

S1 states that executions respect the required mover sequences, i.e., no violation is reachable in the tracking semantics. S2 (the cooperation condition) states that every procedure call in the left-mover phase can be completed. The repeated application of S1 allows us to complete partial executions of Q. Note that S2 also captures asynchronous calls to procedures P with $P \in \Sigma$, since the operational semantics rewrites `async` P into `call` P.

Refinement. $\mathit{Refinement}(\mathcal{P}, Q, \Sigma, \rho, \alpha)$ comprises the following two conditions:

R1 $\rho \cap \{g{\cdot}\ell \mid (g, (\ell, \mathcal{P}.Q), \varnothing) \xrightarrow{\Sigma}{}^{*} \lightning\} = \varnothing$;

R2 $\rho \circ \{(g{\cdot}\ell, g'{\cdot}\ell', \Omega) \mid (g, (\ell, \mathcal{P}.Q), \varnothing) \xrightarrow{\Sigma}{}^{*} (g', (\ell', \texttt{skip}), \Omega)\} \subseteq \alpha$.

R1 states that the gate of A is strong enough to filter out all failures of Q, and R2 states that the transition relation of A captures all non-failing executions of Q.

5 Verifying Synchronization

In this section we show how the premises of the SYNCHRONIZE rule can be efficiently checked in practice. The *MoverValid* and *Refinement* premises both lead to standard verification conditions. In particular, the constraints of *MoverValid* state the commutativity of individual atomic actions, and the constraints of *Refinement* state that a sequential procedure is summarized by a transition relation, which can be readily handed off to logical reasoning engines. Thus we focus on *Sync* which we decompose as follows:

$$\frac{\mathit{StaticSync}(\mathcal{P}, M, Q, \Sigma, \mathit{Pre}) \quad \mathit{Safe}(\mathcal{P}, \mathit{Pre}) \quad \mathit{Terminates}(\mathcal{P}, \Sigma, \mathit{Pre}, \mathit{Red})}{\mathit{Sync}(\mathcal{P}, M, Q, \Sigma)}$$

We establish *Sync* in three steps. First, *StaticSync* is a static control-flow analysis that over-approximates the tracking semantics. It uses the domain of a *precondition mapping* *Pre*, a partial mapping from procedure names to sets of stores. If *StaticSync* succeeds, it guarantees S1 (i.e., that $\bot$ cannot be reached) and that all procedures P called with mover phase LM in S2 are in $\mathit{dom}(\mathit{Pre})$. Second, we over-approximate the possible stores $g'{\cdot}\ell'$ at these calls. For that, *Safe* requires that *Pre* denotes valid preconditions, i.e., if `call` P is reachable with store $g'{\cdot}\ell'$, then $g'{\cdot}\ell' \in \mathit{Pre}(P)$ for all $P \in \mathit{dom}(\mathit{Pre})$. Then finally, to establish S2, it remains to show that there is some terminating sequential execution from $(g', (\ell', \texttt{call}\ P), \varnothing)$ for every $P \in \mathit{dom}(\mathit{Pre})$ and $g'{\cdot}\ell' \in \mathit{Pre}(P)$. *Terminates* reduces these cooperation checks to standard termination checks on a restricted program. In particular, the *restriction function* *Red* limits the nondeterministic behavior of some atomic actions. Then showing that *all* executions in the restricted program terminate implies that there is *some* terminating execution in the original program (given that *Red* is not allowed to make atomic actions blocking).

StaticSync. Let $\mathcal{E}$ be the function that maps a mover type to the corresponding set of edges in $\mathcal{A}$, e.g., $\mathcal{E}(\mathrm{R}) = \{\mathrm{RM} \to \mathrm{RM}, \mathrm{RM} \to \mathrm{LM}\}$. We define an interprocedural control flow analysis that lifts $\mathcal{E}$ to a mapping $\widehat{\mathcal{E}}$ on statements, corresponding to the paths a statement may take in the tracking semantics. We write $\mathit{StaticSync}(\mathcal{P}, M, Q, \Sigma, \mathit{Pre})$ if there is a solution $\widehat{\mathcal{E}}(\mathcal{P}.Q) \neq \varnothing$ to the following equations w.r.t. M, Σ and Pre:

