Inductive Sequentialization of Asynchronous Programs

Bernhard Kragl
IST Austria
Klosterneuburg, Austria
bkragl@ist.ac.at

Constantin Enea
Université de Paris, IRIF, CNRS,
Institut Universitaire de France (IUF)
Paris, France
cenea@irif.fr

Thomas A. Henzinger
IST Austria
Klosterneuburg, Austria
tah@ist.ac.at

Suha Orhun Mutluergil
Université de Paris, IRIF, CNRS
Paris, France
mutluergil@irif.fr

Shaz Qadeer
Calibra
Seattle, USA
shaz@fb.com

Abstract

Asynchronous programs are notoriously difficult to reason about because they spawn computation tasks which take effect asynchronously in a nondeterministic way. Devising inductive invariants for such programs requires understanding and stating complex relationships between an unbounded number of computation tasks in arbitrarily long executions. In this paper, we introduce *inductive sequentialization*, a new proof rule that sidesteps this complexity via a *sequential reduction*, a sequential program that captures every behavior of the original program up to reordering of coarse-grained commutative actions. A sequential reduction of a concurrent program is easy to reason about since it corresponds to a simple execution of the program in an idealized synchronous environment, where processes act in a fixed order and at the same speed. We have implemented and integrated our proof rule in the CIVL verifier, allowing us to provably derive fine-grained implementations of asynchronous programs. We have successfully applied our proof rule to a diverse set of message-passing protocols, including leader election protocols, two-phase commit, and Paxos.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: verification, asynchrony, concurrency, invariants, refinement, layers, movers, reduction, abstraction, induction

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385980

ACM Reference Format:
Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive Sequentialization of Asynchronous Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3385980

1 Introduction

Asynchronous programming is widely adopted for building responsive and efficient software. Unlike synchronous procedure calls that block the caller and hence must be executed immediately, asynchronous procedure calls do not block the caller and can be executed in parallel. Depending on the nature of the application, an asynchronous call could either execute in the same process (on another thread), a different process on the same node, or a different node entirely. Asynchronous programming is essential for distributed fault-tolerant software and client-server applications.

Asynchronous programs are notoriously hard to get right. There is inherent nondeterminism in their semantics due to the different orders in which asynchronous calls can execute. This complexity is exacerbated by further nondeterminism due to the execution platform, e.g., network delays and partitions in distributed applications. A promising approach to proving the correctness of realistic implementations is to go through a sequence of *abstraction* steps. Each abstraction step leads to a successively simpler program such that the correctness of the most abstract program implies the correctness of the most concrete program. Alternatively, a realistic implementation could be derived from an abstract (and obviously correct) program through a sequence of *refinement* steps. Devising an automated program verifier that enables refinement proofs is non-trivial and has received a great deal of attention recently [9, 12, 20–23, 28, 46].

Each link in the chain of programs connected together by refinement steps must be justified by a proof rule. A useful proof rule must be sound, broadly applicable, and able to

simplify reasoning about programs. Many such proof rules already exist including (1) variable introduction and elimination useful for changing the state representation, (2) reduction [34] for eliminating preemptions and creating atomic blocks, and (3) summarization for creating summaries of a block of code that executes atomically. These proof rules work together symbiotically and have been implemented to great effect in several refinement-oriented program verifiers [9, 15, 23].

In this paper, we introduce *Inductive Sequentialization* (IS), a new proof rule that works harmoniously with the other aforementioned proof rules, thereby extending the overall applicability of refinement-oriented program verifiers. IS simplifies reasoning about unbounded concurrent executions of an asynchronous program by reducing its correctness to that of a *single* interleaving of the concurrently-executing actions of the program. Our experience shows that the sequential reduction established by IS for a distributed protocol (like Chang-Roberts [10] and Paxos [31]) corresponds to an execution of the protocol in an idealized environment where processes execute in a fixed order, at the same speed, and messages are delivered immediately unless they are lost. This is the simplest execution of the protocol to reason about.

The goal of IS is to show that an asynchronous program $\mathcal{P}$ is a refinement of a sequential program $\mathcal{S}$. Here refinement means that the summary, the relation between initial and terminating states, of $\mathcal{P}$ is included in that of $\mathcal{S}$. Since any non-terminating program can be abstracted by one that terminates after a nondeterministically chosen number of steps, IS is capable of reasoning about all reachable states and arbitrary safety properties of asynchronous programs. IS combines inductive reasoning, showing that $\mathcal{S}$ summarizes a single fixed interleaving π of the asynchronous calls in $\mathcal{P}$, with commutativity reasoning, showing that focusing only on this single interleaving is sound.

The induction argument in IS is based on a user-provided (nondeterministic) procedure $\mathcal{I}$ that represents *all* prefixes of π. The asynchronous procedure calls whose effect is not included in a particular execution of $\mathcal{I}$ remain asynchronous and executable in an arbitrary order. This proof artifact is the analog of an inductive invariant in a proof of safety. However, unlike classical inductive invariants, inductiveness in our proof rule is shown only w.r.t. a *single* operation at a time, determined by π. The verification conditions prescribed by IS check that $\mathcal{S}$ is the "maximal" prefix of $\mathcal{I}$, which represents the complete interleaving π where no asynchronous calls remain to execute.

It remains to be shown that the sequential order determined by π and summarized by $\mathcal{I}$ captures all terminating executions of $\mathcal{P}$. To achieve this goal, we exploit the concept of a *left mover* [34], an atomic operation that may execute earlier than other concurrently-executing operations without changing the final state. If all operations in $\mathcal{P}$ are left movers, then they can be reordered arbitrarily, in particular following the fixed interleaving π, thus allowing us to conclude that every terminating state of $\mathcal{P}$ can be reached by π. This approach does not work on practical protocols because the concrete operations in $\mathcal{P}$ are not left movers. However, we observed that if π is suitably chosen, then for each operation A executed in π, there is an abstraction A' of A such that A' is a left mover and A' behaves identically to A in the context of π. This observation allows us to replace A with A' when performing the inductiveness check in the IS proof rule. This tight combination of induction, reduction, and abstraction is one of the main technical contributions of our work (described in detail in Section 4).

The applicability of IS is governed by two hypotheses which hold in well-designed asynchronous systems: (1) to ensure responsiveness, these systems extensively use short-living asynchronous tasks, e.g., message handlers, that can execute in parallel, and (2) asynchrony is meant to improve performance but not modify the logic of the application, i.e., the asynchronous behaviors should be equivalent to idealized *synchronous* behaviors where processes act at the same speed and the infrastructure, e.g., the network, is synchronous. The second hypothesis in particular has been addressed and justified in the context of a wide class of applications [5, 7, 11, 12, 16]. The first hypothesis offers more opportunities for a proof tactic based on reordering actions in an execution, while the second enables the reduction to reasoning about a single interleaving.

We have implemented IS as an extension of CIVL [23], a verification system for concurrent programs based on automated refinement reasoning. This extension allows IS to be interleaved with the other proof tactics implemented by CIVL. We have evaluated the usability of IS on a diverse set of message-passing protocols, including leader election protocols like Chang-Roberts, a non-trivial version of two-phase commit where nodes can abort early, and Paxos. We demonstrate that our proof rule enables sequentializations of all these protocols with a high degree of automation. Our evaluation shows that IS supports simple sequential reductions of complex protocols. Furthermore, the proof artifacts needed to establish the soundness of these reductions are also devised thinking only about a single fixed interleaving. Exploiting IS and other proof rules already implemented in CIVL, we are able to derive protocol implementations comprising fine-grained, verified event handlers which are similar to the unverified implementations written by programmers today.

In summary, this paper contributes: (1) a new proof rule called Inductive Sequentialization for eliminating concurrency from asynchronous programs, (2) an implementation of this rule in the CIVL verifier, and (3) a demonstration of its usefulness on a variety of challenging examples.

2 Overview

In this section we provide an overview of inductive sequentialization (IS). We motivate the challenges of deductive verification of asynchronous programs on a running example and then illustrate the concepts of IS on this example.

2.1 Motivation

Verification of concurrent programs. We present our verification technique in a general framework based on (gated) atomic actions over shared state and asynchronous thread creation, which abstracts away the details of any particular programming system irrelevant to our development. We will illustrate these concepts and our contribution on an example.

Figure 1-① shows a simple *broadcast consensus* protocol for n nodes (numbered from 1 to n) to agree on a common value. The local states of the nodes are represented using arrays, i.e., `value`[i] holds the input value of node i and `decision`[i] stores the final decision of node i. For each node i there are two concurrent threads created by the asynchronous calls in procedure `main`: one thread executes procedure `broadcast`(i) which sends the value of node i to every other node j, and the other thread executes procedure `collect`(i) which receives n values and stores the maximum as its decision. We consider the channels `CH`[i] for exchanging messages to be multisets (or bags) which models a network where messages can be delayed and delivered out-of-order, and the `receive` statement is blocking. Since every node receives the values of all other nodes, it is the case that, after the protocol finishes, all nodes must have decided on the same value, i.e.,

$$\forall i, j \in [1, n].\ \texttt{decision}[i] = \texttt{decision}[j], \tag{1}$$

where $[1, n]$ denotes the set of integers from 1 to n. However, proving this property directly on the code in Figure 1-① is notoriously complicated, i.e., requires an inductive invariant that is disproportionally complicated given the simplicity of the protocol. The challenge is that the send and receive operations across all nodes can execute in many different orders. An inductive invariant has to capture all of these orders, and represent every possible intermediate state that can occur. In (2) below we show that, even after reduction, the required inductive invariant remains complicated. This is in contrast to the following intuitive reasoning a programmer would employ to understand the correctness of the protocol:

> "First, all nodes send their values to each other (the order does not matter), and then, consequently, every node receives the same set of n values to compute the maximum (the order does not matter)."

