Synchronizing the Asynchronous

Thomas A. Henzinger and Bernhard Kragl and Shaz Qadeer

Technical Report No. IST-2017-853-v1+1
Deposited at 03 Aug 2017 22:34
https://repository.ist.ac.at/853/1/main.pdf

Synchronizing the Asynchronous

Preprint, 4 August 2017

THOMAS A. HENZINGER, IST Austria
BERNHARD KRAGL, IST Austria
SHAZ QADEER, Microsoft Research

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 asynchronous computation threads. We show that specifications and correctness proofs for asynchronous programs can be structured by introducing the fiction, for proof purposes, that intermediate, non-quiescent states of asynchronous operations can be ignored. Then, the task of specification becomes relatively simple and the task of verification can be naturally decomposed into smaller sub-tasks. The sub-tasks iteratively summarize, guided by the structure of an asynchronous program, the atomic effect of non-atomic operations and the synchronous effect of asynchronous operations. This structuring of specifications and proofs corresponds to the introduction of multiple layers of stepwise refinement for asynchronous programs. We present the first proof rule, called *synchronization*, to reduce asynchronous invocations on a lower layer to synchronous invocations on a higher layer. We implemented our proof method in CIVL and evaluated it on a collection of benchmark programs.

1 INTRODUCTION

Concurrent programs are difficult to reason about because concurrently executing activities could be arbitrarily interleaved. Programmers are often unable to comprehend the enormous set of behaviors and consequently make mistakes leading to bugs that are difficult to identify, reproduce, and fix. Automatic verification tools would be tremendously valuable to the programmer in the process of iterative design and debugging. Full automation, however, remains an elusive goal because the state spaces of even moderately realistic concurrent programs are extremely large and often unbounded. Systematic testing of concurrent interleavings [11, 25, 46, 51] scales to realistic programs and is useful for finding bugs, but progressive increase in test coverage requires prohibitively high amounts of computing resources. Static analysis uses abstractions to search for proofs but has demonstrated only modest scalability for concurrent programs. This paper pursues *computer-assisted deductive verification* [40, 50], an alternative approach that promises to be more scalable through user involvement in the invention of proof outlines, while the justification for individual proof steps remains fully automated.

The central concept in deductive verification is the *inductive invariant*, an assertion that must hold in the initial state and which is preserved by all transitions of the program. An inductive invariant, if it can be expressed compactly, is an amazing artifact that concisely justifies the correctness of an enormous set of behaviors. Deductive verification, despite its elegance, has not found favor with researchers focused on automated verification for two reasons. First, verification conditions that capture the correctness of an inductive invariant are often expressed in logics [9] whose complexity is high, ranging from NP-complete to undecidable. Second, the programmer is required to invent the inductive invariant, a task that is considered too difficult for most programmers. This is especially the case for concurrent programs, whose invariants must capture all possible interleavings and interferences of concurrently executing processes. The first concern is being addressed adequately by advances in satisfiability solving and automated theorem proving. This paper addresses the latter concern by designing a mode of programmer-verifier interaction that reduces the intellectual effort of specifying inductive invariants for concurrent programs and leads to a more scalable proof methodology.

This paper focuses on *asynchronous concurrent programs*, a large class that includes distributed fault-tolerant protocols, message-passing programs, client-server applications, event-driven mobile applications, workflows, device drivers, and embedded and cyber-physical systems. An important 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 much more difficult. The behavior of a *synchronously-completing* 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 *asynchronously-completing* operation is much harder to specify because multiple operations could be in flight at the same time and partial results from one operation may have already affected the state before the asynchronously-completing operation finishes.

The traditional deductive verification approach to reasoning about asynchronous concurrent systems is to model the operational semantics as a flat transition relation over the entire state of the program, including shared variables, message buffers, and local states of concurrently executing processes. The correctness specification is modeled as a safety predicate that must be satisfied by all reachable states. Finally, an inductive invariant is invented to justify the correctness of the safety assertion. This approach leads to inductive invariants that are complicated for two reasons. First, the invariant must characterize all reachable states, even those where operations are executed partially. Second, modeling the operational semantics as a flat transition system leads to a huge amount of case analysis in the invariant, which makes it tedious to write and brittle in the face of program modification.

We attack the problem guided by three principles. First, we model an asynchronous program not as a flat transition system but as a program in a structured asynchronous programming language. This language provides syntax both for the usual sequential programming constructs and for asynchronous task creation. Second, we focus not on reachable states but on reachable quiescent states where all spawned tasks have finished execution. It is significantly easier to specify the correct behavior for quiescent states than for all intermediate states. Our safety specification is a single-state predicate constraining the initial states of the program (similar to a precondition) and a two-state predicate specifying the relationship between an initial state and any quiescent state that may result from it (similar to a postcondition). Finally, we exploit syntactic features of our programming language to simplify the programmer-computer interaction fundamental to a deductive proof. Instead of using a monolithic inductive invariant to justify the correctness of our safety specification, we perform the proof by using program layers. Two successive program layers are connected by a syntax-directed program transformation that simplifies the program while preserving quiescent states; thus proving our safety specification on the final program also proves it for the initial program. Instead of writing a monolithic invariant justifying the safety correctness specification, we write a sequence of simple invariants to justify the correctness of program transformations.

The main novel contribution of our work is a new program transformation called *synchronization*, a generalization of reduction [21, 42] targeted towards asynchronous programs. 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 arbitrary asynchronous computation, executed by a potentially unbounded number of concurrently-executing threads. Synchronization reduces the number of interleavings that need to be considered; it allow us to pretend, for the purposes of proof, that asynchronous calls complete synchronously and atomically, which leads to significantly simpler invariants.

The proof of soundness for the synchronization transformation is nontrivial. In addition to the usual commutativity conditions required for reduction, it also requires imposing a different *cooperation* condition on the asynchronous code block that is being synchronized. In essence, cooperation requires that every synchronous execution of the synchronized code block *can* terminate, i.e., there *exists* a terminating execution. This prevents a program failure that is possible in the asynchronous program to be hushed up by nonterminating (but safe) executions of the synchronized program. We provide a reduction from the problem of checking cooperation to a safety and termination check on a reduced sequential program.

We have implemented our proof method atop CIVL [29], a modular and automated refinement verifier. CIVL already implements reduction; we extended this implementation to support synchronization. By combining synchronization with another transformation called *abstraction*, we provide the capability to convert an asynchronous code block into a single atomic action. By iterating these transformations, guided by the program structure, we obtain a structured proof that alternates the inlining of asynchronous calls (synchronization) with the summarization of synchronous and atomic behavior (abstraction). But the program transformations do not come for free: each transformation imposes proof obligations about the commutativity of atomic actions, which are discharged by automatic provers.

From an annotated input program, our implementation automatically generates verification conditions that encode program correctness. Each verification condition concerns only a local part of the program, which is important in two ways. First, the verification conditions are small and thus automatically discharged by an SMT solver. Second, a failed proof obligation gives directed feedback and is easily attributed to a bug in the program or a bug in the proof attempt.

We successfully experimented with several realistic benchmarks, including a verified implementation of the two-phase commit protocol (2PC).

2 MOTIVATING EXAMPLES

In this section we illustrate our new verification technique based on the synchronizing program transformation and the associated proof obligations on three simple examples. We use the notation `[...]` to denote atomic actions, i.e., the statements inside square brackets are considered to execute indivisibly.

2.1 Example 1: Synchronizing asynchronous calls

Consider the program in Figure 1. The program comprises a single procedure `Main` that uses a global variable `x` and a local variable `i`. The first while loop creates a hundred new threads, each invoking the atomic action `[x := x + 1]`, that atomically increments `x` by one. The second while loop is similar, except that the hundred created asynchronous threads invoke the atomic action `[x := x - 1]`, that atomically decrements `x` by one. Due to asynchronous thread creation, the execution of individual instances of increments and decrement can be arbitrarily interleaved. However, since there are equally many increments and decrements, we know that after all threads terminated, the value in variable x is equal to its initial value. Thus, the "initial-to-quiescent-state" behavior of `Main` is the same as that of the atomic action `[skip]`.

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 asynchronous increment threads that already terminated, minus the number of asynchronous decrement threads that already terminated". Furthermore, additional ghost code has to be introduced to keep track of the progress of each thread. By contrast, our new proof rule allows us to wrap the whole body of `Main` into an atomic block in which all asynchronous calls are replaced with synchronous calls. From there it is easy to prove that a hundred increments followed by a hundred decrements leave the variable x unchanged. Thus, we reduced the reasoning about a complicated asynchronous program to reasoning about a simple sequential program. How is that possible? Our key technique is the extension of commutativity-based reduction [42] to asynchronous programs. In particular, the atomic actions that are invoked due to asynchronous calls have to be *left movers*; they have to commute to the left, i.e., earlier in time, with respect to all atomic actions in the program. This ensures that all asynchronous interleaved executions can be summarized by a synchronous execution, without losing any failing or terminating behaviors. In `Main`, both the increment and decrement action commute with themselves and each other. Thus, our transformation is sound.

```
global var x : int
local  var i : int

Main {
  i := 0
  while (i < 100) {
    async [x := x + 1]
    i := i + 1
  }
  i := 0
  while (i < 100) {
    async [x := x - 1]
    i := i + 1
  }
}
```

Fig. 1. Asynchronous increments and decrements.

2.2 Example 2: Termination

Consider the program in Figure 2. This is a recursive formulation of the first loop in the previous example, creating a hundred asynchronous instances of atomic increments to x. Using a similar commutativity argument as before, our new proof rule would allow us to rewrite the asynchronous calls to `RecInc` into synchronous ones, would it not be for the bug that the parameter passed to `RecInc` is `i`, instead of `i-1`.