$$\widehat{\mathcal{E}}(\texttt{skip}) = \mathcal{E}(\mathrm{B})$$
$$\widehat{\mathcal{E}}(\texttt{call}\ A) = \mathcal{E}(M(A))$$
$$\widehat{\mathcal{E}}(s_1; s_2) = \widehat{\mathcal{E}}(s_1) \circ \widehat{\mathcal{E}}(s_2)$$
$$\widehat{\mathcal{E}}(\texttt{if}\ \mathit{le}\ \texttt{then}\ s_1\ \texttt{else}\ s_2) = \widehat{\mathcal{E}}(s_1) \cap \widehat{\mathcal{E}}(s_2)$$

$$\widehat{\mathcal{E}}(\texttt{call}\ P) = \begin{cases} \widehat{\mathcal{E}}(\mathcal{P}.P) & \text{if } P \in \mathit{dom}(\mathit{Pre}) \\ \widehat{\mathcal{E}}(\mathcal{P}.P) \cap \{\mathrm{RM}\}^2 & \text{if } P \notin \mathit{dom}(\mathit{Pre}) \end{cases}$$

$$\widehat{\mathcal{E}}(\texttt{async}\ P) = \begin{cases} \{\mathrm{LM}\}^2 & \text{if } P \notin \Sigma \\ \{\mathrm{LM}\}^2 \cap \widehat{\mathcal{E}}(\mathcal{P}.P) & \text{if } P \in \Sigma \cap \mathit{dom}(\mathit{Pre}) \\ \varnothing & \text{if } P \in \Sigma \setminus \mathit{dom}(\mathit{Pre}) \end{cases}$$

The equations on the left capture regular control-flow propagation. The equation for `call` P has two cases. If $P \in \mathit{dom}(\mathit{Pre})$, we do not restrict the call since P is cooperative. However, if $P \notin \mathit{dom}(\mathit{Pre})$ we must restrict the call to stay in the right-mover phase, because we cannot rely on the cooperation condition to complete partial executions of Q. The equation for `async` P has three cases. If $P \notin \Sigma$, we do not synchronize P and thus only require the

caller to be followed by only left movers. If $P \in \Sigma \cap dom(Pre)$, we additionally require the invoked procedure P to be only left movers. For synchronized procedures we always have to establish cooperation, thus the case $P \in \Sigma \setminus dom(Pre)$ is not allowed.

If $StaticSync(\mathcal{P}, M, Q, \Sigma, Pre)$, then S1 holds and for every `call` P reachable with LM in S2 we have $P \in dom(Pre)$. Hence, we must check cooperation for all procedures in $dom(Pre)$.

Safe. Now that we know the procedures that need to be checked for cooperation, we want to know the stores from which to check cooperation. For that, Pre must denote valid preconditions. We write $Safe(\mathcal{P}, Pre)$, if $(g, (\ell, \texttt{call } Main)) \Rightarrow^* (g', TC[\ell'][\texttt{call } P] \uplus \mathcal{T})$ implies $g' \cdot \ell' \in Pre(P)$ for all $P \in dom(Pre)$.

Terminates. Finally, we establish S2 by showing that all procedures P in $dom(Pre)$ cooperate from states in $Pre(P)$. Suppose that cooperation holds, but termination (which is stronger) does not. Such a difference between termination and cooperation must be due to nondeterminism. Thus, if we suitably restrict the nondeterminism to eliminate nonterminating behaviors, proving termination for the restricted program implies cooperation for the original program. Formally, a *restriction function* Red is a partial mapping from atomic action names to atomic actions, such that for all $A \in dom(Red)$ with $\mathcal{P}.A = (\rho, \alpha)$ it holds that $Red(A) = (\rho, \alpha')$ with $\alpha' \subseteq \alpha$ and $Red(A)$ is nonblocking. Let $\mathcal{P}^{Red}$ be the program equal to $\mathcal{P}$, except that $\mathcal{P}^{Red}.A = Red(A)$ for $A \in dom(Red)$.

We write $Terminates(\mathcal{P}, \Sigma, Pre, Red)$, if for all $P \in dom(Pre)$ and $g \cdot \ell \in Pre(P)$, there is no infinite sequential synchronized $\mathcal{P}^{Red}$-execution $(g, (\ell, \texttt{call } P), \varnothing) \xrightarrow{\Sigma} \cdots$. Notice that these termination checks can now be solved by a standard termination checker for sequential programs. While it is possible for the programmer to explicitly provide restricted atomic actions, in practice we did not found this necessary for any of our examples. Instead, a fixed policy to resolve nondeterministic branching (e.g., always take the *then* branch) was enough. For example, recall the program in Figure 1 (c). Always taking the *then* branch (i.e., resolving the nondeterministic choice to *true*) allows us to prove termination and thus implies cooperation of the original program.