Our proof rule is designed to facilitate this kind of reasoning about only a representative set of execution orders. In particular, we enable the programmer to think and reason about the program *sequentially*. To justify that we can focus the reasoning task on certain sequential execution orders and ignore all other concurrent execution orders, we build on the theory of mover types and reduction [15, 23, 34].

Atomic actions, mover types, and reduction. An execution of the program in Figure 1-① is naturally understood as an interleaving of small atomic (i.e., uninterruptible) actions of different threads. For instance, reading or writing a variable, sending a message, and spawning a new thread are all examples of fine-grained atomic actions. However, atomic actions are equally well suited to specify coarser-grained operations, and then the verification task can be understood as the sound summarization of fine-grained concurrent executions by large atomic actions. Concretely, we consider atomic actions of the form (ρ, τ), where ρ is a set of states (or one-state predicate), called *gate*, that specifies the states from which the action does not fail (like an assertion), and τ is a transition relation (or two-state predicate) that specifies the possible state transitions when the action executes (possibly including newly created threads). Note that the separation of ρ and τ is important to distinguish failure from blocking.

To formalize the idea that the execution order of atomic actions sometimes does not matter, we assign a *mover type* [18] to every atomic action in a program. An atomic action is a *left (right) mover* if it can be commuted to the left (right) of every other atomic action executed by a different thread, without altering the outcome of the execution. For example, over bag channels as in Figure 1-①, where messages can be received in an arbitrary order, `receive` is a right mover and `send` is a left mover. Furthermore, asynchronous calls (i.e., just the action of creating a new thread) are left movers, and local variable accesses like reading `value`[i] and writing `decision`[i] are both left and right movers (because no two concurrent threads access them at the same index i). Note that commutativity is checked pairwise among the pool of actions in a given program, only using the action definitions without considering reachable program executions. Thus an action can be a mover in one program, but not in another.

Given the mover types of the atomic actions in a program, consider a thread that, according to the static program order, executes a sequence of atomic actions with the following mover types: first a sequence of zero or more right movers, then at most one non-mover, and finally a sequence of zero or more left movers. We call such a sequence *atomic*, because any execution where these actions are interleaved with actions from other threads can be permuted into an equivalent execution where the atomic sequence is uninterrupted by other threads. Following this argument, the *reduction* method lets us summarize atomic sequences into bigger atomic actions. Figure 1-② shows the result of applying reduction to Figure 1-①, where all three procedures are atomic; `main` is a sequence of left-moving asynchronous calls, `broadcast` is a sequence of left-moving sends and both-moving reads of `value`[i], and `collect` is a sequence of right-moving receives and both-moving reads and writes of `decision`[i].

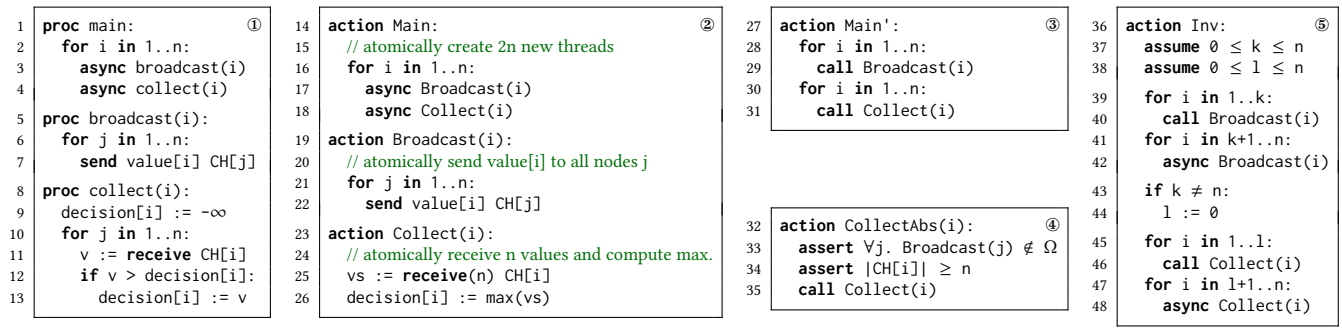

Figure 1. Broadcast consensus protocol. ① Original program. ② Program after reduction to atomic actions. ③ Sequentialization. ④ Abstraction of `Collect` action. ⑤ Partial sequentialization.

Here we want to stress two important points. First, we conveniently represent atomic actions as code blocks. While this makes, e.g., the action `Broadcast`(i) (Figure 1-②) visually appear the same as the procedure `broadcast`(i) (Figure 1-①), it represents an atomic broadcast of `value`[i] to all other nodes in one single step. Second, atomic actions can create new concurrent threads, represented as asynchronous calls. For example, executing action `Main` (Figure 1-②) has the effect of atomically creating $2n$ new threads (n `Broadcast`'s and n `Collect`'s), without yet executing any of their steps. We call these new threads *pending asyncs* (PAs), since their future effect is not summarized into the parent action. Formally, a PA is an action name together with parameter values, and we denote a set of pending asyncs with the variable Ω.

For the presentation in this paper we assume that programs are given as a set of atomic actions with PAs. In practice, this means that reduction is typically applied before our new technique, e.g., using the framework of layered concurrent programs [27]. In theory, this assumption is without loss of generality, since a non-atomic sequence of actions `A;B` can be represented with `A` having a PA to its continuation `B`.

Atomic actions are no silver bullet. Reducing a program to atomic actions with PAs is no panacea for the deductive verification of concurrent programs. In general, PAs still cause many different concurrent execution orders, and an inductive invariant has to capture all of them. For example, consider the inductive invariant for Figure 1-②:

$$\begin{aligned}
&(\Omega = \{\texttt{Main}\} \wedge (\forall i \in [1,n].\ \texttt{CH}[i] = \varnothing)) \\
\vee\ &(\exists D \subseteq [1,n].\ (\forall i \in [1,n].\ \texttt{CH}[i] = \{\texttt{value}[j] \mid j \in D\}) \wedge \\
&\quad \Omega = \{\texttt{Broadcast}(i) \mid i \in [1,n] \setminus D\} \uplus \\
&\quad\quad \{\texttt{Collect}(i) \mid i \in [1,n]\}) \qquad (2) \\
\vee\ &(\exists D \subseteq [1,n].\ (\forall i \in [1,n] \setminus D.\ \texttt{CH}[i] = \{\texttt{value}[j] \mid j \in [1,n]\}) \wedge \\
&\quad (\forall i \in D.\ \texttt{decision}[i] = \max\{\texttt{value}[j] \mid j \in [1,n]\}) \wedge \\
&\quad \Omega = \{\texttt{Collect}(i) \mid i \in [1,n] \setminus D\})
\end{aligned}$$

The first disjunct captures the initial state with a single PA to `Main` and all channels empty, the second disjunct captures the intermediate states where any subset of nodes D performed their `Broadcast` and the remaining `Broadcast`'s and all `Collect`'s are still pending, and the third disjunct captures the intermediate states where any subset of nodes D performed their `Collect`. Setting $D = [1,n]$ in the third disjunct implies the correctness property (1) and that no PAs are left (i.e., $\Omega = \varnothing$). Note that in this example the `Collect`'s happen after the `Broadcast`'s, because the `Collect`'s block until there are n messages in their corresponding channel. However, there are still two sources of complexity in reasoning with invariant (2) that our new method addresses. First, the ordering of `Broadcast`'s before `Collect`'s is not made explicit in the invariant; to show the inductiveness of (2) we have to prove that in a state with remaining `Broadcast`'s (i.e., satisfying the second disjunct) the `Collect`'s are blocked. Second, the execution order among the `Broadcast`'s and among the `Collect`'s does not matter, and thus we only want to reason about the "sequential" execution of `Broadcast`'s happening in order from 1 to n, and similarly for `Collect`'s.

2.2 Inductive Sequentialization

In this paper we provide an approach to enable sequential reasoning about asynchronous concurrent programs in the form of a program-transforming (refinement) proof rule called *inductive sequentialization (IS)*. A first idea of IS is to exploit mover types to *eliminate* PAs from atomic actions. By that we mean instead of an action creating a PA that takes effect asynchronously at a later time, we establish conditions that let us reason about the PA taking effect immediately, and thus combine it with the calling action. In particular, this is the case if the PA is a left mover, because then it can be moved earlier in an execution, to immediately follow its caller. However, atomic actions can generally create unboundedly many PAs, and the elimination of one PA can also introduce new ones if the eliminated PA has PAs itself. Our solution with IS to eliminate unboundedly many PAs at once is an *induction* scheme that has to address the following challenges:

- **C1** How to express intermediate results during the elimination of unboundedly many PAs?
- **C2** How to control the order of eliminating PAs to enable sequential reasoning?
- **C3** How to eliminate PAs that are not left movers?

We illustrate these challenges and how they are solved by IS on the consensus protocol in Figure 1-②. In particular, we

show how an application of IS derives that the consensus protocol is a sound refinement of the sequential program Main' in Figure 1-③. Main' represents a very simple schedule of the consensus protocol where all Broadcast's are executed before all Collect's, and in a round-robin fashion.

Challenge C1 is addressed by an *invariant action*, namely Inv in Figure 1-⑤. It represents (summarizes) the intermediate results during the induction, i.e., all prefixes of the schedule defining Main'. Therefore, either only some pending Broadcast's are already eliminated and the rest of the PAs are still pending (when $k < n$), or all Broadcast's and some number of Collect's are already eliminated (when $k = n$). Note that the number of Broadcast's or Collect's that are summarized by Inv is chosen nondeterministically. This allows Inv to summarize *all* prefixes of the schedule defining Main', one prefix for every choice of k and l. While we believe that the code of Inv is quite simple to understand, this is of course not the only way to represent prefixes of Main'. In general, IS is not sensitive to a particular representation.