What would happen if our proof rule could be applied to this program? In the original program, there is an assertion after the call to `RecInc` in `Main`, which fails any execution that reaches that location. Since asynchronous invocations do not block, there are certainly failing executions of the original program. However, in the synchronized program `RecInc` never terminates due to the bug in the recursive call. The assertion in `Main` never gets executed in a synchronous execution, and thus the program is safe. Applying our proof rule would be unsound.

```
global var x : int
local  var i : int

RecInc (i : int) {
  if (i > 0) {
    call [x := x + 1]
    async RecInc(i) // bug
  }
}

Main {
  async RecInc(100)
  assert false
}
```

Fig. 2. Non-terminating recursion.

Note that nonterminating executions are not a problem per se; the original asynchronous program also has a nonterminating execution where the assertion gets postponed forever. The problem is that the nontermination of the atomic and synchronized block we are introducing effectively *blocks* executions; it gets stuck in the atomic block and thus masks failures of the original program that would occur after the block finishes. A simple approach to remedy the situation is to always require the termination of the atomic and synchronized block we are introducing. This approach is sound and works for the current example (i.e., if the bug is fixed, then the condition holds). However, as we will show in the next example, requiring the termination of all executions is too restrictive.

2.3 Example 3: Cooperation

Finally, consider the program in Figure 3. It is a simple agreement protocol between two processes A and B. The global variables model the internal states of the processes; `val_a` is an internal value of process A, and `val_b` is an internal value of process B. The goal of the processes is to agree on the same value. In `Main`, process A initiates the protocol by sending a proposal to process B, modeled as an asynchronous call to the corresponding message handler `propose_by_a` of B. Not that this modeling choice corresponds to message-passing in an unreliable network, that might drop or reorder messages. Upon the receipt of the proposal, process B nondeterministically chooses to either accept the value proposed by A and send back an acknowledgment message (`ack_by_b`), or to reject the value and propose a new value of its own to A (`propose_by_b`). The implementations of `propose_by_b` and `ack_by_a` are symmetric and not shown in Figure 3.

```
global var val_a : int
global var val_b : int

Main {
  async propose_by_a(val_a)
}

propose_by_a (val) {
  if (*) {
    call [val_b := val]
    async ack_by_b()
    assert val_a == val_b
  } else {
    call [havoc val_b]
    async propose_by_b(val_b)
  }
}

ack_by_b() {
  assert val_a == val_b
}
```

Fig. 3. Simple agreement protocol.

We want to apply our new proof rule to make the protocol invocation in `Main` atomic and, at the same time, transform all asynchronous calls into synchronous calls. Recall that our justification for introducing atomic blocks is based on commutativity theory. In particular, asynchronously invoked atomic actions need to be left movers. Since the communication in this example is a simple back and forth, it is easy to show that no two atomic actions can be about to execute at the same time. We do not go into detail and turn our attention to the termination issue. Obviously, we cannot show termination for this program; there is a non-terminating execution where both processes keep on proposing new values to each other. However, non-terminating executions per se do not harm the soundness of our program transformation. We just have to make sure that failures of the original program are preserved in the transformed program. Thus our *cooperation condition* requires that *for all* partial executions of the atomic and synchronized block we are introducing, there *exists* a terminating or failing extension. That is, we do not require general termination, but the possibility to terminate. In our example it is easy to see that each process can always take the if-branch when processing a proposal which leads to the termination of the protocol. The situation is very similar in many other asynchronous programs, in particular distributed protocols. There are corner cases of adversarial executions that cause indefinite retries, but the common case is an immediate resolution. For example, in the Paxos consensus protocol there is a tiny chance that several proposers compete indefinitely to get their proposals accepted, always invalidating their competitors proposals. However, the common case is a single proposer that gets incoming transactions accepted on the first attempt. This is exactly the intuition we exploit in practice to establish our cooperation condition. The programmer provides a restriction of the nondeterminism in the program and then we perform a standard termination check on the restricted program. Note that this termination check is not on a concurrent program anymore. It only concerns the modified part of the program, which is now sequential.

3 AN ASYNCHRONOUS PROGRAMMING LANGUAGE

In this section we introduce an asynchronous programming language, its operational semantics, and the notion of refinement between programs.

Variables and stores. Let $\mathcal{V}$ be a set of *variables* partitioned into *global variables* $\mathcal{V}_G$ and *local variables* $\mathcal{V}_L$. Furthermore, there is a set of *return variables* $\mathcal{V}_R \subseteq \mathcal{V}_L$. 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*. We will use the notation $g \cdot \ell$ to denote both the combination of g and ℓ into a store, as well as the separation of a store into g and ℓ. 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 $\ell_1 \rhd \ell_2$ at the caller as

$$\ell_1 \rhd \ell_2(v) = \begin{cases} \ell_1(v) & \text{if } v \in \mathcal{V}_R \\ \ell_2(v) & \text{if } v \notin \mathcal{V}_R \end{cases} .$$

Atomic actions. An *atomic action* is a pair (ρ, α), where the *gate* ρ is a predicate over stores and the *action* α is a (transition) relation over stores [17]. In an execution, the invocation of an atomic action first evaluates the gate in the current store. If ρ evaluates to *false*, the execution fails. Otherwise, the store is updated according to α. We will write $\sigma \xrightarrow{\alpha} \sigma'$ for $(\sigma, \sigma') \in \alpha$.

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 following form.

$$s ::= \texttt{skip} \mid s;s \mid \texttt{if } le \texttt{ then } s \texttt{ else } s \mid \texttt{while } le \texttt{ do } s \mid \texttt{call } A \mid \texttt{call } P \mid \texttt{async } P \mid \texttt{atomic } \rho\ s$$

In addition to the standard constructs (skip, sequencing, conditionals, and loops), the language supports the invocation of *atomic actions*, the *synchronous* and *asynchronous* invocation of *procedures*, and the formation of *atomic blocks*. A program is *well-formed* if (1) it has a dedicated main procedure *Main* that serves as an entry point for executions, and (2) every atomic action name respectively procedure name appearing in a call statement is properly mapped to an atomic action respectively statement. We assume that all programs are well-formed. We will use dot notation and write $\mathcal{P}.A$ and $\mathcal{P}.P$ instead of $\mathcal{P}(A)$ and $\mathcal{P}(P)$. We denote by $\mathit{MaybeInvoked}(\mathcal{P})$ the set of procedure and atomic action names that are called (directly or indirectly, synchronously or asynchronously) from *Main*.

Atomic blocks play a special role in our formalization. Similar to an atomic action, an atomic block `atomic` $\rho\ s$ has a gate ρ that needs to be satisfied in order to execute the block. If so, the statement s is executed to completion without preemption from any other thread. In this sense, an atomic block is a precursors to an atomic action that still contains the details of sequential execution steps. In Section 4 we present a program transformation to rewrite a statement into an atomic block, and in Section 7 we show how to convert an atomic block into an atomic action. This separates the task of coarsening atomicity from the task of summarizing sequential execution. Thus, for technical convenience, we assume that the original input program does not contain atomic blocks, and that the introduction of an atomic block is always followed by the conversion into an atomic action.

Sequential step $\longrightarrow$

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

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

$$\frac{(\ell \in le \land s' = s_1) \lor (\ell \notin le \land s' = s_2)}{(g,\ TC[\ell][\texttt{if}\ le\ \texttt{then}\ s_1\ \texttt{else}\ s_2]) \longrightarrow (g,\ TC[\ell][s'])}\ \textsc{If} \qquad \frac{(\ell \in le \land s' = s; \texttt{while}\ le\ \texttt{do}\ s) \lor (\ell \notin le \land s' = \texttt{skip})}{(g,\ TC[\ell][\texttt{while}\ le\ \texttt{do}\ s]) \longrightarrow (g,\ TC[\ell][s'])}\ \textsc{While}$$

$$(g,\ TC[\ell][\texttt{call}\ P]) \longrightarrow (g,\ (\ell,\ \mathcal{P}.P){\cdot}TC[\ell][\texttt{skip}])\ \textsc{Call} \qquad (g,\ (\ell_1,\ \texttt{skip}){\cdot}TC[\ell_2][s]) \longrightarrow (g,\ TC[\ell_1 \triangleright \ell_2][s])\ \textsc{Return}$$

Concurrent step $\Longrightarrow$

$$\frac{(g,\ t) \longrightarrow (g',\ t')}{(g,\ \mathcal{T} \uplus t) \Longrightarrow (g',\ \mathcal{T} \uplus t')}\ \textsc{Step} \qquad \frac{g{\cdot}\ell \in \rho \quad (g,\ (\ell,\ s)) \longrightarrow^* (g',\ (\ell',\ \texttt{skip}))}{(g,\ \mathcal{T} \uplus TC[\ell][\texttt{atomic}\ \rho\ s]) \Longrightarrow (g',\ \mathcal{T} \uplus TC[\ell'][\texttt{skip}])}\ \textsc{AtomicStep}$$

$$\frac{(g,\ t) \longrightarrow \lightning}{(g,\ \mathcal{T} \uplus t) \Longrightarrow \lightning}\ \textsc{Fail} \qquad \frac{g{\cdot}\ell \notin \rho \lor (g,\ (\ell,\ s)) \longrightarrow^* \lightning}{(g,\ \mathcal{T} \uplus TC[\ell][\texttt{atomic}\ \rho\ s]) \Longrightarrow \lightning}\ \textsc{AtomicFail}$$

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

Fig. 4. Small-step operational semantics.

The logical expression *le* in conditionals and loops is not allowed to access global variables. We identify *le* with the set of local stores that satisfy the expression. Thus, we write $\ell \in le$ if *le* evaluates to *true* under ℓ, and $\ell \notin le$ otherwise. All updates to variables, and, in particular, accesses to global variables, are encapsulated within atomic actions.