▶ **Theorem 2.** *If we have* $StaticSync(\mathcal{P}, M, Q, \Sigma, Pre)$, $Safe(\mathcal{P}, Pre)$, *and* $Terminates(\mathcal{P}, \Sigma, Pre, Red)$, *then* $Sync(\mathcal{P}, M, Q, \Sigma)$ *holds.*

6 Eliminating Pending Asynchrony

In the previous two sections we showed how the Synchronize rule allows to summarize procedures to atomic actions, by either directly synchronizing asynchronous calls or keeping them as pending asyncs. In this section we present the complementary PendingAsyncElim rule to eliminate pending asyncs from atomic actions.

Let A be an atomic action with pending asyncs to a procedure P. Eliminating those pending asyncs requires that (1) P is summarized to an atomic action, say B, and (2) B must be a left mover, since we will directly compose its effect with A. Now we show the construction of the new gate and update for A. The new gate is obtained by filtering out all states from the gate of A that can cause B to fail. In other words, we strengthen A's gate such that it cannot make a transition to a state where B fails:

$$Gt(\rho_A, \alpha_A, \rho_B, P) = \left\{ \sigma \in \rho_A \mid \forall \begin{array}{c} g', \ell', \\ \ell_P, \Omega \end{array} : \begin{array}{c} (\sigma, g' \cdot \ell', (\ell_P, P) \uplus \Omega) \in \alpha_A \\ \implies g' \cdot \ell_P \in \rho_B \end{array} \right\}$$

The new update consists of two parts. First, we take all transitions without pending asyncs to P:

$$Upd_1(\alpha_A, P) = \{(\sigma, \sigma', \Omega) \in \alpha_A \mid \neg\exists \ell_P : (\ell_P, P) \in \Omega\}$$

Second, we compose all transitions with a pending async to P with the transitions of B:

$$Upd_2(\alpha_A, \alpha_B, P) = \left\{ (\sigma, g''\cdot\ell', \Omega \uplus \Omega') \mid \exists \begin{array}{c} g', \ell_P, \\ \Omega, \ell'' \end{array} : \wedge \begin{array}{l} (\sigma, g'\cdot\ell', (\ell_P, P) \uplus \Omega) \in \alpha_A \\ (g'\cdot\ell_P, g''\cdot\ell'', \Omega') \in \alpha_B \end{array} \right\}$$

Notice that the transitions of B can have pending asyncs that are absorbed into the resulting transition. Combining all pieces, we obtain the following rule for eliminating pending asyncs:

$$\boxed{\textsc{PendingAsyncElim}}$$
$$\frac{\begin{array}{ccc} \mathcal{P}.P = \texttt{call}\ B & \mathcal{P}.B = (\rho_B, \alpha_B) & M(B) \in \{\mathrm{L}, \mathrm{B}\} \\ \rho'_A = Gt(\rho_A, \alpha_A, \rho_B, P) & \alpha'_A = Upd_1(\alpha_A, P) \cup Upd_2(\alpha_A, \alpha_B, P) & \end{array}}{\mathcal{P} \uplus [A \mapsto (\rho_A, \alpha_A)] \rightsquigarrow \mathcal{P} \uplus [A \mapsto (\rho'_A, \alpha'_A)]}$$

▶ **Example 3.** Recall our motivating lock service example from Section 2.2. Eliminating the pending async in `ACQUIRE` is a formal application of PendingAsyncElim with $P =$ `Callback`, $A =$ `ACQUIRE`, and $B =$ `CALLBACK`. The resulting action (the new A) is `ACQUIRE'`.

▶ **Theorem 4.** *If* $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ *using the* PendingAsyncElim *rule, then* $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.

PendingAsyncElim eliminates a single pending async to P in A. Iterative application of the rule allows us to eliminate finitely many pending asyncs. In theory, PendingAsyncElim can be generalized with an induction schema to eliminate unboundedly many pending asyncs, but we did not find this necessary in practice.

7 Evaluation

We implemented our verification method in CIVL [9], a verification system for concurrent programs based on automated and modular refinement reasoning. In CIVL, a program is specified and verified across multiple layers of refinement. At each layer, procedures can be declared to refine atomic actions and henceforth appear atomic to higher layers. This means that an input program with layer annotations implicitly describes the program at multiple levels of abstraction, and CIVL automatically checks refinement between programs on adjacent layers.