Customary for an induction, IS has a base case and an induction step. The *base case* of IS, i.e., that the effect of Main is captured by Inv, is satisfied with $k = 0$. For the *induction step*, i.e., that the elimination of a Broadcast or Collect PA from Inv is still captured by Inv, we want to proceed with our sequential intuition and thus have to address challenge C2.

Every Broadcast PA created by a transition of Inv is a left mover, and thus any one of them could be eliminated next. However, the natural choice is to eliminate Broadcast$(k+1)$. For Inv this is also the only way to satisfy the induction step, by advancing from k to $k+1$. To communicate this choice to IS, the proof rule is parameterized with a *choice function* that selects the next PA to eliminate from any state with PAs left to eliminate. The choice function for our example always selects the Broadcast(i) PA with the smallest parameter i, as long as there exists one, and otherwise it selects the Collect(i) PA with the smallest parameter i.

The Collect actions are, however, not left movers, manifesting challenge C3. First, receives do not commute to the left of sends, and second, left movers also have to satisfy a *non-blocking* condition, namely that it is always possible to execute the action (from any state that satisfies its gate). A Collect action blocks in every state that has less than n messages to receive. The solution provided by IS is that *abstractions* for the atomic actions to be eliminated can be provided, which are used both for establishing left-moverness and to eliminate the PA selected by the choice function in the induction step. Note that there always exists a trivial abstraction that satisfies the mover conditions. Given an inductive safety invariant I, e.g., the one in Equation 2, every action can be abstracted to an arbitrary step between two states satisfying the invariant (i.e., an action defined by $\rho = I$ and $\tau = I \wedge I'$, where I' uses primed variables to represent the end state of a transition), which commutes with itself. This abstraction is of course not useful, since our goal is to avoid reasoning about this invariant in the first place.

We abstract Collect to CollectAbs given in Figure 1-④, which strengthens the gate (represented as assertion) from $\rho_{\texttt{Collect}} = \mathit{true}$ to $\rho_{\texttt{CollectAbs}} = \forall j.\ \texttt{Broadcast}(j) \notin \Omega \wedge |\texttt{CH}[i]| \geq n$. This assertion represents a condition which holds in the schedule defining Main', i.e., there are no concurrent Broadcast's when a Collect action is spawned and the channel accessed by the Collect already contains n messages. This makes CollectAbs non-blocking and a left mover. Thus IS is applicable and we show how CollectAbs is used in the induction step.

Assume that some prefix of Collect's from 1 to l are already eliminated, and Collect$(l+1)$ should be eliminated next (as indicated by the choice function). This is where the supplied abstraction comes in; instead of Collect$(l+1)$ we perform the induction step with CollectAbs$(l+1)$. In particular, this means that we need to show that after the transition of Inv it holds that Ω contains no Broadcast's and $|\texttt{CH}[l+1]| \geq n$. Observe that this is an entirely sequential verification condition, which holds because all Broadcast's happen before the Collect's in Inv. There are two important points to note about abstractions supplied to IS. First, these abstractions are merely proof artifacts used during IS. They are neither introduced into the program before nor left in the program after IS. Second, an abstraction is always only used for the single PA selected by the choice function. In particular, in Inv (Figure 1-⑤), CollectAbs is neither used for the already sequentialized Collect's (line 46) nor for the remaining PAs after $l+1$ (line 48). While in this example the gate of CollectAbs also holds there, this is not the case in general (see Section 4). Thus abstraction during IS is more powerful than abstraction before applying IS.

Finally, similar to a sequential loop invariant, which allows us to fast-forward through all iterations of a loop, the invariant action in IS allows us to fast-forward through all eliminations of PAs. For Inv (Figure 1-⑤) this means that we want to fast-forward to the point where all Broadcast's and Collect's have been eliminated. This is the case when $k = l = n$, and thus the result obtained by IS is the atomic action Main' in Figure 1-③. The formal guarantee of IS is that Main (Figure 1-②) *refines* Main' (Figure 1-③). Hence, we can replace reasoning about Main with reasoning about Main'. This action captures exactly how we imagined the broadcast consensus protocol to execute sequentially, and IS guarantees that this is a sound summary of all concurrent executions. Now we can prove property (1) using simple sequential reasoning, i.e., using sequential loop invariants for a particular execution order, as opposed to the complicated flat inductive invariant (2). Note also that the proof artifacts required to apply IS, i.e., the invariant action Inv and the abstraction CollectAbs, were themselves devised from this particular execution order only.

3 Preliminaries

In this section we provide the necessary definitions to formalize IS in the next section.

Variables and stores. Let $\mathcal{V}$ be a set of *variables* partitioned into *global variables* $\mathcal{V}_G$ and *local variables* $\mathcal{V}_L$. 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.

Actions and programs. Let $\mathcal{A}$ be a set of *action names* (usually denoted by uppercase letters like A in this paper). A *pending async (PA)* is a pair (ℓ, A) of a local store ℓ and an action name A (ℓ holds parameter values for A). A *gated atomic action*, or *action* for short, is a pair (ρ, τ), where the *gate* ρ is a set of stores and the *transition relation* τ is a set of transitions (σ, g, Ω) where σ is a (combined global and local) store, g is a global store, and Ω is a finite multiset of pending asyncs. A *program* is a finite mapping from action names to actions. Every program $\mathcal{P}$ must contain the dedicated action name `Main`, i.e., $\texttt{Main} \in \mathrm{dom}(\mathcal{P})$, and every action name that appears in $\mathcal{P}$ must be mapped to an action. For a set of action names $\mathcal{E}$ and a transition $t = (\sigma, g, \Omega)$ we define $PA_{\mathcal{E}}(t)$ to be the set of PAs to $\mathcal{E}$ in Ω, i.e., $PA_{\mathcal{E}}(t) = \{(\ell, A) \in \Omega \mid A \in \mathcal{E}\}$. To simplify the notation we will identify a PA (ℓ, A) with the singleton multiset $\{(\ell, A)\}$, and thus write $(\ell, A) \uplus \Omega$ for adding (ℓ, A) to Ω. We write $\mathcal{P}[A \mapsto a]$ to denote the program $\mathcal{P}'$ that is equal to $\mathcal{P}$ except that $\mathcal{P}'(A) = a$.

Executions. A *configuration* is a pair (g, Ω) of a global store g and a finite multiset of pending asyncs Ω,[1] or a unique *failure configuration* $\lightning$. We define the transition relation $\xrightarrow{\mathcal{P}}$ (omitting $\mathcal{P}$ when it is understood from the context) as

$$\frac{g \cdot \ell \in \rho \quad (g \cdot \ell, g', \Omega') \in \tau}{(g, \underline{(\ell, A)} \uplus \Omega) \to (g', \Omega \uplus \Omega')} \qquad \frac{g \cdot \ell \notin \rho}{(g, \underline{(\ell, A)} \uplus \Omega) \to \lightning}$$

where $\mathcal{P}(A) = (\rho, \tau)$. In a configuration (g, Ω), any PA $(\ell, A) \in \Omega$ can be scheduled to execute next; if the gate of A does not hold (i.e., $g \cdot \ell \notin \rho$) then the program "fails", otherwise a transition $(g \cdot \ell, g', \Omega') \in \tau$ atomically updates the global store to g' and creates new PAs Ω' (that are added to Ω). Underlining optionally denotes the PA that is executed in a transition. An *execution* π is a sequence of configurations $c_0 \to c_1 \to \cdots$. We call an execution *initialized* if it starts in a configuration $(g, (\ell, \texttt{Main}))$ with a single PA to `Main`, *terminating* if it ends in a configuration $(g, \varnothing)$ with no PAs, and *failing* if it ends in the failure configuration $\lightning$.

Refinement. We define the notion of refinement between both actions and programs [23]. Let $\circ$ denote the relation composition operator (sets are unary relations). In particular, $\rho \circ \tau = \{(\sigma, g, \Omega) \in \tau \mid \sigma \in \rho\}$ denotes the subset of transitions in τ that start from a store $\sigma \in \rho$.

[1] In our formalization we use multisets of PAs both "statically" in the definition of actions, and "dynamically" in configurations.

Definition 3.1. An action $a_1 = (\rho_1, \tau_1)$ *refines* an action $a_2 = (\rho_2, \tau_2)$, denoted $a_1 \preccurlyeq a_2$, if (1) $\rho_2 \subseteq \rho_1$ and (2) $\rho_2 \circ \tau_1 \subseteq \tau_2$. We also say that a_2 *abstracts* a_1.

The first condition states that a_2 has to preserve the failures of a_1. The second condition states that a_2 has to preserve the transitions of a_1 for initial stores from which a_2 cannot fail. Thus, a_2 can fail more often than a_1. For programs we are interested in the preservation of failing and terminating behaviors of initialized executions. Let $\mathit{Good}(\mathcal{P})$ be the set of initial stores from which $\mathcal{P}$ cannot fail, and $\mathit{Trans}(\mathcal{P})$ the relation between initial and final stores of terminating executions:

$$\mathit{Good}(\mathcal{P}) = \left\{g \cdot \ell \mid \neg \left(g, (\ell, \texttt{Main})\right) \xrightarrow{\mathcal{P}}{}^{*} \lightning\right\};$$

$$\mathit{Trans}(\mathcal{P}) = \left\{(g \cdot \ell, g') \mid \left(g, (\ell, \texttt{Main})\right) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing)\right\}.$$

Definition 3.2. A program $\mathcal{P}_1$ *refines* a program $\mathcal{P}_2$, denoted $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, if (1) $\mathit{Good}(\mathcal{P}_2) \subseteq \mathit{Good}(\mathcal{P}_1)$ and (2) $\mathit{Good}(\mathcal{P}_2) \circ \mathit{Trans}(\mathcal{P}_1) \subseteq \mathit{Trans}(\mathcal{P}_2)$. We also say that $\mathcal{P}_2$ *abstracts* $\mathcal{P}_1$.