Semantics. A *frame* f is a pair (ℓ, s) of a procedure-local store ℓ together with a statement s that remains to be executed. 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 $\mathcal{T}$. We present our semantics as reduction semantics [20] and define *statement contexts SC*, *frame contexts FC*, and *thread contexts TC* as follows.

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

The contexts are nested and a top-level thread context contains two unique holes $\bullet$ that can be filled with a statement and a local store, respectively. We write $TC[\ell][s]$ to denote the thread obtained by filling the respective holes with ℓ and s. Intuitively, this thread has the frame (ℓ, s) on the top of its stack. The operational semantics is formalized as the sequential transition relation $\longrightarrow$ and the concurrent transition relation $\Longrightarrow$ according to the rules in Figure 4. An *execution* π is a sequence of states $x_0 \Longrightarrow x_1 \Longrightarrow \dots$, and we write $\pi : x_0 \Longrightarrow^* x_n$ to denote that π is an execution that starts in x_0 and ends in x_n.

Remark 3.1. For the remainder of the paper we assume that every statement s in a program $\mathcal{P}$ has an implicit unique identity. Thus, (a) any two different occurrences of syntactically equal statements are distinguished, and (b) every occurrence of a statement in an execution can be attributed to a unique syntactic statement in the program.

Specifications. 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 $(g, (\ell, \mathcal{P}.Main))$.

In particular, we are interested in executions that either fail or terminate. We define $Bad(\mathcal{P})$ to be the set of initial stores associated with *failing executions*, and $Good(\mathcal{P})$ to be the relation between initial stores and final global stores associated with *terminating executions*. Formally,

$$Bad(\mathcal{P}) = \{g{\cdot}\ell \mid (g, (\ell, \mathcal{P}.Main)) \Rightarrow^* \lightning\} \qquad Good(\mathcal{P}) = \{(g{\cdot}\ell, g') \mid (g, (\ell, \mathcal{P}.Main)) \Rightarrow^* (g', \varnothing)\}$$

A program $\mathcal{P}_1$ *refines* a program $\mathcal{P}_2$, denoted $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$, if

(1) $\overline{Bad(\mathcal{P}_1)} \subseteq Bad(\mathcal{P}_2)$ and
(2) $\overline{Bad(\mathcal{P}_2)} \circ Good(\mathcal{P}_1) \subseteq Good(\mathcal{P}_2)$.

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 $\mathcal{P}_1$.

Lemma 3.2. *If $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$ and $\mathcal{P}_2 \preccurlyeq \mathcal{P}_3$, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}_3$.*

Program transformations. In the following sections we define program transformations $\mathcal{P} \rightsquigarrow \mathcal{P}'$ that transform program $\mathcal{P}$ into $\mathcal{P}'$. For convenience, we will sometimes write transformations $s \rightsquigarrow s'$ between statements to mean the transformation of a procedure body from s into s', i.e.,

$$\frac{s \rightsquigarrow s'}{\mathcal{P} \cup [P \mapsto s] \rightsquigarrow \mathcal{P} \cup [P \mapsto s']} \ .$$

In order to conveniently specify rewrite rules on statements we define *rewrite contexts* RC as follows.

$$\begin{aligned} RC ::= {} & \bullet_{Stmt} \mid RC; s \mid s; RC \mid \texttt{if}\ le\ \texttt{then}\ RC\ \texttt{else}\ s \mid \texttt{if}\ le\ \texttt{then}\ s\ \texttt{else}\ RC \\ & \mid \texttt{while}\ le\ \texttt{do}\ RC \mid \texttt{atomic}\ \rho\ RC \end{aligned}$$

4 REDUCTION OF UNBOUNDED ASYNCHRONOUS COMPUTATIONS

Our main novel contribution in this paper is a new program transformation called *synchronization*, a generalization of reduction [21, 42] targeted towards asynchronous programs. While reduction classically 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 arbitrary asynchronous computation, executed by a potentially unbounded number of concurrently-executing threads. More precisely, synchronization allows to rewrite a statement s into `atomic` $\rho\ s'$, where s' is completely synchronous.

The soundness of the synchronization transformation requires two technical innovations. First, we extend the usual *commutativity* conditions required for reduction to account for asynchronous thread creation. Second, synchronization requires to impose a *cooperation* condition on s', that guarantees that partial sequential executions of s' can be completed. In this section we state and prove the correctness of synchronization w.r.t. abstract formulations of both conditions. Then, in Section 5 we present a type system to establish the commutativity conditions, and in Section 6 we show how cooperation can be established via a standard safety and termination check on a reduced sequential program.

Synchronization. For a statement s with potential asynchronous invocations we want to define a synchronous version s'. Since asynchronous invocations can happen indirectly, we must not only rewrite asynchronous calls in s, but also introduce synchronized versions of the procedures called by s. For a program $\mathcal{P}$, let f be a mapping between procedure names. Then for a statement s we define $Sync_f(s)$ to be the statement equal to s, except that every `call` P and `async` P is replaced with `call` $f(P)$. The mapping f is called a *synchronization function*, if for every procedure P, $\mathcal{P}.f(P) = Sync_f(\mathcal{P}.P)$. Note that for convenience we defined f to provide a synchronized version of every procedure, although our program transformation only uses the ones actually called from s.

Remark 4.1. Since asynchronous calls cannot return values to their callers, we assume that all asynchronously called procedures do not modify any return variable in $\mathcal{V}_R$. This allows us to safely execute a synchronized call directly (i.e., synchronously) in the thread of the caller, without its local variables being corrupted.

Definition 4.2. Let $\mathcal{P}$ be a program without any atomic blocks, and let f be a synchronization function for $\mathcal{P}$. Then the synchronization rule allows to make a statement s in $\mathcal{P}$ atomic and synchronous as follows.

$$\frac{s' = \mathit{Sync}_f(s) \quad \mathit{Atomic}(s, \mathcal{P}) \quad \mathit{Cooperative}(\rho, s')}{\mathit{RC}[s] \rightsquigarrow \mathit{RC}[\texttt{atomic}\ \rho\ s']}\ \textsc{Synchronize}$$

Let $\mathcal{P}, s, s', \rho, f$ be fixed for the remainder of this section. Recall that, by Remark 3.1, it is always possible to distinguish the execution steps that are due to the particular rewritten occurrence of statement s from those that are due to potentially other occurrences of s in the program. In the following we will only refer to the former. The soundness of our synchronizing reduction rule relies on two proof obligations.

The first proof obligation $\mathit{Atomic}(s, \mathcal{P})$ states that the statement s has to be *atomic* in the context of program $\mathcal{P}$, which is based on commutativity theory and the notion of left mover and right mover originally coined by Lipton [42]. Intuitively, an atomic action is called a *right mover*, if it commutes to the right (i.e., later in time) with respect to all other atomic actions in $\mathcal{P}$. Analogously, an atomic action is called a *left mover*, if it commutes to the left (i.e., earlier in time) with respect to all other atomic actions in $\mathcal{P}$. An atomic action can be both a left and right mover. Concretely, for every $A_1, A_2 \in \mathcal{P}$ the following conditions need to hold [30] (see Appendix A for complete formalization).

- **Commutativity:** If A_1 is a right mover or A_2 is a left mover, then the effect of executing A_1 followed by A_2 in two different threads can be achieved by executing A_2 followed by A_1.
- **Forward preservation:** If A_1 is a right mover or A_2 is a left mover, then the failure of A_2 immediately after the execution of A_1 is implies that A_2 must also fail before the execution of A_1.
- **Backward preservation:** If A_2 is a left mover (and A_1 is an arbitrary atomic action), then the failure of A_1 immediately before the execution of A_2 implies that A_1 must also fail immediately after the execution of A_2.
- **Nonblocking:** If A_2 is a left mover, then A_2 cannot block.

Let m be the mapping from atomic actions to their respective *mover type*: B (both mover), L (left mover), R (right mover), A (atomic non-mover). Note that every execution step other than the invocation of an atomic action is naturally a both mover, since it neither reads from nor writes to the global store.

Definition 4.3. $\mathit{Atomic}(s, \mathcal{P})$ holds, if the execution steps of every occurrence of s in every concurrent execution of $\mathcal{P}$ satisfy: (1) the sequence of mover types (including steps in asynchronously created threads) has the form $\mathrm{R}^*\mathrm{A}^?\mathrm{L}^*$, and (2) the sequence of mover types in every individual asynchronous thread has the form L^*.

Intuitively, condition (1) allows us to commute the steps in an execution such that s executes without interruption from other threads; other threads that existed before s started to execute and their child threads that were created during the execution of s. Condition (2) allows us to arrange asynchronous calls such that they take effect immediately. To obtain a sound refinement, we further need to establish a *cooperation* condition. Consider the situation where s was executed partially and then another thread in the environment failed. We need to show that this failure is preserved when

s executes atomically and synchronously (i.e., in an execution of s'). If all steps of s so far were right movers, they can be completely elided from the execution and thus the failure can occur before s' even starts. However, if s already executed some non-right mover, the cooperation condition states that there must be *some* possibility to first run s to completion, such that the failure can occur after s' finished. Thus, the cooperation condition prevents failing executions of the original program to be turned into blocking executions of s' in the synchronized program.

Definition 4.4. $\mathit{Cooperative}(\rho, s')$ holds, if every partial sequential execution of s' that starts in a store that satisfies ρ and already executed a non-right mover can be extended to a terminating or failing execution.

We will now state and prove the correctness of our new synchronization rule. Since both of our proof obligations are, in general, undecidable, we show in the following two Sections 5 and 6 how they can be efficiently checked in practice.