We implemented and verified a collection of nine benchmarks, of which five expand on our motivating example from Section 2.1, one is a ping-pong agreement protocol that exercises the notion of cooperation, and the remaining three examples are discussed in the remainder of this section to illustrate (1) the interaction with CIVL and modular verification via pending asyncs, (2) the applicability to challenging concurrency, and (3) one-shot synchronization of nested asynchronous calls. Overall, our benchmarks capture realistic patterns of asynchronous computation. All benchmarks are verified by our tool in less than three seconds. The implementation and benchmarks are available at `https://github.com/boogie-org/boogie`.

The proof rules introduced in this paper are crucial to preserving the layered verification approach in CIVL and exploiting it to construct compact and highly-automated proofs with simple invariants [12]. Without our new rules, CIVL proofs of our benchmarks would amount to single-layer proofs with monolithic invariants in a style similar to classical proofs of distributed systems in modeling frameworks such as TLA+ [13].

```
action {:atomic}{:layer 1,1}
CAS_l (oldval:Tid, newval:Tid) returns (b:bool) {
  if   (l == oldval) { l := newval; b := true;  }
  else               {              b := false; }
}

procedure {:layer 1}{:refines ACQUIRE}
Acquire (tid:Tid) {
  var b:bool;
  b := false;
  while (!b) call b := CAS_l(nil, tid);
  async call Callback(tid);
}
```

```
action {:atomic}{:layer 2,3} ACQUIRE (tid:Tid) {
  assert tid != nil;
  assume l == nil;
  l := tid;
  async call Callback(tid);
}

procedure {:layer 2}{:refines CALLBACK}
Callback (tid:Tid) { /* not shown */ }

action {:left}{:layer 3} CALLBACK (tid:Tid) {
  assert tid != nil && l == tid;
  x := x + 1;
  l := nil;
}
```

Figure 6 Lock service in CIVL (excerpt).

7.1 Lock Service

In this section we illustrate how synchronization and pending async elimination are offered to a programmer in CIVL by revisiting the lock service example from Section 2.2.

Figure 6 shows a fragment of our CIVL implementation. First, let us understand the layer annotations in more detail. A procedure has a single layer number x that denotes the layer at which the procedure is shown to refine an atomic action. At all layers up to x calls to the procedure behave according to its implementation, and at layers higher than x calls to the procedure behave according to its refined atomic action. Atomic actions have an associated layer range $[x, y]$, which denotes at which layers the action is "available". For each layer, the set of available atomic actions is subject to pairwise commutativity checks. In Figure 6, the procedure `Acquire` is declared to refine the atomic action `ACQUIRE` at layer 1, which causes CIVL to apply synchronization. The implementation makes two calls, a synchronous call to a compare-and-swap operation which is already atomic at layer 1, and an asynchronous call to `Callback`. Since `Callback` is refined at the higher layer 2, the asynchronous call results in a pending async in the atomic action `ACQUIRE`. Thus, at layer 2, `ACQUIRE` is exactly the client-independent specification of `Acquire` we presented in Figure 2 (b).

Now `Callback` (whose implementation is not shown) is declared to refine `CALLBACK` at layer 2. This causes CIVL to apply pending async elimination in `ACQUIRE` at layer 3; the pending async to `Callback` is replaced with the effect of `CALLBACK`. Thus, at layer 3, `ACQUIRE` corresponds to `ACQUIRE'` in Figure 2 (e).

This example illustrates two important aspects of our proof method and its integration into CIVL. First, on the conceptual side, our method enables independent and modular reasoning about the lock service implementation and its client. The atomic action `ACQUIRE` can be (1) proved for a different implementation of the lock without the need to re-verify the client, and (2) used to reason about a different client by letting CIVL apply pending async elimination for a different client (i.e., `Callback` implementation). Second, on the practical side, the application of synchronization and pending async elimination in CIVL is driven by layer annotations. The programmer does not have to explicitly write the program under consideration at every layer of abstraction and specify the transformation that connects them. Instead, CIVL automatically constructs per-layer versions of procedures and atomic actions.