Intuitively, this notion of refinement establishes a relationship between the summaries (input-output relations) of $\mathcal{P}_1$ and $\mathcal{P}_2$. If the programs contain no assertions (i.e., $\mathit{Good}(\mathcal{P}_1)$ and $\mathit{Good}(\mathcal{P}_2)$ contain all possible stores), it requires that the summary of the "concrete" program $\mathcal{P}_1$ is included in the summary of the "abstract" program $\mathcal{P}_2$. When assertions are present, it requires that $\mathcal{P}_2$ fails more often (condition 1) and that the summary of $\mathcal{P}_1$, restricted to initial states where $\mathcal{P}_2$ does not fail, is included in the summary of $\mathcal{P}_2$ (condition 2). This is sound in the sense that if $\mathcal{P}_2$ does not fail, then (1) $\mathcal{P}_1$ does not fail as well, and (2) any property of terminating states of $\mathcal{P}_2$ is also valid for the terminating states of $\mathcal{P}_1$.

Proposition 3.3. *If $a \preccurlyeq a'$, then $\mathcal{P}[A \mapsto a] \preccurlyeq \mathcal{P}[A \mapsto a']$.*

Left movers. An action $l = (\rho_l, \tau_l)$ is a *left mover w.r.t. an action* $x = (\rho_x, \tau_x)$ if
(1) the gate of l is *forward-preserved* by x, i.e., if ρ_l remains true after executing x whenever it was true before,

$$g \cdot \ell_l \in \rho_l \wedge (g \cdot \ell_x, g', \Omega) \in \rho_x \circ \tau_x \implies g' \cdot \ell_l \in \rho_l;$$

(2) the gate of x is *backward-preserved* by l, i.e., if ρ_x is true before executing l whenever it is true afterwards,

$$(g \cdot \ell_l, g', \Omega) \in \rho_l \circ \tau_l \wedge g' \cdot \ell_x \in \rho_x \implies g \cdot \ell_x \in \rho_x;$$

(3) l *commutes to the left of* x, i.e., if executing x before l leads to a global store that is also possible when executing l before x,

$$g \cdot \ell_l \in \rho_l \wedge (g \cdot \ell_x, \bar{g}, \Omega_x) \in \tau_x \circ \rho_x \wedge (\bar{g} \cdot \ell_l, g', \Omega_l) \in \tau_l \implies \exists \hat{g}.\ (g \cdot \ell_l, \hat{g}, \Omega_l) \in \tau_l \wedge (\hat{g} \cdot \ell_x, g', \Omega_x) \in \tau_x;$$

(4) l is *non-blocking*, i.e., if it contains a transition $(\sigma, g, \Omega) \in \tau_l$ from any store σ satisfying the gate ρ_l.

Furthermore, l is a *left mover w.r.t. a program* $\mathcal{P}$, denoted by $\mathit{LeftMover}(l, \mathcal{P})$, if it is a left mover w.r.t. every action in $\mathcal{P}$.

4 Inductive Sequentialization

In this section we present the *inductive sequentialization (IS)* proof rule. The context of IS is a program $\mathcal{P}$, an action name M, and a set of action names $\mathcal{E}$. The goal of IS is to eliminate all PAs to $\mathcal{E}$ from M, i.e., to summarize M together with the future effects of the PAs to $\mathcal{E}$ it creates. In particular, the IS proof rule replaces M with a new action that contains no PAs to $\mathcal{E}$. Formally, $\mathcal{P}$ is transformed into a program $\mathcal{P}'$ that is equal to $\mathcal{P}$, except that the action name M is re-mapped to a new action $(\rho_{M'}, \tau_{M'})$, i.e., $\mathcal{P}' = \mathcal{P}[M \mapsto (\rho_{M'}, \tau_{M'})]$. Notice that in general, M does not have to be the `Main` action of $\mathcal{P}$.

The correctness requirement for IS is that $\mathcal{P}$ refines $\mathcal{P}'$, which means that $\mathcal{P}'$ has to preserve both failing and terminating behaviors of $\mathcal{P}$ (see Definition 3.2). In particular, every terminating state of $\mathcal{P}$ must also be reachable by $\mathcal{P}'$:

$$(g, (\ell, \texttt{Main})) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, (\ell, \texttt{Main})) \xrightarrow{\mathcal{P}'}{}^{*} (g', \varnothing).$$

The natural strategy to prove this property is to show that every terminating $\mathcal{P}$-execution π can be rewritten into a terminating $\mathcal{P}'$-execution π' with the same final state, by turning every transition of M in π into a transition of M' in π'. We illustrate this process in Figure 2, where ① shows the final part of a $\mathcal{P}$-execution. First M executes from a configuration with two other PAs to X and Y, which creates two new PAs to A and B, and then X, B, Y, A execute to reach a terminating configuration. Suppose $\mathcal{E} = \{A, B\}$, and thus our goal is to obtain the execution in ⑥ which executes M' instead of M, which does not create PAs to A and B. We do so by setting up an induction that stepwise eliminates A and B from the execution in ①. The central artifact for this induction is an *invariant action* I that has to be provided as input to IS. Then the first step in ②, constituting the base case of the induction, is to execute I instead of M, which requires that every transition of M is also a transition of I (or more precisely, that M refines I). At this point the transition of I denotes an "empty sequentialization" which we are going to extend in the next steps to "partial sequentializations", until we obtain the "complete sequentialization" M'. In doing so we control the constructed sequentialization through a *choice function* that determines for every partial sequentialization a *single* PA to sequentialize next. Concretely, in ② we first want to sequentialize A and then B, and thus the choice function selects A in the transition of I (marked with a box around A). We commute A to the left of Y, B, and X to obtain ③, which requires that A is a *left mover* w.r.t. the actions in $\mathcal{P}$. Then the *induction condition* of IS guarantees that the composition of I and A is possible as a single transition of I (corresponding to an extended partial sequentialization), and thus we obtain ④. Crucially, the transition of I in ③ only has to be inductive w.r.t A. Now we proceed similarly with the PA to B—commute B to the left of X and absorb it into I—to obtain ⑤. However, it might be the case that B is not an unconditional left mover. Therefore it is possible to

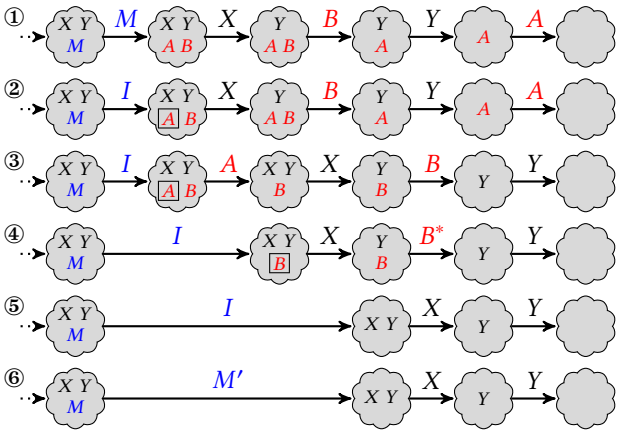

Figure 2. Illustration of the induction argument. Clouds represent the set of PAs in a configuration (stores are not shown) and the arrow labels indicate the actions that execute in the transitions from one configuration to the next.

supply abstractions for actions in $\mathcal{E}$ to IS, like B^* for B in ④. These abstractions can take into account the context in which they are sequentialized, e.g., B^* can rely on the fact that A already executed. Finally, in ⑥ we replace I with M', which is constructed from I by removing every transition that has PAs to $\mathcal{E}$, and thus obtain the desired $\mathcal{P}'$-execution.

We remark that in general, IS sequentializes not only the PAs created directly by M, but also transitively created PAs. This capability is essential to sequentialize unbounded sequences of PAs. Furthermore, M' can still have PAs to actions disjoint from $\mathcal{E}$. Then IS can be applied to M' again, and in Section 5.3 we show that iterated application of IS can be beneficial in practice.

Inductive sequentialization. The formal definition of the IS proof rule is given in Figure 3. Besides the program $\mathcal{P}$, action name M, and set of action names $\mathcal{E}$ which frame the rule, an *invariant action* (ρ_I, τ_I), a *choice function* f, an *abstraction function* α, and a *well-founded order* over configurations $\gg$ have to be provided. The choice function f selects from every transition t of the invariant action that creates PAs to $\mathcal{E}$ (i.e., $PA_{\mathcal{E}}(t) \neq \varnothing$) one of these PAs. The *abstraction function* α is such that $\alpha(A)$ is an abstraction of $\mathcal{P}(A)$ for every $A \in \mathcal{E}$. Note that we can set $\alpha(A) = \mathcal{P}(A)$ for every A that should not be abstracted. The induction argument outlined above is enabled by three *induction conditions*. To start the induction (cf. Figure 2, ①-②), condition **(I1)** requires the invariant action to be an abstraction of the action we rewrite. To end the induction (cf. Figure 2, ⑤-⑥), condition **(I2)** requires M to be re-mapped to an action that abstracts the invariant action with all transitions that contain PAs to $\mathcal{E}$ removed. To absorb a PA into the invariant action (cf. Figure 2, ③-④), condition **(I3)** is two-fold, corresponding to the failure and behavior preservation requirement. First, after every transition t of the invariant action, if A is the PA selected by the choice