Theorem 4.5. *If $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ using the* Synchronize *rule, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}2$.*

Proof. $\mathit{Bad}(\mathcal{P}_1) \subseteq \mathit{Bad}(\mathcal{P}_2)$. Let $\pi_1 : (g, (\ell, \mathcal{P}_1.\mathit{Main})) \Rightarrow^* ↯$ be an arbitrary failing $\mathcal{P}_1$-execution. We show how to construct a failing $\mathcal{P}_2$-execution $\pi_2 : (g, (\ell, \mathcal{P}_2.\mathit{Main})) \Rightarrow^* ↯$. First, by condition (1) of $\mathit{Atomic}(s, \mathcal{P})$ we can rewrite π_1 into $\pi_1' : (g, (\ell, \mathcal{P}_1.\mathit{Main})) \Rightarrow^* ↯$, such that all steps of any top-level occurrence of s happen without any outside interleavings. This follows from a standard commutativity argument, where all right movers are moved to the right and all left movers are moved to the left, until they meet. Now we show that all asynchrony can be eliminated in π_1'. We rewrite π_1' stepwise into $\pi_1'' : (g, (\ell, \mathcal{P}_1.\mathit{Main})) \Rightarrow^* ↯$ by individually considering every occurrence of s from left to right and doing the following. If the store right before the start of s does not satisfy ρ, we are done. Otherwise, we consider every position from left to right at which there exists an unfinished thread created by s, and preserve the invariant that the portion to the left of the current position corresponds to a valid sequential execution. There is always a youngest thread (corresponding to the topmost frame in a synchronous execution) that we need to make the next step. If this thread makes the step at the current position, we move on. Otherwise there are two cases, either the thread has a step downstream in the execution or not. In the first case, this step has to be a left mover by condition (2) of $\mathit{Atomic}(s, \mathcal{P})$ and we can move it to the current position. In the second case, observe that we can apply $\mathit{Cooperative}(\rho, s')$ to the partial execution to the left of the current position to obtain a terminating of failing extension π_{ext}. Furthermore, all atomic actions in π_{ext} are left movers. We consider only the prefix π_{ext}' of π_{ext} that terminates the youngest thread. Because of the properties of left movers, we can insert all steps of π_{ext}' at the end of our concurrent execution, right before the failure, and commute them back to the concurrent position. Observe that there can be only a finite number of unfinished threads that we have to terminate like this. At this point we eliminated all asynchronous interleavings within s. Now we need to consider the case where the execution of s neither fails nor terminates. If the execution so far only invoked right moving atomic actions, we can commute them all the way to the right in the concurrent execution, over the final failure, and thus eliminate them from the execution altogether. Otherwise, we can apply $\mathit{Cooperative}(\rho, s')$ as before to obtain a failing or terminating extension of left movers that we can insert into the execution to complete s. Finally we obtain π_2 from π_1'' by simulating every step of s with a corresponding step in s'. Here we rely on the assumption from Remark 4.1.

$\overline{\mathit{Bad}(s_2)} \circ \mathit{Good}(s_1) \subseteq \mathit{Good}(s_2)$. Let $\pi_1 : (g, (\ell, \mathcal{P}_1.\mathit{Main})) \Rightarrow^* (g', \varnothing)$ be an arbitrary terminating $\mathcal{P}_1$-execution. We proceed exactly as above. If we do not introduce a failure, we obtain a terminating $\mathcal{P}_2$-execution $\pi_2 : (g, (\ell, \mathcal{P}_2.\mathit{Main})) \Rightarrow^* (g', \varnothing)$. Otherwise we obtain a failing $\mathcal{P}_2$-execution $\pi_2 : (g, (\ell, \mathcal{P}_2.\mathit{Main})) \Rightarrow^* ↯$. □

5 CHECKING *Atomic*$(s, \mathcal{P})$

Condition (1) of *Atomic*$(s, \mathcal{P})$ states that the sequence of mover types along every execution of s has to induce a path in the *atomicity automaton* $\mathcal{A}$ shown in Figure 5. Let $\mathcal{L}$ be the function that labels edges of $\mathcal{A}$ with mover types, and its inverse $\mathcal{L}^{-1}$ maps a mover type to the corresponding set of edges. We define a type system that assigns to every statement in $\mathcal{P}$ a set of edges in $\mathcal{A}$, corresponding to the locations at which it is allowed to appear in an execution. Recall that m is the mapping of atomic action names to their respective mover types. Furthermore, our type system is parameterized by a set C of synchronized loops and procedure names that will be subject to the cooperation check. Based on m and some C, let M be a solution to the typing equations in Figure 6. The greatest solution (w.r.t. $\subseteq$) can be obtained using standard fixed point iteration. Alternatively, procedures can be annotated with a fixed mover type and the annotation of each procedure is checked modularly.

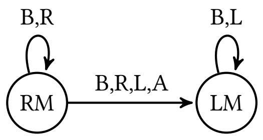

Fig. 5. Atomicity automaton $\mathcal{A}$.

$$M(\texttt{skip}) = \mathcal{L}^{-1}(\mathrm{B}) \qquad M(\texttt{call } A) = \mathcal{L}^{-1}(m(A))$$

$$M(s_1; s_2) = M(s_1) \circ M(s_2) \qquad M(\texttt{if } le \texttt{ then } s_1 \texttt{ else } s_2) = M(s_1) \cap M(s_2)$$

$$M(\texttt{while } le \texttt{ do } s) = \begin{cases} M(s) \cap \{\mathrm{RM}{\rightarrow}\mathrm{RM}, \mathrm{LM}{\rightarrow}\mathrm{LM}\} & \text{if } \mathit{Sync}_f(\texttt{while } le \texttt{ do } s) \in C \\ M(s) \cap \{\mathrm{RM}{\rightarrow}\mathrm{RM}\} & \text{if } \mathit{Sync}_f(\texttt{while } le \texttt{ do } s) \notin C \end{cases}$$

$$M(\texttt{call } P) = \begin{cases} M(\mathcal{P}.P) & \text{if } f(P) \in C \\ M(\mathcal{P}.P) \cap \{\mathrm{RM}{\rightarrow}\mathrm{RM}\} & \text{if } f(P) \notin C \end{cases}$$

$$M(\texttt{async } P) = \begin{cases} M(\mathcal{P}.P) \cap \{\mathrm{LM}{\rightarrow}\mathrm{LM}\} & \text{if } f(P) \in C \\ \varnothing & \text{if } f(P) \notin C \end{cases}$$

Fig. 6. Typing equations.

First, let us restrict our attention to the cases $\in C$. The equations propagate type information along the control flow of program statements, where calls to atomic actions are typed according to their mover type, and calls to procedures inherit the type of the body of the called procedure. For example, consider the equation for while loops. Since loop bodies are generally executed multiple times, we do not allow the body of a loop to act as the edge RM→LM, since this would allow a non-mover to be executed multiple times. There is no equation for atomic blocks, since we assumed that $\mathcal{P}$ does not contain any atomic blocks. If we can type s such that $M(s) \neq \varnothing$, we are guaranteed that the sequence of mover types along any execution of s has the form $\mathrm{R}^*\mathrm{A}^?\mathrm{L}^*$. Note that it is perfectly fine to type statements outside of s with $\varnothing$. Furthermore, the equation for asynchronous calls makes sure that all steps in threads created by s are left movers. Thus, our type system establishes *Atomic*$(s, \mathcal{P})$.

Theorem 5.1. *Let M be a solution to the equations in Figure 6 such that $M(s) \neq \varnothing$, then Atomic$(s, \mathcal{P})$.*

Now we turn our attention to the cases $\notin C$ in the typing equations, that enforce additional constraints on the typing of loops and calls. Loops and recursive calls are the potential sources of nonterminating behaviors. Since *Cooperative*(ρ, s') requires the existence of cooperating extensions to partial synchronous executions of s only if already some non-right mover took effect, we a free to type any loop or procedure call with $\{\mathrm{RM}{\rightarrow}\mathrm{RM}\}$. Then we are guaranteed that only right movers executed so far. However, if we want to assign a richer type to a loop or procedure call, they have

(a)
```
global var x : int

async [ x := x + 1 ]
call [ assert x == 1 ]
```

(b)
```
global var x : int

async [ x := x + 1 ]
call [ assert x <= 1 ]
```

(c)
```
global var x : int

call [ x := 0 ]
async [ x := x + 1 ]
```

Fig. 7. Synchronization is permitted in program (b), but not in (a) or (c).

to be included in the set C in order to be accounted for in the cooperation check described in the next section. Notice that the loops and procedures in C are already synchronized.

Example 5.2. Let us revisit our motivating example in Figure 1. In order to synchronize the asynchronous calls to the increment and decrement atomic actions, we have to check that they are left movers with respect to themselves and each other. Going though the list, we have commutativity (addition in general is commutative), forward and backward preservation (the gate of both atomic actions is *true*), and nonblocking (addition is enabled for all integers). Since both atomic actions are left movers, the asynchronous calls can be typed as {LM→LM}. Then both while loops also have to be typed as {LM→LM}, and thus their synchronizations have to be included in C for the cooperation check. Finally, sequential composition types the whole body of `Main` as {LM→LM}.

Example 5.3. To gain further intuition on the left mover requirement, let us discuss the applicability of synchronization in the programs in Figure 7. Program (a) asynchronously invokes an increment operation and asserts that `x` is equal to one in the gate of a second atomic action. Synchronizing the invocation of the increment in this program is not permitted, since it is not a left mover in the context of the program. In particular, backward preservation with respect to the assertion does not hold. Indeed, there is an asynchronous execution of the program with `x` initially zero that postpones the increment and fails, while the synchronous execution does not fail.

By contrast, in program (b) the gate is weaker and only asserts that `x` is less than or equal to one. Here the increment is a left mover and synchronization is applicable. It is an easy exercise to check that program (b) refines its synchronized version.

