Reduction for Structured Concurrent Programs

Namratha Gangamreddypalli[1], Constantin Enea[1], and Shaz Qadeer[2]

[1] LIX, Ecole Polytechnique, CNRS and Institut Polytechnique de Paris, France
{namratha, cenea}@lix.polytechnique.fr
[2] Microsoft

Abstract. Commutativity reasoning based on Lipton's movers is a powerful technique for verification of concurrent programs. The idea is to define a program transformation that preserves a subset of the initial set of interleavings, which is sound modulo reorderings of commutative actions. Scaling commutativity reasoning to routinely-used features in software systems, such as procedures and parallel composition, remains a significant challenge.

In this work, we introduce a novel reduction technique for structured concurrent programs that unifies two key advances. First, we present a reduction strategy that soundly replaces parallel composition with sequential composition. Second, we generalize Lipton's reduction to support atomic sections containing (potentially recursive) procedure calls. Crucially, these two foundational strategies can be composed arbitrarily, greatly expanding the scope and flexibility of reduction-based reasoning. We implemented this technique in Civl and demonstrated its effectiveness on a number of challenging case studies, including a snapshot object, a fault-tolerant and linearizable register, the FLASH cache coherence protocol, and a non-trivial variant of Two-Phase Commit.

1 Introduction

Commutativity reasoning is a powerful technique for verification of concurrent programs. This method derives from the observation that certain pairs of concurrently-executing statements can be reordered without affecting program behavior, i.e., such statements commute. Interleavings (i.e., concurrent execution sequences) that differ only in the ordering of commuting statements are considered equivalent. As a result, it is sufficient to verify the correctness of a single representative interleaving from each equivalence class. These techniques are also called *reduction* techniques because they reduce reasoning to a smaller set of representative interleavings.

In static verification of concurrent programs, a standard method to exploit commutativity reasoning is to capture the reduced program via a syntactic transformation of the original program. The proof of correctness is then done on the transformed program and the reduction argument is used to carry over the results of the verification to the original program. Lipton [35] introduced atomic sections as a simple method to capture such a transformation. Since then, atomic sections have been used extensively to specify non-interference in and simplify reasoning about concurrent programs [19,17,11,23,15].

In this paper, we focus on the application of commutativity reasoning towards deductive verification of concurrent programs. Applying commutativity reasoning to real-world programs is challenging. Software systems routinely use procedures for code structuring and scaling software engineering. Concurrent systems, in addition, are performance oriented and often launch multiple tasks in parallel, collecting results as the tasks complete. These features, procedures and dynamic concurrency, are not adequately addressed by existing approaches. Lipton [35] only addresses the problem of concurrent programs with bounded threads and atomic sections of bounded size. QED [11] and Civl [23] handle atomic sections containing loops, but do not handle procedure calls (unless they are inlined beforehand) or parallel composition. Kragl et al. [30] present a program transformation that synchronizes asynchronous procedure calls by demonstrating that the called procedure can be summarized as a single atomic action that commutes to the left of any other program action (a so-called left mover). While asynchronous calls can be viewed as a restricted form of parallel composition, this model is not general enough for our purposes (see Section 9).

We introduce a novel reduction technique for structured concurrent programs that unifies two key advances. First, we present a reduction strategy that soundly replaces parallel composition with sequential composition, addressing a dimension orthogonal to atomic section introduction explored in prior work. Second, we generalize Lipton's reduction to support atomic sections containing (potentially recursive) procedure calls. Crucially, these two foundational strategies can be composed arbitrarily, greatly expanding the scope and flexibility of reduction-based reasoning.

Our reduction technique is based on a concept of *movers* or commuting statements [35,11,23]. The soundness of atomic section introduction relies on demonstrating that it consists of a sequence of *right* movers, followed by an arbitrary statement, and then a sequence of *left* movers. Right movers are statements that commute to the right of any other statement in the program, while left movers commute to the left; this terminology imagines time flowing from left to right. Importantly, mover classification is relative to the set of actions under consideration: whether a statement is a right or left mover depends on how it interacts with the other actions in the program. This straightforward intuition for movers is deceptive; precise definitions are non-trivial since they must account for statements that may fail or are non-deterministic.

To handle structured code, we extend the notion of movers to procedures through a type system [19] that analyzes their bodies. This enables the sound introduction of atomic sections that may include recursive procedure calls (as before, soundness relies on ensuring a well-structured sequencing of right and left movers). For example, if the body of a procedure Q consists solely of right mover statements–including nested procedure calls, which are recursively typed–then Q is classified as a right mover procedure. An analogous classification applies to left mover procedures, with the additional requirement that they must terminate when executed in isolation. This termination condition ensures the preservation of failure behaviors: a non-terminating procedure could otherwise unsoundly eliminate potential failures. Notably, although we are reasoning about concurrency, this condition relies solely on the behavior of the procedure when executed in isolation, a surprising and useful aspect of our framework.