$$\frac{\begin{array}{l}\textbf{Given: } \mathcal{P}, M, \mathcal{E} \qquad \textbf{To invent: } \rho_I, \tau_I, f, \alpha, \gg \\ \mathcal{P}(M) = (\rho_M, \tau_M) \qquad \mathcal{E} \subseteq \mathrm{dom}(\mathcal{P}) \qquad \mathit{WellFounded}(\gg) \\ \mathrm{dom}(f) = \{t \in \tau_I \mid PA_{\mathcal{E}}(t) \neq \emptyset\} \qquad t \in \mathrm{dom}(f) \implies f(t) \in PA_{\mathcal{E}}(t) \\ \mathrm{dom}(\alpha) = \mathcal{E} \qquad A \in \mathcal{E} \implies \mathcal{P}(A) \preccurlyeq \alpha(A) \\ \textbf{(I1)}\ (\rho_M, \tau_M) \preccurlyeq (\rho_I, \tau_I) \qquad \textbf{(I2)}\ (\rho_I, \{t \in \tau_I \mid PA_{\mathcal{E}}(t) = \emptyset\}) \preccurlyeq (\rho_{M'}, \tau_{M'}) \\ \textbf{(I3)}\ t = (\sigma, g, (\ell, A) \uplus \Omega) \in \rho_I \circ \tau_I \wedge f(t) = (\ell, A) \wedge \alpha(A) = (\rho_{A^*}, \tau_{A^*}) \implies \\ \qquad g{\cdot}\ell \in \rho_{A^*} \wedge \left((g{\cdot}\ell, g', \Omega') \in \tau_{A^*} \implies (\sigma, g', \Omega \uplus \Omega') \in \tau_I\right) \\ \textbf{(LM)}\ A \in \mathcal{E} \implies \mathit{LeftMover}(\alpha(A), \mathcal{P}) \\ \textbf{(CO)}\ A \in \mathcal{E} \wedge \alpha(A) = (\rho_{A^*}, \tau_{A^*}) \wedge g{\cdot}\ell \in \rho_{A^*} \implies \\ \qquad \exists g', \Omega'.\ (g{\cdot}\ell, g', \Omega') \in \tau_{A^*} \wedge (g, (\ell, A) \uplus \Omega) \gg (g', \Omega \uplus \Omega')\end{array}}{\mathcal{P} \preccurlyeq \mathcal{P}[M \mapsto (\rho_{M'}, \tau_{M'})]}$$

Figure 3. Inductive sequentialization (IS) proof rule.

function and A^* the abstraction of A, then the gate of A^* has to be satisfied. In other words, any potential failure of A^* has to be propagated into the gate of I. Second, when t is composed with a transition of A^*, then the resulting composite transition must be contained in I and thus possible in a single step. To commute the actions in $\mathcal{E}$ to the appropriate position in the sequentialization (cf. Figure 2, ②-③), the *left-mover condition* **(LM)** requires that every abstracted action $\alpha(A)$ is a left mover w.r.t. the original actions in the program $\mathcal{P}$. Thus, abstracted actions do not have to be left movers w.r.t. each other, which is evident in Figure 2 where at most one abstraction at a time is part of the execution. Finally, the *cooperation condition* **(CO)** strengthens the standard non-blocking conditions of left movers. It states that is must be possible to execute every abstracted action such that the configuration decreases in some well-founded order $\gg$. (Recall that $\gg$ is well-founded, if there is no infinite sequence $c_0, c_1, c_2, \ldots$ of configurations such that $c_i \gg c_{i+1}$ for every $n \in \mathbb{N}$.) This ensures that it is always possible for the PA elimination process to eventually finish, instead of indefinitely introducing new PAs to be eliminated. While the cooperation condition might seem exotic, its sole purpose is the prevention of unsound IS on pathological corner cases, and the condition is easy to satisfy in practice (see below).

Example 4.1. Recall the broadcast consensus protocol from Figure 1. We formally apply IS to transform `Main` to `Main'` by eliminating all PAs to `Broadcast` and `Collect`. Thus we set M = `Main`, $\mathcal{E}$ = {`Broadcast`, `Collect`}, M' = `Main'`, and I = `Inv`. Recall that `Inv` represents all partial sequentializations where `Broadcast`'s execute in the fixed order from 1 to n, followed by the `Collect`'s executing in the fixed order from 1 to n. Thus **(I1)** `Main` is summarized by `Inv` (for $k = l = 0$) and **(I2)** `Main'` summarizes the only execution of `Inv` without remaining PAs (for $k = l = n$). We define the choice function f such that it either selects the `Broadcast` PA with the smallest parameter if one exists, or otherwise the `Collect` PA with the smallest parameter. `Broadcast` is a left mover w.r.t. {`Main`, `Broadcast`, `Collect`} and does not need to be abstracted. However, `Collect` is not a left mover because it does not commute with `Broadcast` and it also does not satisfy the non-blocking condition. Thus we supply the abstraction α(`Collect`) = `CollectAbs` that strengthens the gate to assert that there are no `Broadcast`'s left and at least n messages to receive, which makes `CollectAbs` a left mover. Now the induction condition **(I3)** requires us to discharge the gate of `CollectAbs` in a sequential context when we compose it with `Inv`. This is possible since `Inv` executes all `Broadcast`'s before any `Collect`. We set $\gg$ such that $c \gg c'$ if and only if c has more PAs than c'. Then $\gg$ is clearly well-founded because the number of PAs cannot be negative, and the cooperation condition **(CO)** is satisfied because the execution of `Broadcast`/`Collect` decreases the number of PAs (since they do not create new PAs).

Cooperation is necessary. We illustrate the need for the cooperation condition on the following program.

```
1 action Main:     action Rec:     action Fail:
2   async Rec        async Rec       assert false
3   async Fail
```

This program can fail in two steps by executing `Main` followed by `Fail`. Suppose we want to apply IS with M = `Main`, $\mathcal{E}$ = {`Rec`}, and I = `Main`, which satisfies all conditions except cooperation. In particular, notice that `Fail` is not in $\mathcal{E}$ and thus the induction condition does not apply to it. But then M' is constructed from I by removing all transitions from `Main` that have PAs to `Rec`, which means all transitions. Thus, the transition relation of M' is empty, i.e., $\tau_{M'} = \emptyset$ (which we can also represent as `assume false`). Then replacing M with M' would result in a program that cannot fail, which is unsound according to the definition of refinement. The cooperation condition prevents the application of IS because the execution of `Rec` in any configuration results in exactly the same configuration, and thus it is impossible to decrease in any well-founded order.

Checking cooperation is easy. The cooperation condition in Figure 3 is *global* in the sense that it requires a decrease $(g, (\ell, A) \uplus \Omega) \gg (g', \Omega \uplus \Omega')$ for any possible Ω. This is the most general condition needed in our soundness proof, but in practice we did not find it necessary for $\gg$ to depend on Ω. Instead it is natural for $\gg$ to be *monotonic*, i.e., that $(g, \Omega) \gg (g', \Omega')$ implies $(g, \Omega \uplus \Omega'') \gg (g', \Omega' \uplus \Omega'')$. Then cooperation can be checked *locally* by showing $(g, (\ell, A)) \gg (g', \Omega')$. Furthermore, we found the following simple generic pattern to apply to all of our examples in Section 5. Devise a map g from configurations c to tuples $(x_1, \ldots, x_n)$, such that every x_i is either the number of messages in a certain channel, or the number of PAs of a certain action. Then define $c \gg c'$ if and only if $g(c) > g(c')$, where $>$ denotes the lexicographic order of tuples of natural numbers. This construction guarantees that $\gg$ is well-founded and monotonic, and the cooperation condition is easy to check syntactically.

4.1 Soundness Proof

In this section, let $\mathcal{P} \preccurlyeq \mathcal{P}'$ be derived by an application of IS.

Lemma 4.2. *For every failing $\mathcal{P}$-execution π starting with a transition of M there exists a failing $\mathcal{P}'$-execution π' starting from the same configuration with a transition of M, i.e.,*

$$(g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}}{}^{*} \lightning \implies (g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}'}{}^{*} \lightning .$$

Moreover, π' does not execute more PAs to M than π.

Proof. Let π be a failing $\mathcal{P}$-execution that starts with a transition of M:

$$\pi = (g, \underline{(\ell, M)} \uplus \Omega) \rightarrow \cdots \rightarrow \lightning .$$

We show how to rewrite π into a failing $\mathcal{P}'$-execution

$$\pi' = (g, \underline{(\ell, M)} \uplus \Omega) \rightarrow \cdots \rightarrow \lightning .$$

We (conceptually) replace the first transition of M in π with the invariant action I.

Case 1. $g{\cdot}\ell \notin \rho_I$. Then because $\rho_{M'} \subseteq \rho_I$ we obtain

$$\pi' = (g, \underline{(\ell, M)} \uplus \Omega) \rightarrow \lightning .$$

Case 2. $g{\cdot}\ell \in \rho_I$. Then because of $(\rho_M, \tau_M) \preccurlyeq (\rho_I, \tau_I)$ we must have

$$\pi = (g, \underline{(\ell, M)} \uplus \Omega) \rightarrow (g', \Omega \uplus \Omega') \rightarrow \cdots \rightarrow \lightning .$$

Furthermore, some transition $t \in \tau_I$ can simulate the first transition in π, and we denote π'' the remainder of π after the first transition:

$$t = (g{\cdot}\ell, g', \Omega') \in \tau_I; \qquad \pi'' = (g', \Omega \uplus \Omega') \rightarrow \cdots \rightarrow \lightning .$$

We consider the lexicographically ordered measure on π'' comprising (1) the length of π'', ordered by $\geq$, and (2) the final configuration in π'' before the failure, ordered by $\gg$.

Case 2.1. $PA_{\mathcal{E}}(t) = \varnothing$. Then $t \in \tau_{M'}$ and we obtain $\pi' = \pi$.

Case 2.2. $PA_{\mathcal{E}}(t) \neq \varnothing$. Let $(\ell', A) \in \Omega'$ be the PA selected by the choice function, i.e., $f(t) = (\ell', A)$. Let A^* be the abstraction of A. By the induction condition the gate of A^*, which is stronger than the gate of A, holds at the beginning of π'' (i.e., $g{\cdot}\ell' \in \rho_{A^*}$) and because of forward preservation it also holds in every later configuration of π''. In particular, the execution of (ℓ', A) cannot be the failing transition.