Finally, program (c) initializes the global variable `x` to zero before it asynchronously invokes the increment operation. Since increment does not commute with the initialization operation, it is not a left mover and synchronization is not allowed. In this program it is easy to see that this two atomic actions will never be concurrently enabled and about to execute. In other words, we would never have to commute an increment action to the left of an initialization action to recover a sequential execution from an asynchronous one. However, computing which atomic actions can execute concurrently is, in general, as undecidable as verification itself. Thus, primitive atomic actions usually have to be equipped with stronger gates to make them left movers. For example, if a shared variable is protected via a lock, the gate of an atomic action accessing the variable states that the lock must be held. In Section 8 we illustrate the use of permissions [8] to aid in commutativity checks.

6 CHECKING *Cooperative*(ρ, s')

Let π be a partial synchronous execution of s' that started in a store that satisfies ρ and already executed some non-right mover. We have to guarantee that there exists a terminating or failing extension of π. There are two crucial observations.

- First, there always exists some extension of π. This is because there are only left movers remaining to be execute, and those cannot block. Still, there is a difference between some or all extensions of π to be terminating or failing. This is because of nondeterminism, which

is confined to atomic actions in our programming language. However, if we can suitably restrict the nondeterminism to eliminate all nontermination, we can reduce the search for some terminating or failing extension to a standard termination check on the restricted program.
- Second, assume that there does not exist a terminating or failing extension of π; i.e., all extensions are nonterminating. Then in any such execution there must either occur a loop or procedure call from C that does not terminate. Thus it suffices to show the termination of all loops and procedures in C from the reachable states of s'.

Based on those two observations we reduce $\mathit{Cooperative}(\rho, s')$ to standard sequential safety and termination checks as follows. A *safety annotation Pre* is a mapping from C to state predicates, such that $\mathit{Pre}(x)$ always holds before the execution of x.

Definition 6.1. $\mathit{Safe}(\rho, s', \mathit{Pre})$ holds, if for every execution of s' that starts in a store that satisfies ρ, $\mathit{Pre}(P)$ is a valid precondition for every procedure $P \in C$, and $\mathit{Pre}(\texttt{while}\ \mathit{le}\ \texttt{do}\ s)$ is a valid loop invariant for every loop $\texttt{while}\ \mathit{le}\ \texttt{do}\ s \in C$.

At this point we want to stress that, in practice, $\mathit{Safe}(\rho, s', \mathit{Pre})$ is established without any extra burden for the programmer nor the verifier. That is, for *Pre* it is sufficient to use the preconditions and loop invariants that are necessary to prove the safety of the transformed program.

To restrict the nondeterminism in the program, we allow the programmer to supply restricted versions of the atomic actions called in C. Since all those atomic actions are left movers, we also have to preserve the nonblocking property. Formally, a *restriction function r* is a partial mapping from atomic action names to atomic action names, such that for all $A \in \mathcal{P}$ with $\mathcal{P}.A = (\rho, \alpha)$ and $r(A) = A'$, it holds that $\mathcal{P}.A' = (\rho, \alpha')$ with $\alpha' \subseteq \alpha$ and A' is nonblocking. Checking this conditions is a purely logical reasoning task performed by a theorem prover. Given a restriction function r, let $\mathcal{P}^r$ be the program equal to $\mathcal{P}$, except that $\mathcal{P}^r.A = r(A)$ for A in the domain of r.

Now we are ready to state the termination checks that can be solved by any standard termination checker for sequential programs. In particular, for a termination checker based on user-provided ranking functions, the required annotations can be provided directly in the source code of the original program. Intuitively we require that every loop and procedure in C terminates with restricted nondeterminism from every store in *Pre*.

Definition 6.2. $\mathit{Terminates}(C, \mathit{Pre}, r)$ holds, if all the following $\mathcal{P}^r$ executions terminate:
- executions of P with $P \in C$ that start in $\mathit{Pre}(P)$, and
- executions of $\texttt{while}\ \mathit{le}\ \texttt{do}\ s$ with $\texttt{while}\ \mathit{le}\ \texttt{do}\ s \in C$ that start in $\mathit{Pre}(\texttt{while}\ \mathit{le}\ \texttt{do}\ s)$.

Theorem 6.3. *Let M be a solution to the equations in Figure 6 w.r.t. a set C, such that $M(s) \neq \varnothing$. Let Pre be a safety annotation and let r be a restriction function such that $\mathit{Safe}(\rho, s', \mathit{Pre})$ and $\mathit{Terminates}(C, \mathit{Pre}, r)$. Then $\mathit{Cooperative}(\rho, s')$.*

Example 6.4. Recall the simple agreement protocol from Figure 3. Figure Figure 8 illustrates how we establish the cooperation condition; (a) shows an excerpt of the original nondeterministic if condition; (b) shows the corresponding implementation in our programming language using a call to an atomic action that nondeterministically initializes a local boolean variable b; (c) shows a user-provided restriction of the atomic action that always sets b to `true`. Now the termination check for the restricted program is trivial, since all recursive calls have been eliminated.

(a)

```

if (*) {
  // accept
} else {
  // reject
}
```

(b)

```
call [ havoc b ]
if (b) {
  // accept
} else {
  // reject
}
```

(c)

```
call [ b := true ]
if (b) {
  // accept
} else {
  // reject
}
```

Fig. 8. Showing cooperation for the agreement protocol from Section 2.3.

7 A REFINEMENT-BASED PROOF SYSTEM FOR ASYNCHRONOUS PROGRAMS

The synchronization rule introduced in the previous section establishes a refinement relation between two programs. Thus it can be generally combined with any other technique that is compatible with this interface. In this section we present proof rules that complement synchronization to form the basis of our flexible and practical proof methodology for asynchronous programs.

Abstraction. First and foremost, after the formation of an atomic block using our synchronization rule, we usually desire to transform the atomic block further into the invocation of an atomic action. To do so, we need to show that the atomic block behaves as that atomic action. Formally, an atomic block $\texttt{atomic}\ \rho_1\ s$ *refines* an atomic action (ρ, α), denoted $\texttt{atomic}\ \rho_1\ s \preccurlyeq (\rho, \alpha)$, if

(1) $\overline{\rho_1} \cup \{g \cdot \ell \mid (g, (\ell, s)) \to^* \lightning\} \subseteq \overline{\rho_2}$ and
(2) $\rho_2 \circ \{(g \cdot \ell, g' \cdot \ell)) \mid (g, (\ell, s)) \to^* (g', (\ell', \texttt{skip}))\} \subseteq \alpha_2$.

Note that this is a purely sequential condition. We call the resulting program transformation *atomization* since the stepwise behavior of s is summarized by a single atomic action.

$$\frac{\texttt{atomic}\ \rho\ s \preccurlyeq \mathcal{P}.A}{RC[\texttt{atomic}\ \rho\ s] \rightsquigarrow RC[\texttt{call}\ A]}\ \textsc{Atomize}$$

Thus, synchronization and atomization together summarize in a single atomic action an arbitrary asynchronous computation executed by a potentially unbounded number of concurrently-executing threads.

Furthermore, we can replace atomic actions with more abstract ones. An atomic action (ρ_1, α_1) refines an atomic action (ρ_2, α_2), denoted $(\rho_1, \alpha_1) \preccurlyeq (\rho_2, \alpha_2)$, if

(1) $\rho_2 \subseteq \rho_1$, and
(2) $\rho_2 \circ \alpha_1 \subseteq \alpha_2$.

$$\frac{\mathcal{P}.A \preccurlyeq \mathcal{P}.A'}{RC[\texttt{call}\ A] \rightsquigarrow RC[\texttt{call}\ A']}\ \textsc{Abstract}$$

Abstraction is the main antagonists to synchronization and usually necessary between two synchronization steps in order to make the atomic actions after an application of synchronization commute as desired for the next application of synchronization.

Finally, if the body of a procedure was reduced to a single atomic action invocation, we can replace calls to the procedure with calls to the atomic action.

$$\frac{\mathcal{P}.P = \texttt{call}\ A}{RC[\texttt{call}\ P] \rightsquigarrow RC[\texttt{call}\ A]}\ \textsc{Collapse}$$

Atomic action & procedure introduction. We can always introduce new atomic actions and procedures into the program. Note that we require programs to be well-formed. Thus, mutually recursive procedures have to be introduced together.

$$\frac{A \notin \mathcal{P}}{\mathcal{P} \rightsquigarrow \mathcal{P} \cup [A \mapsto (\rho, \alpha)]}\ \textsc{ActionIntro} \qquad \frac{\forall i : P_i \notin \mathcal{P}}{\mathcal{P} \rightsquigarrow \mathcal{P} \cup \bigcup_i [P_i \mapsto s_i]}\ \textsc{ProcIntro}$$

ProcIntro is the formal justification for introducing synchronized procedures into $\mathcal{P}$ in order to apply Synchronize. ActionIntro is the formal justification to introduce restricted atomic actions for our cooperation check, as well as for the abstraction rules defined above.

Atomic action & procedure elimination. Contrary to introduction, we can eliminate atomic actions and procedures from the program, if they are not called from *Main*.

$$\frac{A \notin \mathit{MaybeInvoked}(\mathcal{P})}{\mathcal{P} \cup [A \mapsto (\rho, \alpha)] \rightsquigarrow \mathcal{P}}\ \textsc{ActionElim} \qquad \frac{\forall i : P_i \notin \mathit{MaybeInvoked}(\mathcal{P})}{\mathcal{P} \cup \bigcup_i [P_i \mapsto s_i] \rightsquigarrow \mathcal{P}}\ \textsc{ProcElim}$$

Since the mover types of atomic actions depend on all other atomic actions in the program, the elimination of unused atomic actions is particularly important to establish the desired mover types.