7.2 Two-phase Commit

In this section we show that our method applies to realistic programs with intricate concurrency by verifying full functional correctness of the two-phase commit (2PC) protocol. The protocol employs a *coordinator* process to consistently replicate transactions among a set of

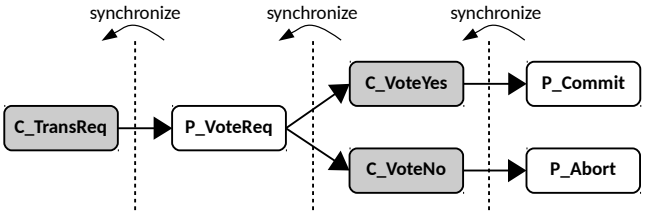


Figure 7 2PC call hierarchy (from left to right) and proof outline (right to left).

participant processes. In the first phase, the coordinator broadcasts incoming request to all participants, which respond either with a "yes" vote to commit, or a "no" vote to abort. In the second phase, the coordinator processes incoming votes as follows: (1) If *all* participants voted "yes" it broadcasts a "commit" message, or (2) as soon as *a single* participant votes "no" it broadcasts an "abort" message. Due to asynchrony and message reordering, the protocol implementation must be robust against unexpected situations. For example, a participant can receive an abort message before it receives the corresponding vote request.

Figure 7 shows the message handlers of the protocol we implemented in CIVL, together with the asynchronous communication structure. For example, `P_VoteReq` is a participants handler for vote requests, which asynchronously invokes either the coordinators `C_VoteYes` or `C_VoteNo` handler. To reason about the protocol, we use a variable `state` such that for every transaction `xid` and process `pid`, `state[xid][pid]` is one of `INIT`, `COMMIT`, or `ABORT`. We prove a top-level atomic action specification for `C_TransReq` that states that for a fresh `xid`, `state[xid]` is consistently updated, i.e., there are no two processes such that one is `COMMIT` and the other one `ABORT`. Figure 7 also shows the proof outline, making repeated use of synchronization. Here we focus on the first synchronization of `P_Commit` and `P_Abort`, which requires them to be left movers. A priori these operations do not commute, because they write the conflicting values `COMMIT` and `ABORT` to `state[xid][pid]`, respectively. However, by making it explicit that the coordinator has to decide on a transaction first, the following abstractions are commutative:

```
action P_Commit (pid,xid):
  assert state[xid][C] == COMMIT
  state[xid][pid] := COMMIT
```

```
action P_Abort (pid,xid):
  assert state[xid][C] == ABORT
  state[xid][pid] := ABORT
```

Our proof of 2PC confirms that the benefit of reduced invariant complexity in structured multi-layer refinement proofs [9] carries over to the asynchronous setting. In particular, we could state the central correctness invariant in terms of the protocol mechanism (i.e., voting and phases) after hiding low-level implementation details (i.e., counting).

7.3 Task Distribution Service

Finally, we verified a task distribution service inspired by a set of benchmarks from [1]. This example captures a whole class of similar benchmarks, where a set of *independent* tasks is processed by passing through a sequence of stages. The result of every stage is asynchronously communicated to the next stage, and different tasks can run through different stages. However, concurrent tasks do not interfere with each other. With this key difference to examples like 2PC, we can avoid the overhead of stepwise synchronization over several layers. Instead, synchronization can be applied to eliminate long (and even unbounded) chains of asynchronous calls in a single layer.

To summarize, synchronization is applicable to tightly interfering programs using program layers, and less interference leads to even simpler proofs.

8 Related Work

The idea of taming concurrency through synchrony is also at the heart of other works. Brisk [1] computes canonical sequentializations of message-passing programs by matching sends with corresponding receives. Our work differs in the programming model (dynamic thread creation vs. parametric processes with blocking receives) and the verification goal (deductive functional correctness vs. automatic deadlock-freedom). The work in [2] proposes the notion of robustness against concurrency as correctness condition for a class of event-driven programs. That is, the sequential behavior of a program is the underlying specification, and asynchronous executions are checked to conform to sequential executions. In contrast, we use synchronization to simplify the verification of safety properties.