Case 2.2.1. (ℓ', A) executes in π''. We turn this transition of A into on of A^*, which can simulate the transition of A. Because A^* is a left mover, we stepwise commute it to the left in π'' such that it becomes the first transition. Let t' be the corresponding transition in A^*. (Note that some action X that we move A^* to the left of could now fail and thus π'' could be shortened.) By the induction condition t and t' can be composed into a single transition $t'' \in \tau_I$. We are in Case 2 with decreased measure.

Case 2.2.2. (ℓ', A) does not execute in π''. We insert a transition of (ℓ', A^*) into π'' right before the failure. Recall that the gate of A^* is satisfied, and by the cooperation condition we can execute A^* such that the final configuration decreases according to $\gg$. Because of backward preservation, the original failure is preserved after A^*. We proceed as in Case 2.2.1 to move A^* to the left and absorb it into I. Then we are again in Case 2 with decreased measure.

The above rewriting process obviously does not introduce new transitions of M into π'. □

Lemma 4.3. *For every terminating $\mathcal{P}$-execution π starting with a transition of M there exists a $\mathcal{P}'$-execution π' that starts from the same configuration as π with a transition of M and either fails or ends in the same configuration as π, i.e.,*

$$(g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, \underline{(\ell, M)} \uplus \Omega) \xrightarrow{\mathcal{P}'}{}^{*} c$$

where $c \in \{\lightning, (g', \varnothing)\}$. Moreover, π' does not execute more PAs to M than π.

Proof. We rewrite π into π' exactly the same way as in the proof of Lemma 4.2. If no failure is introduced (which is possible in Case 1 and Case 2.2.1) we are guaranteed to preserve the final configuration of π (and never reach Case 2.2.2). Otherwise we obtain a failing π' from Lemma 4.2. □

Theorem 4.4. *The IS proof rule in Figure 3 is sound.*

Proof. By repeated application of Lemma 4.2 and Lemma 4.3 every $\mathcal{P}$-execution π can be rewritten into a refinement-preserving $\mathcal{P}'$-execution π'. □

5 Evaluation

In this section, we report on our experience of using IS for the verification of functional correctness of a diverse set of example programs (see Table 1). We argue that IS is *applicable* (to realistic programs), *automatable* (using sequential verifiers), and *user-friendly* (by appealing to sequential intuition). Our tool and all examples are publicly available as part of the Boogie project [1] and long-term archived [26].

5.1 Implementation

We implemented IS as an extension of CIVL [23], a verification system for concurrent programs based on automated and modular refinement reasoning. CIVL implements the framework of layered concurrent programs [27] where the input consists of a description of a sequence of (related) programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ and the verification goal is to establish the chain of refinement $\mathcal{P}_1 \preccurlyeq \cdots \preccurlyeq \mathcal{P}_n$. The justification for every refinement step between two subsequent programs $\mathcal{P}_i \preccurlyeq \mathcal{P}_{i+1}$ is compiled into sequential verification conditions of the Boogie verifier [3], which are then discharged automatically by an SMT solver. We seamlessly integrated IS into CIVL, such that every refinement step can now either be an IS transformation or an existing CIVL transformation.

Our integration of IS into CIVL comprises roughly 2500 lines of C# code, addressing the following challenges. First, we adapted the type checker to integrate IS into the language of layered concurrent programs, which avoids extensive repetition of program parts that do not change in most refinement steps (observed as hindering, e.g., in [9]). Second, we designed a representation of PAs as multisets in the generalized array theory [13] and extended the existing refinement checker with the capability to summarize unboundedly many asynchronous calls as PAs. Third, the actual conditions of IS (Figure 3) are compiled to sequential verification conditions in Boogie. In particular, one sequential Boogie procedure encodes each of the following checks: (1) M refines I, (2) I is inductive w.r.t. the abstraction of a chosen PA, (3) I without transitions that create PAs to $\mathcal{E}$ refines M', and (4) A refines its abstraction $\alpha(A)$ for $A \in \mathcal{E}$. The left-mover conditions are automatically discharged by the existing mover engine. Due to this fine-grained decomposition, we can generate targeted error messages for failed checks that are local to at most two actions.

5.2 Verification Methodology

In this section we report on the verification methodology we followed to verify our examples and specifically illustrate it on our most significant example, Paxos.

Paxos. The Paxos [31] protocol establishes consensus among a set of unreliable nodes in an asynchronous network without a central coordinator. We consider a single-decree Paxos variant that establishes consensus on a single value. Conceptually, Paxos operates in a sequence of (increasingly numbered) rounds, where each round is associated to a *proposer* node. The proposers communicate with a set of *acceptor* nodes to try to either decide on a newly proposed value or to learn about a previously decided value. Since the proposers operate concurrently on different rounds, Paxos resolves conflicts using a mechanism that requires proposers to collect in two subsequent phases "enough" responses (called quorum) from acceptors, while acceptors stop working on a round when they hear about a higher round. Thus every round can remain undecided (in general, consensus cannot be guaranteed in an asynchronous network), but we want to prove that two rounds never decide on conflicting values.

```
var acceptorState: Node -> AcceptorState
var decision: Round -> Option<Value>
var joinChannel: Round -> Bag<JoinResponse>
var voteChannel: Round -> Bag<VoteResponse>

proc Paxos()                proc Conclude(r: Round, v: Value)
proc StartRound(r: Round)   proc Join(r: Round, n: Node)
proc Propose(r: Round)      proc Vote(r: Round, n: Node, v: Value)
```

(a) Concrete implementation

```
datatype VoteInfo(value: Value, nodes: Set<Node>)
var joinedNodes: Round -> Set<Node>
var voteInfo: Round -> Option<VoteInfo>
var pendingAsyncs: Bag<PA>

action Propose(r: Round) returns (pending_async PAs: Bag<PA>):
  var ns: Set<Node>, v: Value

  assert Propose(r) ∈ pendingAsyncs
  assert voteInfo[r] = None()

  if (*):
    assume ns ⊆ joinedNodes[r] ∧ IsQuorum(ns)
    ... // compute v from r, ns, and voteInfo
    voteInfo[r] := Some(VoteInfo(v, ∅))
    PAs := {Vote(r, n, v) | n: Node} ⊎ {Conclude(r, v)}
  ...
```

(b) Atomic actions for applying inductive sequentialization

```
action ProposeAbs(r: Round) returns (pending_async PAs: Bag<PA>):
  assert {StartRound(r') ∈ pendingAsyncs | r' ≤ r} = ∅
  assert {Join(r', n')   ∈ pendingAsyncs | r' ≤ r} = ∅
  ... // same as Propose

action Paxos() seq Paxos' with PaxosInv
elim StartRound, StartRoundAbs elim Propose, ProposeAbs elim ...
  ...

action Paxos'():
  assert ∀ r. decision[r] = None()
  havoc decision with ∀ r1, r2, v1, v2.
    decision[r1] = Some(v1) ∧ decision[r2] = Some(v2) ⟹ v1 = v2

action PaxosInv() returns (pending_async PAs: Bag<PA>, choice c: PA)
  ...
```

(c) Inductive sequentialization

Figure 4. Excerpts from our Paxos proof.

Implementation. Our examples are implemented as low-level concurrent programs $\mathcal{P}_1$ that only use primitive atomic actions, like reading or writing a single memory address, and sending or receiving a single message. Figure 4(a) shows the variable and procedure declarations of our Paxos implementation. The procedures `Propose` and `Conclude` are associated to the proposer role in the Paxos protocol, while `Join` and `Vote` are associated to the acceptor role. A client calls `Paxos`, which creates an arbitrary number of asynchronous `StartRound` tasks. For each round, the corresponding `StartRound` task creates one `Join` task per acceptor and one `Propose` task. According to each acceptor's current state (in `acceptorState`), `Join` sends a `JoinResponse` message to a channel in `joinChannel`. `Propose` executes a loop that receives from this channel and aggregates the response messages. If a quorum is reached, it proposes a value by creating one `Vote` task per acceptor and one `Conclude` task. Then, the

`Vote` tasks send `VoteResponse` messages that are aggregated in a loop by `Conclude`. If a quorum is reached, `Conclude` updates `decision` for the corresponding round from `None()` to `Some(v)` where `v` is the decided value.

Atomic actions. IS operates on atomic actions, and thus we first apply an existing CIVL transformation on $\mathcal{P}_1$ to obtain suitable atomic actions that summarize the low-level procedures, forming $\mathcal{P}_2$. Crucially, this step does not require any concurrent invariants related to the correctness of the protocol. However, the subsequently enabled application of IS significantly simplifies the construction of the actual proof of functional correctness. Furthermore, we note that the structured proofs obtained by our methodology are not only—in our experience—simpler to construct, but also more modular and thus better suited for change than flat inductive invariants. For example, changing low-level details in the implementation only requires a revision of $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, but does not affect the rest of the proof.

For Paxos we also make use of CIVL's capability to change the state representation of the program. Concretely, we hide the implementation variables `acceptorState`, `joinChannel`, and `voteChannel`, and instead introduce the abstract variables `joinedNodes` and `voteInfo` shown in Figure 4(b). Also, we introduce `pendingAsyncs` to hold the current set of pending asyncs. As an example, Figure 4(b) shows the action summary of `Propose`. Instead of a loop that aggregates messages from a channel, it atomically initializes `voteInfo[r]` to `VoteInfo(v,∅)` if there is a quorum in `joinedNodes[r]`, where `v` is the proposed value and ∅ the initially empty set of acceptors that voted on it. Notice that the action `Propose` has an additional specially-declared output variable `PAs` that represents the PAs created by the action.

Inductive sequentialization. In our experience, the key to apply IS is the intuition of idealized, sequential executions of the program. The main creative task is the invention of this sequentialization, while all required proof artifacts are derived from it. In particular, the invariant action I and the choice function f are determined from partial sequential executions, M' summarizes completed sequential executions, and left-moving abstractions α can assert to only execute in the sequential context.