Variable introduction & elimination. Since it is not central to our innovation on synchronizing asynchronous computations, we do not formalize the introduction and elimination of local and global program variables. However, we note that in practice it is immensely helpful to allow different sets of variables to exist on different levels of refinement, which is supported in our implementation.

Theorem 7.1. *If $\mathcal{P}_1 \rightsquigarrow \mathcal{P}_2$ using any of the rules defined in this section, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.*

Theorem 7.2. *If $\mathcal{P}_1 \rightsquigarrow^* \mathcal{P}_2$, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$.*

Proof. Follows from $\rightsquigarrow \subseteq \preccurlyeq$ (Theorem 4.5 and Theorem 7.1) and the transitivity of $\preccurlyeq$ (Lemma 3.2). □

8 EVALUATION AND EXPERIENCE

We implemented and evaluated our verification method in the context of CIVL [29], a verification system for concurrent programs based on automated and modular refinement reasoning. CIVL itself is implemented as a conservative extension of the Boogie language and verifier [2]. A CIVL program comprises a set of procedures that are specified and verified across multiple layers of refinement. At each layer, procedures can be declared to refine an atomic action and henceforth appear atomic to higher layers. Refinement is established via a blend of logic-based and automata-based reasoning. Proof hints are usually required in the form of location invariants to aid in non-interference [50] or rely-guarantee [35] reasoning. However, those annotations are not required to be strong enough to prove program correctness but only strong enough to provide the context for refinement checking. Finally, a linear type system enables logical encoding of thread identifiers, permissions, etc., significantly reducing the annotation burden.

Integrating our synchronization proof rule into CIVL required modifications throughout the processing pipeline, including the treatment of asynchronous calls, the type checker for layer annotations, the type checker for mover types, and the verification condition generation algorithm. While the implementation of a termination checker for sequential programs is an orthogonal challenge, we are currently working on a termination feature for Boogie. For now, we hand-translated the termination problems corresponding to the cooperation checks for our examples into Dafny [41], which proved termination without any user-provided annotations.

We evaluated our technique on several examples, which we discuss in the remainder of this section. First, we considered a progression of programs expanding our motivating examples from Section 2. This includes the implementation of a barrier mechanism that allows a client to block

```
const N : int;
axiom N > 0;

// ######################################
// Global shared variable

var {:layer 0} x : int;

// ######################################
// Layer 0: Low level atomic actions

procedure {:layer 0} inc ();
atomic {:both} |{ x := x + 1; }|;

procedure {:layer 0} dec ();
atomic {:both} |{ x := x - 1; }|;

// ######################################
// Layer 1: Main and inc/dec by N

procedure {:layer 1} main ()
atomic {:both} |{ skip; }|;
{
  async call inc_by_N();
  async call dec_by_N();
}
```

```
procedure {:layer 1} {:left} inc_by_N ()
modifies x;
ensures {:layer 1} x == old(x) + N;
{
  var i := 0;
  while (i != N)
  invariant {:layer 1} x == old(x) + i;
  {
    i := i + 1;
    async call inc();
  }
}


procedure {:layer 1} {:left} dec_by_N ()
modifies x;
ensures {:layer 1} x == old(x) - N;
{
  var i := 0;
  while (i != N)
  invariant {:layer 1} x == old(x) - i;
  {
    i := i + 1;
    async call dec();
  }
}
```

Fig. 9. Asynchronous increments and decrements in CIVL.

and wait for the completion of an asynchronous computation. Second, our central touchstone was the specification and verification of the two-phase commit protocol. Our implementation is very fast; the full 2PC benchmark is verified in 3 seconds on a standard Laptop (1.60GHz × 4, 8GB) running Ubuntu. Third, we implemented the Paxos consensus algorithm and a protocol related to a distributed hash table. We are working on the completion of those proofs and report on the atomic action specification.

8.1 Layered Proofs

In Figure 9 we show the CIVL implementation of our motivating example in Figure 1, with the slight modifications that we use a symbolic constant N instead of the fixed constant 100, and both loops creating the increment and decrement threads are factored into separate procedures that are in turn asynchronously called by main.

First we need to understand the semantics of layer annotations to avoid the pitfall of believing that the layer associated with a procedure denotes the one and only layer the procedure exists on. This is not the case! Conceptually a procedure exists across all layers, but with a layer annotation we express that at this particular layer we want to prove something about the procedure, e.g., an atomic action specification that simplifies the program for reasoning at higher layers.

Layer 0. Layer 0 is the lowest layer in our proof and represents the original program we want to reason about. At this layer the procedures inc_by_N, dec_by_N, and main consist just of their implementations as shown in Figure 9. On the other hand, the procedures inc and dec are declared with associated atomic action specifications. Since this procedures do not have bodies, we assume the atomic actions to be given and there is nothing to be proved at layer 0. However, it would be possible to provide implementations for the procedures on an even lower layer, e.g., to model a platform that does not support atomic integer operations. Then the layer annotation would introduce a proof obligation to guarantee that it is sound to reason about `inc` and `dec` using the atomic action specifications at layers above 0. Intuitively, at the boundary of layer 0 the implementations of `inc` and `dec` disappear and only the atomic action specifications remain.

Layer 1. At layer 1 our main goal is to prove the atomic action specification of `main`, that states that higher layers can forget about all the asynchronous computation, and just pretend as if `main` executes atomically and leaves the global state unchanged. Formally, this is achieved via a combination of synchronization and atomization. In the code of Figure 9, `inc_by_N` and `dec_by_N` are annotated with a mover type to indicate that they should participate in synchronization and which type they should be assigned for type checking. We already discussed the type checking in Example 5.2. The cooperation condition is easy to establish because the while loops are statically bounded. Now synchronization allows us to consider, at layer 1, (a) all asynchronous calls as synchronous calls, and (b) the body of `main` to execute atomically. Thus we can resort to the sequential technique of using preconditions and postconditions to reason about procedure calls, which allows atomization (i.e., the sequential reasoning engine of CIVL) to conclude that `main` leaves the global state unchanged, establishing its atomic action specification. Note that the stated postconditions and the loop invariants to prove them are not valid at layer 0, where we still have asynchronous calls.

8.2 Verifying the Two-Phase Commit Protocol

The *two-phase commit protocol (2PC)* is a distributed algorithm that allows a set of processes to collectively agree on the outcome of a transaction, i.e., whether to commit or abort a transaction. Thus it is used, e.g., in database systems to implement distributed atomic transactions. The protocol works as follows. There is one dedicated *coordinator* process, and a fixed number of *participant* processes. For every incoming request, the coordinator initializes a transaction and forwards the request to all participants. Each participant decides if it is able to commit the transaction, and replies to the coordinator either with a “yes” vote to commit, or a “no” vote to abort. However, in the case of a “yes” vote, a participant does not yet persistently commit the transaction. The coordinator processes incoming votes as follows: (1) If all participants voted “yes”, the coordinator declares the transaction as committed and sends a “commit” message to all participants. (2) If, on the other hand, a single participant voted “no”, the coordinator immediately declares the transaction as aborted and sends an “abort” message to all participants. Due to asynchrony and message reordering, the protocol implementation must be robust against unexpected situations, e.g., a participant receiving an abort message for a transaction it was not yet asked to vote for.

In CIVL we implement every message handler as a procedure and model the sending of a message as an asynchronous call to the corresponding message handler (in another process). This novel way of expressing message passing eliminates shared-state message buffers and captures the standard limitations of message delivery in computation networks, i.e., potentially lost and reordered messages. Our approach could also account for duplicated messages via nondeterministic loops around calls to message handlers. However, in this example we did not consider message duplication. Figure 10 shows the message handlers in 2PC together with their call hierarchy. We use the convention of prefixing their names with `C` for coordinator or `P` for participant, e.g., `P_VoteReq` is the handler of a participant process for vote request messages sent by the coordinator. Figure 11 shows the implementation of message handlers together with their atomic action specifications.

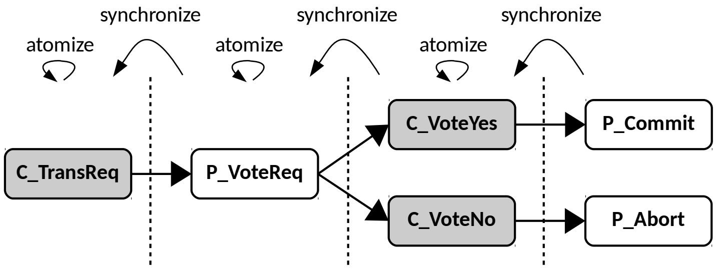

Fig. 10. 2PC call hierarchy (from left to right) and proof outline (right to left).

The ghost variable state holds the belief about the state of every transaction for every process, which can be *undecided*, *committed*, or *aborted*. The real state is held in the votes variable, which is used by the coordinator to count the number of "yes" votes it received for every transaction. The correctness of 2PC is expressed by the predicate xConsistent in the top-level specification of C_TransReq, that states that for the allocated transaction there are no two processes such that one believes the transaction to be committed and the other one aborted. To prove this property we use the combined power of our synchronization rule together with the existing atomization in CIVL, illustrated in Figure 10.

The first step is to synchronize the call to P_Commit in C_VoteYes, and the call to P_Abort in C_VoteNo. For that, both actions have to be left movers with respect to themselves, each other, and the actions C_VoteYes_Update and C_VoteNo_Update which perform the local state update in the coordinator, i.e., all actions called within the context of C_VoteYes and C_VoteNo. For example, P_Commit and P_Abort commute because their gates have opposing assertions about the state of the coordinator; P_Commit asserts that the coordinator is already committed, while P_Abort asserts that the coordinator is already aborted. It is important to highlight that those assertions are not only used for left mover checks. They also serve as safety specifications about intermediate states during program execution that are verified by our methodology.