There are several recent papers on mechanized verification of distributed systems. IronFleet [8] embeds TLA-style state-machine modeling [13] into the Dafny verifier [14] to refine high-level distributed systems specifications into low-level executable implementations. They use a fixed 3-layer design and one-shot reductions to atomic actions, while our program layers are more flexible. Ivy [18] organizes the search for an inductive invariant as a collaborative process between automatic verification attempts and user guided generalizations of counterexamples to induction in a graphical model. They use a restricted modeling and specification language that makes their verification conditions decidable. We rely on small partitioned verification conditions that can be discharged by an SMT solver [3]. PSync [4] uses a synchronous round-based model of communication for the purpose of program design and verification, shifting the complexity of efficient asynchronous execution to a runtime system. We allow explicit control over low-level details at the potential cost of increased verification effort. Verdi [21] lets the programmer provide a specification, implementation, and proof of a distributed system under an idealized network model. Then the application is automatically transformed into one that handles faults via verified system transformers. The rely-guarantee rule of [7] and the ALS types of [11] target a weaker form of asynchrony, where a single task queue atomically executes one task at a time.

Concurrent separation logic (CSL) [16] was devised for modular reasoning about multi-threaded shared-memory programs, focusing on the verification of fine-grained concurrent data structures. CSL adequately addresses the problem of reasoning about low-level concurrency related to dynamic memory allocation, but still suffers from the complications of a monolithic approach to invariant discovery for protocol-level concurrency. Recently, CSL has been applied to message-passing programs. The approach in [17] uses CSL to link implementation steps to atomic actions, and then relies on a model checker to explore the interleavings of those atomic actions. The work in [19] addresses the composition of verified protocols using ideas from separation logic. The actor services of [20] focus on compositional verification of response properties of message-passing programs.

9 Conclusion

The contribution of this paper are proof rules to simplify the reasoning about asynchronous concurrent programs. The impact of our work must be understood in the context of our two-pronged strategy for aiding interactive and automated verification of asynchronous programs. First, our proof rules enable asynchronous computation to be summarized analogous to the summarization of synchronous computation by pre- and post-conditions. This capability enables the construction of syntax-driven and structured proofs of asynchronous programs. Second, the program simplification enabled by our proof rules attacks the nemesis of complex invariants induced by a large number of interleaved executions. Instead of writing a large

and complex invariant justifying the overall correctness of the program, the programmer may now write a sequence of simpler invariants, each justifying a program simplification.

Our proof method decomposes the task of proving the correctness of a large asynchronous program into formulating and automatically discharging smaller independent proof obligations. These proof obligations show that an atomic action commutes with other atomic actions; that an atomic action summarizes the effect of a statement in a given context; and that an assertion is an inductive invariant for a simpler program, where asynchronous procedure calls are replaced by synchronous (immediate) atomic actions. Using our method, the automatable part of a concurrent verification problem – i.e., the safety proof given an inductive invariant – remains automatable, and the creative part – i.e., the discovery of an appropriate invariant – is greatly simplified by structuring it into smaller proof obligations, each of which can still be discharged automatically.

References

1 Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. In *OOPSLA*, 2017. doi:10.1145/3133934.

2 Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. Verifying robustness of event-driven asynchronous programs against concurrency. In *ESOP*, 2017. doi:10.1007/978-3-662-54434-1_7.

3 Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008. doi:10.1007/978-3-540-78800-3_24.

4 Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, 2016. doi:10.1145/2837614.2837650.

5 Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, 2009. doi:10.1145/1480881.1480885.

6 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, 2003. doi:10.1145/781131.781169.

7 Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *CONCUR*, 2015. doi:10.4230/LIPIcs.CONCUR.2015.483.

8 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, 2015. doi:10.1145/2815400.2815428.

9 Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV*, 2015. doi:10.1007/978-3-319-21668-3_26.

10 Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. Technical Report MSR-TR-2015-8, Microsoft Research, February 2015. URL: https://www.microsoft.com/en-us/research/publication/automated-and-modular-refinement-reasoning-for-concurrent-programs/.

11 Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. Asynchronous liquid separation types. In *ECOOP*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.396.

12 Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In *CAV*, 2018. doi:10.1007/978-3-319-96145-3_5.

13 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

14 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010. doi:10.1007/978-3-642-17511-4_20.

15 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975. doi:10.1145/361227.361234.

16 Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.

17 Wytse Oortwijn, Stefan Blom, and Marieke Huisman. Future-based static analysis of message passing programs. In *PLACES*, 2016. doi:10.4204/EPTCS.211.7.

18 Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, 2016. doi:10.1145/2908080.2908118.

19 Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. In *POPL*, 2018. doi:10.1145/3158116.

20 Alexander J. Summers and Peter Müller. Actor services - modular verification of message passing programs. In *ESOP*, 2016. doi:10.1007/978-3-662-49498-1_27.

21 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, 2015. doi:10.1145/2737924.2737958.