The sequentialization idea for Paxos is to execute one round at a time (in increasing order), and within each round execute actions in a fixed order. In particular, abbreviating action names with their first letter and denoting round boundaries by a vertical bar, the sequentialization looks as follows:

S(1) J(1,1) J(1,2) P(1) V(1,1,_) V(1,2,_) C(1,_) | S(2) J(2,1) ...

To preserve all original behaviors of the protocol, we observed that the effect of rounds being blocked from reaching a decision due to overlapping proposals or out-of-order message delivery is equivalent to both acceptors and proposers nondeterministically dropping incoming messages. For example, notice that the state update in `Propose` is guarded by a nondeterministic conditional on line 16 (which is not present in the low-level implementation $\mathcal{P}_1$ but introduced in $\mathcal{P}_2$).

Our goal is to apply IS to transform `Paxos` to `Paxos'` in Figure 4(c). `Paxos'` is a straight-forward high-level specification of Paxos, stating that the protocol consistently updates `decision`, i.e., no two rounds decide on conflicting values. A client could be provided with an API to query `decision` and would then use `Paxos'` to reason about its own consistency. The application of IS is declared on action `Paxos` (line 26), which prescribes the use of invariant action `PaxosInv` to simultaneously eliminate all other actions using the left-mover abstractions given in the `elim` clauses (line 27). For example, `ProposeAbs` in Figure 4(c) strengthens the gate of `Propose` with the information that, in the sequentialization, only `Join` and `StartRound` actions with higher round numbers can still be pending. All our abstractions are of this simple kind. The choice function is specified by the programmer using a special output variable `c` of `PaxosInv`, see line 33.

Our invariant action `PaxosInv` consists of four parts:

1. *Sequentialization*: Rounds execute one after another, and within rounds there is a fixed order of phases.
2. *Quorum before decision*: If there was a decision for value v, then there must have been a proposal and a quorum of nodes that voted for v (in the same round).
3. *Voting after decision*: If there was a decision in round r_1 for value v_1 and some higher round r_2 votes on value v_2, then $v_2 = v_1$.
4. *Safety*: If two rounds reach a decision, then it is on the same value.

Property 1 encodes the sequentialization order and lets us discharge the gates of our left-mover abstractions. Properties 2/3/4 capture the core mechanism of the protocol and are quite easy to state.

Invariant complexity. We demonstrate the significant simplifications afforded by IS in terms of invariant complexity by comparing against the baseline of standard "asynchrony-aware" inductive invariants (over the original asynchronous program). In particular, we compare to the well-documented Ivy invariants given in [39], but stress that these invariants are representative for other systems like [9, 22, 32, 47]. While these works have excellent contributions elsewhere, the methodology to deal with the protocol complexity boils down to the above baseline. The only other approaches we know of that focus on improving this particular aspect are [2, 12, 28, 46], but they do not apply to our example programs (see Section 6).

Properties 2/3/4 above correspond roughly to formulas (4)-(7) in [39]. However, the Ivy invariant requires five additional conjuncts (8)-(12), which capture more complicated dependencies of overlapping rounds and are much harder to invent. Due to sequentialization, we do not need any analogue of these in our invariant.

Table 1. Examples verified with IS.

Example	#IS	#LOC Total	#LOC IS	#LOC Impl	Time sec
Broadcast consensus	2	396	108	121	1.0
Ping-Pong	1	281	91	106	0.9
Producer-Consumer	1	225	65	93	0.9
N-Buyer	4	681	251	256	2.6
Chang-Roberts	2	377	117	135	1.1
Two-phase commit	4	553	181	222	1.4
Paxos	1	1168	534	302	4.2

5.3 Other Case Studies

We demonstrate the broad applicability of IS by applying it on the examples listed in Table 1. These examples cover a wide variety of characteristics of concurrent programs, including modes of concurrency (tightly synchronized, mostly independent, coordination-focused, phase-oriented, long-running), communication topologies (complete, star, ring, pipeline), channel types (bags, queues), and specifications (consensus, unique leader, assertions). We avoided any hidden simplifications in the communication between processes (e.g., arranging broadcast-receive communication with a set of nodes sequentially instead of concurrently), and included realistic performance optimizations which generally complicate verification.

Column #IS reports the number of IS applications. For some programs we preferred the repeated application of IS, although the proof could be accomplished by a single application. This is because an action that is eliminated in one IS application disappears from the pool of actions w.r.t. which left-moverness has to be established in a subsequent IS application. For example, as an alternative to the one-shot proof of the broadcast consensus protocol in Figure 1 we performed a proof that first eliminates `Broadcast` in one IS application, and then `Collect` in a second IS application. Then the abstraction `CollectAbs` in Figure 1-④ does not need the gate on line 33, because `CollectAbs` does not have to commute to the left of `Broadcast`.

The #LOC columns report numbers of CIVL lines of code. CIVL, as Boogie, is an intermediate verification language not optimized for conciseness. Our files contain a lot of boilerplate code that would be part of a library for any frontend language. Concretely, this includes declarations of builtin SMT types, type declarations for pending asyncs, theory axioms (e.g., for sets), primitive atomic actions (e.g., send/receive), etc. Thus, besides (1) the total lines we also report (2) the lines related to IS steps, and (3) the lines related to the implementation $\mathcal{P}_1$ and existing CIVL step $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.

The last column reports the total verification time. Our tool is fast and thus suitable for interactive development. However, we acknowledge observing run-time fluctuations caused by small (semantically irrelevant) modifications, likely due to heuristics for quantifier reasoning. Improving the robustness of checkers for complex verification conditions is an important avenue for future work.

We finally provide a brief description of the remaining examples besides broadcast consensus and Paxos.

Ping-Pong. In this example a Ping process sends increasing numbers to a Pong process, expecting the number to be acknowledged back. Our sequentialization makes the alternation of the Ping and Pong process explicit. We verify assertions in the program, which state that the Pong processes receives increasing numbers, and the Ping process receives correct acknowledgments.

Producer-Consumer. This is a variation of the Ping-Pong example, where a producer enqueues increasing numbers into a shared queue, and a consumer dequeues numbers from the queue and verifies that they are indeed increasing. The Producer-Consumer example has more concurrent executions than the Ping-Pong example, because the producer can be arbitrarily faster than the consumer, and thus the queue can grow arbitrarily big. However, IS reduces the program to a sequentialization where the producer and consumer alternate, and thus the queue contains at most one element.

N-Buyer. In this example n buyer processes coordinate the purchase of an item from a seller. That is, one buyer requests a quote for the item from the seller, then the buyers coordinate their individual contribution, and finally if the contributions are enough to buy the item, and order is placed. This example was adapted from [8] and is representative for the coordination protocols targeted by session types. We added and verified a functional correctness specification that states that if a final order is placed then the sum of contributions promised by the buyers actually adds up to the price of the ordered item.

Chang-Roberts. This is a leader election protocol in a ring topology [10]. Each node starts by sending its own (unique) ID to its neighbor in the ring, and then forwards incoming messages with IDs greater than its own. When a node receives its own ID, it declares itself as leader. We prove that there can be at most one leader. Our sequentialization follows from the intuition that only the node with the highest ID, say m, can become a leader, and for that its ID has to traverse the ring once. We sequentialize the program such that each node runs to completion, starting with the neighbor of m, then the neighbor of the neighbor of m, and so on, and finally m.

Two-phase commit (2PC). 2PC is a protocol for collectively deciding on *committing* or *aborting* a transaction. The protocol consists of a coordinator and n participants, and proceeds in two phases. During the first phase, the coordinator sends vote requests to all the participants and collects their votes, which can indicate to either commit or abort the transaction. If all of the participants have voted for committing the transaction, the coordinator initiates the second phase by sending *commit* messages to all participants. Otherwise, it sends an

abort message. When the participants receive the decision message from the coordinator, they finalize the transaction.

We consider an optimized and realistic implementation of the 2PC protocol. First, in both phases, the coordinator broadcasts a message and then waits to receive responses. Second, the coordinator can send "*early abort*" messages. While receiving votes, it can terminate the first phase and abort the transaction as soon as it receives a negative vote, without waiting for the remaining votes. Thus some of the participants might receive a decision message even before seeing a request. Therefore, the last optimization is that the participants can process request and decision messages concurrently, in contrast to processing the decision message sequentially after the request message.

We verified that all participants consistently commit or abort a transaction, and that commit only happens if all participants voted for commit. We established a sequential reduction of 2PC using 4 applications of IS (each IS application enlarging the sequentialized prefix until removing asynchrony altogether). The sequential reduction follows the natural flow of the protocol: broadcasting vote request messages, followed by vote responses from a nondeterministic number of participants, followed by the broadcast of the decision message, and the finalization of the transaction.

6 Related Work

We review works concerning the design of proof systems for reasoning about concurrent or distributed systems. We focus first on proof systems that include some form of reduction, i.e., behavior-preserving transformations that reduce the number of interleavings, which are closer to our work, and subsequently discuss other related works.

Reduction. Lipton's reduction theory [34] introduced the concept of *movers* to define a program transformation that creates atomic blocks of code. QED [15] expanded the scope of Lipton's theory by introducing iterated application of reduction and abstraction over gated atomic actions. CIVL [23] builds upon the foundation of QED, adding invariants [24, 38], refinement layers [27], and pending asyncs [28]. Inductive sequentialization builds upon this prior work, introduces a new scheme for reasoning inductively over unbounded concurrent executions, and thus provides an alternative to the classic approach of inductive invariants.

The work described above takes the general approach of reasoning about concurrent programs via simplifying program transformations. Recent research projects have advocated the need to incorporate an increasing set of sound program transformations. CSPEC [9] takes an approach similar to CIVL but mechanizes all metatheory within the Coq theorem prover [45] for flexibility and sound extensibility. Armada [35] also has flexible and mechanized metatheory whose usefulness is demonstrated by implementing a variety of program transformations, including those catering to fine-grained concurrency and weak memory models.