In our next synchronization step we want to synchronize both the call of C_VoteYes and C_VoteNo in P_VoteReq. However, for that we first have to convert them into atomic actions by means of atomization. Both atomic action specifications state that the current state is *extended*, such that it remains consistent and no process goes from committed or aborted back to undecided. The atomization as well as the subsequent left mover checks for C_VoteYes and C_VoteNo make use of linear permissions as follows. In C_TransReq, for every transaction with identifier xid we also allocate a ghost set of permissions consisting of all pairs (xid,pid), where pid is a process identifier of a participant. The calls to P_VoteReq and in turn C_VoteYes and C_VoteNo get passed the appropriate permission p, corresponding to their xid and pid parameter, which is expressed by the predicate p.perm(xid,pid). In C_VoteYes, all incoming permissions are collected in a ghost set B. The linear type checker ensures that the collection of permissions stored in linear variables live at any time during execution never contain the same permission twice. In particular, we are guaranteed that no two instances of C_VoteYes and C_VoteNo ever execute with the same values for xid and pid. And further, capturing the correctness mechanism of 2PC, if all permissions for a particular xid arrived in B, there cannot be any pending instance of C_VoteNo for that xid.

The remaining atomization of P_VoteReq, the synchronization of P_VoteReq in C_TransReq, and the atomization of C_TransReq are simple propagation.

```
var state : Xid × Pid → TransactionState; // ghost
var  votes : Xid → int;
var linear B : Set of (Xid × Pid);  // ghost
```

```
C_TransReq () {
  call xid, linear perms := AllocateXid();
  for (i in 1..numParticipants) {
    async call P_VoteReq(xid, i, perms[i]);
  }
  return xid;
}
```

```
  havoc state[xid];
  assume xConsistent(state[xid]);
```

```
P_VoteReq (xid, pid, linear_in p) {
  if (*) {
    async call C_VoteYes(xid, pid, p);
  } else {
    call SetParticipantAborted(xid, pid, p);
    async call C_VoteNo(xid, pid, p);
  }
}
```

```
assert p.perm(xid, pid);
assert xConsistent(state[xid]);
havoc state[xid];
assume xExtends(old(state[xid]), state[xid]);
```

```
C_VoteYes (xid, pid, linear_in p) {
  call commit := C_VoteYes_Update(xid, pid, p);
  if (commit) {
    for (i in 1..numParticipants) {
      async call P_Commit(xid, i);
    }
  }
}
```

```
assert p.perm(xid, pid);
assert xConsistent(state[xid]);
add p to B;
if (xAllPermsInB(xid)) {
  havoc state[xid];
  assume xExtends(old(state[xid]), state[xid]);
}
```

```
C_VoteNo (xid, pid, linear_in p) {
  call abort := C_VoteNo_Update(xid, pid);
  if (abort) {
    for (i in 1..numParticipants) {
      async call P_Abort(xid, i);
    }
  }
}
```

```
assert p.perm(xid, pid);
assert xUndecidedOrAborted(state[xid]);
havoc state[xid];
assume xUndecidedOrAborted(state[xid]);
assume xExtends(old(state[xid]), state[xid]);
```

```
SetParticipantAborted (xid, pid, linear p)
assert p.perm(xid, pid);
assert xUndecidedOrAborted(state[xid]);
state[xid][pid] := ABORTED;
```

```
C_VoteYes_Update (xid, pid)
assert VotesEqState(votes[xid], state[xid][C_Pid]);
if (votes[xid] != -1) {
  votes[xid] := votes[xid] + 1;
  if (votes[xid] == numParticipants) {
    state[xid][C_Pid] := COMMITTED;
    return true;
  }
}
return false;
```

```
C_VoteNo_Update (xid, pid)
assert !Committed(state[xid][C_Pid]);
if (votes[xid] != -1) {
  state[xid][C_Pid] := ABORTED;
  votes[xid] := -1;
  return true;
}
return false;
```

```
P_Commit (xid, pid)
assert participantPid(pid);
assert Committed(state[xid][C_Pid]);
assert xUndecidedOrCommitted(state[xid]);
state[xid][pid] := COMMITTED;
```

```
P_Abort (xid, pid);
assert participantPid(pid);
assert Aborted(state[xid][C_Pid]);
assert xUndecidedOrAborted(state[xid]);
state[xid][pid] := ABORTED;
```

Fig. 11. Excerpts from our 2PC implementation and proof (solid boxes show procedure implementations, dashed boxes show atomic action specifications). Thick lines represent the layers of bottom-up synchronization and atomization (see Figure 10). For some procedures we show both the implementation and the atomic action specification (dashed box attached to a solid box). For some procedures we only show the atomic action specification (detached dashed box).

8.3 Other Distributed Protocols

In addition to 2PC, we implemented and specified the Paxos consensus protocol and a relocation protocol for a distributed hash table. We are currently working on the completion of the full proofs. In this section we focus on the top-level atomic action specifications that provide both guidance in structuring the proof top down and a simple interface to reason about clients building on top of the protocols.

Paxos. Paxos [39] is a consensus protocol that allows a set of processes to agree on a common value. Despite the seemingly simple correctness argument of the protocol, we made a similar experience as previously described elsewhere, e.g., [10]: there are many choices to be made to go from an abstract protocol description found in the literature to a concrete implementation. Those choices are far reaching and often lead to bugs in the running system. Thus it is important to connect the implementation-level mechanism, such as counters and flags, with the abstract protocol mechanism.

Surprisingly, the top-level atomic action specification of our implementation is essentially the same as the one for 2PC. We have a specification variable that represents the final chosen value of every process (which can be uninitialized). Then the atomic action specification of Paxos states that the chosen values have to be consistent (i.e., at most one chosen value), and that a state update (1) keeps the values consistent, and (2) no process that already chose a value can flip its decision.

Paxos is often used to implement a replicated state machine. Using our specification simplifies the task of reasoning about the consistency of a replicated state machine that uses the Paxos implementation to decide on the state machine commands to be executed.

Distributed hash table. A distributed hash table (DHT), for example Chord [58], distributes the storage of individual hash entries among many nodes. To balance the load on the storage nodes, a central operation in a DHT system is to relocate a block of hash table entries from one node to another. The mechanism for such a relocation can be quite complex, in particular if several relocations happen concurrently. However, our top-level atomic action specification for the relocation operation is, that before and after the operation all hash table entries have to be stored on at least one node.

Free from the low-level asynchronous communication, our specification can be readily used to establish the hash table abstraction of the DHT system towards clients.

8.4 Discussion

Based on our experience we now discuss four core contributions of our approach to the state of the art in deductive program verification of concurrent programs.

Proof directionality. Our proof methodology directly suggests a strategy to approach a proof by alternating applications of synchronization and atomization, guided by the syntactic structure of a program. For a distributed protocol like 2PC, a proof is constructed along the communication pattern of the protocol. The presentation in this paper consistently followed a bottom-up approach, going stepwise from a concrete program towards an abstract specification. However, it is equally possible, and often beneficial, to take a top-down approach by stepwise refining an abstract specification into an executable implementation.

Proof structure. Compared to the complexity of writing a monolithic invariant over the entire state of a flat transition system in one shot, our layered approach promotes proof productivity by decomposing the annotation burden into smaller manageable pieces. Establishing the justification for a particular synchronization or atomization step sets a narrow focus of attention. Furthermore, different parts of the invariant can be stated at appropriate levels of abstraction. For example, our

2PC proof separates an invariant about the protocol mechanism (voting and commit phase) from an invariant about its low-level implementation (counting “yes” votes).

Specification. We introduce a simple way of specifying the behavior of asynchronous operations as relations between initial and quiescent states, similar to pre- and postconditions of sequential operations. While proving such a specification *for* an asynchronous operation is still challenging, reasoning *with* the specification is simple. For example, it is easy to verify a client for 2PC that issues multiple transaction requests using the atomic action specification of `C_TransReq`. Furthermore, our proof methodology accommodates safety specifications about intermediate results as follows. First, a programmer can write local assertions that are verified during stepwise refinement, as we demonstrated in our 2PC proof. Second, a programmer can introduce auxiliary history variables that capture specific effects as they occur during execution. This history variables are available in atomic action specifications. Incidentally, the term history is used in [5] to denote a process algebra term that represents all sequences of atomic actions in a concurrent program, which corresponds to a refinement step in our setting.

Message-passing. Our way of modeling message sends as asynchronous procedure calls eliminates message buffers as explicit shared state and thus focuses the correctness argument on the underlying mechanism, instead of its implications on the shape and contents of message buffers. Another interesting consequence is that message handlers of a particular process do not necessarily execute sequential. For example, in our 2PC implementation every process can handle multiple messages concurrently without waiting for previous handlers to terminate.

9 RELATED WORK

This paper is about computer-assisted deductive verification of asynchronous concurrent programs. We build upon the classical works that introduced the central notion of *stability under interference (i.e., noninterference)* to enable invariant-based reasoning about concurrent programs that share state [35, 50]. While pure asynchronous message-passing programs [33] do not explicitly share state, message buffers in their operational semantics are inherently shared and thus become subject to interference [56]. Our goal is to bridge the gap between these theoretical foundations and their applicability in the construction and verification of practical systems. In particular, we provide support for the discovery and construction of inductive invariants via proof structuring mechanisms and automation.