Movers have also been used to define an equivalence-preserving transformation that eliminates buffers in message-passing programs [2, 46]. These works define a restricted class of programs and prove that reasoning about the set of *rendezvous* executions of these programs, where messages are delivered instantaneously, is complete, i.e., any other execution is equivalent to a rendezvous execution, up to reordering of mover actions. Our example programs in Section 5 fall outside this class, e.g., because of ring topology (Chang-Roberts), optimizations (2PC, Paxos), or loop-carried state (Ping-Pong). In general, removing message buffers does *not* necessarily lead to a sequential program. Concurrency can still be present due to the different orders in which messages can be sent or received by different processes. For instance, von Gleissenthall et al. [46] consider a simpler variation of Paxos where the communication between a proposer p and an acceptor a_1 does not interleave with the communication between p and another acceptor a_2. The reduction to rendezvous communication, which remains a *concurrent* program, still contains all the complexity due to acceptors receiving messages from different rounds in an arbitrary order (which is not present in our sequentialization).

In the context of asynchronous programs, Kragl et al. [28] use left movers to derive atomic action summaries for procedures with asynchronous calls, i.e., they define a behavior-preserving transformation where asynchronous calls can be assumed to be synchronous provided that their body is a left mover. Inductive sequentialization solves the orthogonal problem of eliminating an unbounded number of PAs from atomic actions using induction. In particular, the work in [28] does not apply to the examples presented in Section 5 (their versions of Ping-Pong and two-phase commit do not model explicit communication through message-passing).

In the context of message-passing programs, Elrad and Francez's reduction theory [16] introduced the concept of *communication-closed layer*, which is a sequence of actions where every send action is paired with a corresponding receive action. They propose a program transformation that reduces a given program to a sequence of communication-closed layers. This simplifies reasoning since the lifetime of a message is limited to a single layer. Damian et al. [12] provides a concrete instantiation of this theory in the context of fault-tolerant distributed protocols that relies on common implementation idioms. While the result of this transformation is not a purely sequential program as in our case, it does provide a significant reduction in the number of schedules to reason about. Conceptually, our work is phrased in a more generic setting that does not rely on the specifics of the input program. The approach of Damian et al. [12] requires low-level annotations about local variables and messages, and various syntactic constraints on executions. For

instance, Damian et al. [12] cannot deal with Chang-Roberts or our 2PC with "early-abort" (independently of the programming model) because of syntactical constraints on the executions (see Condition V of Definition 2 in that paper). Chang-Roberts is not admitted because the messages do not encode a notion of time and 2PC is not admitted because the coordinator interleaves computation steps (taking a decision) with receiving votes. They can also not deal with the other examples (including our version of Paxos) because they are written using asynchronous procedure calls (Damian et al. [12] deals with protocols written as the composition of a number of long-running processes executing sequential code). Concerning Paxos, Damian et al. [12] considered several optimized variations which we believe are in the reach of IS as well. Given the limited time, we chose to evaluate IS over a diverse set of communication patterns and specifications instead of additional Paxos features.

Verification of distributed systems. There are several recent papers on mechanized verification of distributed systems. IronFleet [22] embeds TLA-style state-machine modeling [32] into the Dafny verifier [33] to refine high-level distributed systems specifications into low-level executable implementations. Ivy [40] 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. Padon et al. [39] presents a methodology for (manually) instrumenting program code which ensures that the verification conditions generated by Ivy fall into the decidable effectively-propositional fragment of first-order logic. Verdi [47] 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 Gavran et al. [19] and the ALS types of Kloos et al. [25] target a weaker form of asynchrony, where a single task queue atomically executes one task at a time. Unlike our approach, all the above perform asynchronous reasoning which significantly complicates the invariants. PSync [14] uses a synchronous model of communication for the purpose of program design and verification, shifting the complexity of efficient asynchronous execution to a runtime system.

Concurrent separation logic (CSL) [36] 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 [37] 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 [43] addresses the composition of verified protocols using ideas from separation logic. The actor services of [44] focus on compositional verification of response properties of message-passing programs.

Sequentialization in bounded model checking. Reducing concurrent program verification to sequential program verification has also been used in the context of bounded model checking, e.g., [4, 6, 17, 29, 30, 42]. In this case, the reduction encodes the control nondeterminism due to the interleaving semantics into data nondeterminism, and assumes a certain bound on interleavings, e.g., a bounded number of context switches [41]. The resulting sequential program still exhibits all the complexity due to interleavings, but is more amenable to symbolic reasoning using SMT solvers.

7 Conclusion

We presented inductive sequentialization, a new induction-based methodology for proving the correctness of an asynchronous program. This methodology establishes sequential reductions, which capture all the behaviors of the original program, up to reordering of commutative actions. The proofs using inductive sequentialization are much simpler than those relying on standard inductive invariants since they sidestep the problem of reasoning about arbitrarily many and arbitrarily long interleavings.

IS is a blend of induction, reduction and abstraction, which derives its power from the tight combination of the three. Its applicability is particularly enhanced in well-designed asynchronous systems which favor short-living asynchronous tasks in place of long-living tasks that reduce responsiveness, and where asynchrony is transparent in the sense that it does not affect the logic of the application. This has been demonstrated through the verification of a number of implementations of paradigmatic distributed protocols.

In the future we plan to further investigate the potential of IS to simplify the construction of formal proofs of distributed systems in other application areas, e.g., Byzantine fault tolerance, and blockchain protocols.

Acknowledgments

This research was supported in part by the Austrian Science Fund (FWF) under grant Z211-N23 (Wittgenstein Award) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

References

[1] 2020. Boogie. https://github.com/boogie-org/boogie
[2] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. In *OOPSLA*. https://doi.org/10.1145/3133934

[3] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*. https://doi.org/10.1007/11804192_17

[4] Ahmed Bouajjani and Michael Emmi. 2012. Bounded Phase Analysis of Message-Passing Programs. In *TACAS*. https://doi.org/10.1007/978-3-642-28756-5_31

[5] Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. 2017. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_7

[6] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. 2011. On Sequentializing Concurrent Programs. In *SAS*. https://doi.org/10.1007/978-3-642-23702-7_13

[7] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *CAV*. https://doi.org/10.1007/978-3-319-96142-2_23

[8] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. In *POPL*. https://doi.org/10.1145/3290342

[9] Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *OSDI*. https://www.usenix.org/conference/osdi18/presentation/chajed

[10] Ernest J. H. Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Commun. ACM* 22, 5 (1979). https://doi.org/10.1145/359104.359108

[11] Ching-Tsun Chou and Eli Gafni. 1988. Understanding and Verifying Distributed Algorithms Using Stratified Decomposition. In *PODC*. https://doi.org/10.1145/62546.62556

[12] Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. 2019. Communication-Closed Asynchronous Protocols. In *CAV*. https://doi.org/10.1007/978-3-030-25543-5_20

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2009. Generalized, efficient array decision procedures. In *FMCAD*. https://doi.org/10.1109/FMCAD.2009.5351142

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

[15] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *POPL*. https://doi.org/10.1145/1480881.1480885

[16] Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982). https://doi.org/10.1016/0167-6423(83)90013-8

[17] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *POPL*. https://doi.org/10.1145/1926385.1926432

[18] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *PLDI*. https://doi.org/10.1145/781131.781169

[19] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. 2015. Rely/Guarantee Reasoning for Asynchronous Programs. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2015.483

[20] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building certified concurrent OS kernels. *Commun. ACM* 62, 10 (2019). https://doi.org/10.1145/3356903

[21] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *PLDI*. https://doi.org/10.1145/3192366.3192381

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

[23] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*. https://doi.org/10.1007/978-3-319-21668-3_26

[24] Cliff B. Jones. 1983. Specification and Design of (Parallel) Programs. In *IFIP Congress*.

[25] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous Liquid Separation Types. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.396

[26] Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive Sequentialization of Asynchronous Programs (Evaluated Artifact). https://doi.org/10.5281/zenodo.3754772

[27] Bernhard Kragl and Shaz Qadeer. 2018. Layered Concurrent Programs. In *CAV*. https://doi.org/10.1007/978-3-319-96145-3_5

[28] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2018.21

[29] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *CAV*. https://doi.org/10.1007/978-3-642-02658-4_36

[30] Akash Lal and Thomas W. Reps. 2008. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *CAV*. https://doi.org/10.1007/978-3-540-70545-1_7

[31] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998). https://doi.org/10.1145/279227.279229

[32] Leslie Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.*

[33] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR*. https://doi.org/10.1007/978-3-642-17511-4_20

[34] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975). https://doi.org/10.1145/361227.361234

[35] Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: Low-Effort Verification of High-Performance Concurrent Programs. In *PLDI*. https://doi.org/10.1145/3385412.3385971

[36] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007). https://doi.org/10.1016/j.tcs.2006.12.035

[37] Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. https://doi.org/10.4204/EPTCS.211.7

[38] Susan S. Owicki and David Gries. 1976. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM* 19, 5 (1976). https://doi.org/10.1145/360051.360224

[39] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. In *OOPSLA*. https://doi.org/10.1145/3140568

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

[41] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *TACAS*. https://doi.org/10.1007/978-3-540-31980-1_7

[42] Shaz Qadeer and Dinghao Wu. 2004. KISS: keep it simple and sequential. In *PLDI*. https://doi.org/10.1145/996841.996845

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

[44] Alexander J. Summers and Peter Müller. 2016. Actor Services - Modular Verification of Message Passing Programs. In *ESOP*. https://doi.org/10.1007/978-3-662-49498-1_27

[45] The Coq Development Team. 2020. The Coq Proof Assistant, version 8.11.0. https://doi.org/10.5281/zenodo.3744225

[46] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. In *POPL*. https://doi.org/10.1145/3290372

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