There are several recent works closely related in spirit to ours. Ivy [52] 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. The core insight is the use of a restricted modeling and specification language that makes their verification conditions decidable. We use a rich specification logic and rely on partitioned verification conditions that can be discharged by an SMT solver [12]. The IronFleet methodology [28] embeds TLA-style state-machine modeling [40] into the Dafny verifier [41] 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. PSync [16] 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 [62] lets a programmer provide a specification, implementation, and proof of a distributed system under a simple albeit unrealistic network model. The application is automatically transformed into one that handles faults via verified system transformers. The work of [23] introduces a rely-guarantee rule for a weaker form of asynchrony, where a single task queue atomically executes one task at a time.

Concurrent separation logic (CSL) [48] was devised for modular reasoning about multi-threaded shared-memory programs, with recent generalizations [15, 36, 47, 59], mechanizations [57], and tools [4, 14, 45] 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 the protocol-level concurrency. For example, [49] uses CSL to link implementation steps of message-passing programs to atomic actions, and relies on a model checker to explore the interleavings of those atomic actions. Conversely, our reduction-based approach allows the flexible construction of a refinement proof that splits the problem of invariant discovery into smaller sub-tasks, but currently does not support the elegance of CSL to reason about dynamic memory. Similar concepts that are used in both approaches are, e.g., the use of permissions [6]. The actor services of [45] focus on compositional verification of response properties of message-passing programs via assertions that allow to specify that particular trigger messages necessarily leads to described response messages. Although for technically different reasons, they require the termination of individual message handlers akin to our cooperation condition.

There is a wide range of research concerned with techniques for fully-automatic analysis of concurrent programs. This includes works on state space exploration based on model checking [34], partial-order reduction [24, 53], abstraction refinement [26, 31, 32, 54, 61], counting/cardinality [19, 60], Petri nets/well-structured transition system [18, 22, 38], cutoff results [37], counter abstraction [3], abstract interpretation [44], etc. These techniques are complementary to our approach and both can benefit from each other. On the one hand, we can use automated procedures in our approach wherever possible to limit the amount of user interaction. On the other hand, our proof structuring results in smaller and simpler input for automatic procedures, and thus makes them amenable to large programs otherwise out of reach. We consider the robust integration of heterogeneous automation techniques into an interactive verification system an important line of future work.

Systematic testing, as in P [13], relies on a user-provided test harness to perform a clever exploration of concurrent interleavings to discover bugs. The correctness notion of robustness against concurrency [7] allows to reduce concurrent interleavings to serial executions in the testing setting.

While our work focuses on asynchronous concurrency, we note that there is a broad body of work on the verification of synchronous concurrency, often in connection with hardware, embedded systems, and robotics (e.g., rendezvous in process algebras [55], events in synchronous languages [27], time in synchronous models [1, 43]).

10 CONCLUSION

The main contribution of this paper is a new way to structure correctness proofs of asynchronous concurrent programs. The classical, unstructured proof method relies on the discovery of an inductive invariant that accounts for all possible interleavings of concurrent processes. Our proof method decomposes the task, guided by the program structure and syntax, into formulating and automatically discharging smaller, independent proof obligations. These proof obligations will require to 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 most importantly, that an assertion is an inductive invariant for a simpler program, where all asynchronous procedure calls are replaced by synchronous (immediate) atomic actions. Thus, 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. Furthermore, counterexamples to proof obligations are local and can be readily used to identify and fix bugs. This makes real concurrent software amenable to semi-automatic verification.

REFERENCES

[1] Rajeev Alur and Thomas A. Henzinger. 1999. Reactive Modules. *Formal Methods in System Design* 15, 1 (1999), 7–48.

[2] 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*.

[3] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. 2010. Context-aware counter abstraction. *Formal Methods in System Design* 36, 3 (2010), 223–245.

[4] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *FM*.

[5] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. 2015. History-Based Verification of Functional Behaviour of Concurrent Programs. In *SEFM*.

[6] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*.

[7] Ahmed Bouajjani, Michael Emmi, Constantin Enea, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. 2017. Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency. In *ESOP*.

[8] John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS*.

[9] Aaron R. Bradley and Zohar Manna. 2007. *The calculus of computation - decision procedures with applications to verification*. Springer.

[10] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *PODC*.

[11] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting Families of Schedules for Asynchronous Programs. In *CAV*.

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

[13] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *PLDI*.

[14] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP*.

[15] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP*.

[16] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*.

[17] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A calculus of atomic actions. In *POPL*.

[18] Michael Emmi, Pierre Ganty, Rupak Majumdar, and Fernando Rosa-Velardo. 2015. Analysis of Asynchronous Programs with Event-Based Synchronization. In *ESOP*.

[19] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs that count. In *POPL*.

[20] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.* 103, 2 (1992), 235–271.

[21] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *PLDI*.

[22] Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 6:1–6:48.

[23] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. 2015. Rely/Guarantee Reasoning for Asynchronous Programs. In *CONCUR*.

[24] Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Vol. 1032. Springer.

[25] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *POPL*.

[26] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*.

[27] Nicolas Halbwachs. 1992. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers.

[28] 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*.

[29] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. Automated and Modular Refinement Reasoning for Concurrent Programs. In *CAV*.

[30] Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015. *Automated and Modular Refinement Reasoning for Concurrent Programs.* Technical Report MSR-TR-2015-8. Microsoft Research. https://www.microsoft.com/en-us/research/publication/automated-and-modular-refinement-reasoning-for-concurrent-programs/

[31] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2004. Race checking by context inference. In *PLDI.*

[32] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. 2003. Thread-Modular Abstraction Refinement. In *CAV.*

[33] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI.*

[34] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295.

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

[36] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL.*

[37] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. In *CAV.*

[38] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2014. A Widening Approach to Multithreaded Program Verification. *ACM Trans. Program. Lang. Syst.* 36, 4 (2014), 14:1–14:29.

[39] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

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

[41] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR.*

[42] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

[43] Kenneth L. McMillan. 2000. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.* 37, 1-3 (2000), 279–309.

[44] Antoine Miné and David Delmas. 2015. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *EMSOFT.*

[45] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI.*

[46] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI.*

[47] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *ESOP.*

[48] Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307.

[49] Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES.*

[50] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6 (1976), 319–340.

[51] Burcu Kulahcioglu Ozkan, Michael Emmi, and Serdar Tasiran. 2015. Systematic Asynchrony Bug Exploration for Android Apps. In *CAV.*

[52] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *PLDI.*

[53] Doron A. Peled. 1993. All from One, One for All: on Model Checking Using Representatives. In *CAV.*

[54] Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. 2014. Reduction for compositional verification of multi-threaded programs. In *FMCAD.*

[55] A. W. Roscoe. 1997. *The Theory and Practice of Concurrency.* Prentice Hall.

[56] Richard D. Schlichting and Fred B. Schneider. 1984. Using Message Passing for Distributed Programming: Proof Rules, Disciplines. *ACM Trans. Program. Lang. Syst.* 6, 3 (1984), 402–431.

[57] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *PLDI.*

[58] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM.*

[59] Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ICFP.*

[60] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI.*

[61] Björn Wachter, Daniel Kroening, and Joël Ouaknine. 2013. Verifying multi-threaded software with impact. In *FMCAD.*

[62] 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*.

A COMMUTATIVITY

Let m be a mapping from atomic action names to mover types $\{R, L, B, A\}$. Then 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 need to hold.

- **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 be achieved by executing A_2 followed by A_1.

$$\forall g, \bar{g}, g', \ell_1, \ell_1', \ell_2, \ell_2' \; \exists \hat{g} : \begin{pmatrix} & g{\cdot}\ell_1 \in \rho_1 \\ \wedge & g{\cdot}\ell_2 \in \rho_2 \\ \wedge & g{\cdot}\ell_1 \xrightarrow{\alpha_1} \bar{g}{\cdot}\ell_1' \\ \wedge & \bar{g}{\cdot}\ell_2 \xrightarrow{\alpha_2} g'{\cdot}\ell_2' \end{pmatrix} \implies \begin{pmatrix} & g{\cdot}\ell_2 \xrightarrow{\alpha_2} \hat{g}{\cdot}\ell_2' \\ \wedge & \hat{g}{\cdot}\ell_1 \xrightarrow{\alpha_1} g'{\cdot}\ell_1' \end{pmatrix}$$

- **Forward preservation:** If $M(A_1) \in \{R, B\}$ or $M(A_2) \in \{L, B\}$, then the failure of A_2 immediately after the execution of A_1 is implies that A_2 must also fail before the execution of A_1. This condition is equivalent to forward preservation of the gate of A_2 by the execution of A_1.

$$\forall g, g', \ell_1, \ell_1', \ell_2 : \begin{pmatrix} & g{\cdot}\ell_1 \in \rho_1 \\ \wedge & g{\cdot}\ell_2 \in \rho_2 \\ \wedge & g{\cdot}\ell_1 \xrightarrow{\alpha_1} g'{\cdot}\ell_1' \end{pmatrix} \implies g'{\cdot}\ell_2 \in \rho_2$$

- **Backward preservation:** If $M(A_2) \in \{L, B\}$, then the failure of A_1 immediately before the execution of A_2 implies that A_1 must also fail immediately after the execution of A_2. This condition is equivalent to backward preservation of the gate of A_1 by the execution of A_2.

$$\forall g, g', \ell_1, \ell_2, \ell_2' : \begin{pmatrix} & g{\cdot}\ell_2 \in \rho_2 \\ \wedge & g{\cdot}\ell_2 \xrightarrow{\alpha_2} g'{\cdot}\ell_2' \\ \wedge & g'{\cdot}\ell_1 \in \rho_1 \end{pmatrix} \implies g{\cdot}\ell_1 \in \rho_1$$

- **Nonblocking:** If $M(A_2) \in \{L, B\}$, then moving the execution of A_2 to the left is not allowed to prevent a (potentially failing) execution from making progress.

$$\forall \sigma \in \rho_2 \; \exists \sigma' : \sigma \xrightarrow{\alpha_2} \sigma'$$