The second key contribution of our technique is the ability to soundly transform a parallel construct, where an arbitrary number of threads are spawned and joined immediately afterward, into a sequential composition of their respective code blocks. This transformation again builds on the notion of movers and is achieved through an iterative process that sequences left-mover code blocks first and right-mover code blocks last.

The two contributions of our technique are integrated within a unified framework that supports arbitrary sequential and parallel composition of procedures. Furthermore, the two contributions work in tandem: transforming parallel constructs may yield left or right mover procedures which may further enable the introduction of atomic sections. Our technique is formalized in a core programming language, where the two reduction principles are invoked via specific keywords, and a type system guarantees their correct and sound composition. We note that our notion of movers is more general than that used in Civl; that is, certain statements commute under our definition but not under Civl's. Moreover, our work provides the first formal proof of the soundness of Civl's reduction theory. This proof played a key role in identifying the most general form of commutativity sufficient to ensure soundness (Section 4).

The soundness of this reduction framework is based on non-trivial arguments. For instance, reductions are defined at code level, which means that even a single reduction can apply an unbounded number of times in an execution (where the same procedure is called multiple times). This requires defining a non-trivial strategy for reordering steps in an execution wrt their commutativity properties. Also, the side conditions for using left or right movers are asymmetric, which may seem counterintuitive. For instance, in parallel reduction, right movers are not allowed to fail, and left mover procedures, when run without interference, are required to terminate—the latter is notable because it concerns sequential executions, even though it is applied to concurrent programs. Furthermore, the restriction that right movers must be non-failing applies only to parallel reduction and not to sequential reduction.

We have implemented our technique as an extension to Civl, preserving compatibility with its existing features. To assess its effectiveness, we applied our implementation to a series of challenging case studies: a parallel implementation of a snapshot object [3], the ABD register [4] which simulates shared memory over message passing, the FLASH cache coherence protocol [32], and a non-trivial variant of the Two-Phase Commit protocol. These examples span diverse domains, including concurrent objects, distributed protocols, and hardware cache coherence, demonstrating the broad applicability of our approach. In particular, the first two case studies are concurrent objects for which we prove that they are linearizable [24]. Proving linearizability for these objects is known to be chal-

```
procedure scan() returns (snapshot: [int]StampVal) {
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    (call r1[1] := read(1)) par
    (call r1[2] := read(2));
    (call r2[1] := read(1)) par
    (call r2[2] := read(2));
    if (r1 == r2) {
      snapshot := r1;
      return;
    }
  }
}
```

```
datatype StampVal {
  StampVal(ts: int, value: Value)
}

var mem: [int]StampVal;

action read (i: int) returns (v:
     StampVal) {
  v := mem[i];
}

action write(i: int, v: Value) {
  mem[i] := StampVal(mem[i]->ts +
     1, v);
}

action scan_spec() returns
     (snapshot: [int]StampVal) {
  assume (snapshot := mem);
}
```

Fig. 1: A snapshot object. The scan procedure carries out two consecutive collects, meaning it reads the entire memory in parallel twice. If both collects yield identical results, the procedure returns. Otherwise, it restarts.

lenging, because it can not be done via so-called fixed linearization points, i.e., the effect of a method invocation cannot be mapped to the execution of a fixed statement in the body of the method, and it requires prophecy variables[1].

Reduction was indispensable for our case studies, each of which involves fine-grained access to shared state by an unbounded number and dynamically-created concurrent tasks. The previous version of Civl could not handle these case studies because reduction was applicable only to sequential code fragments consisting solely of actions (and no procedure calls) and had no notion of parallel reduction. We are not aware of any other proof technique based on reduction that can handle our case studies. Without reduction, proofs based purely on inductive invariants would be substantially more complex: the required invariants would be large and difficult to formulate.

2 Overview

We demonstrate our reduction proof technique on an implementation of a concurrent *snapshot* object [3] that provides two methods: `write(i,v)` that writes value `v` to memory cell `i`, and `scan()` which returns a snapshot of the entire memory. We assume that the memory is represented using an array. These methods can be called concurrently from an arbitrary number of threads. We first describe the implementation and the specification we are trying to prove, and then detail the application of our reduction proof technique.

2.1 A Concurrent Snapshot Object

Implementation. Figure 1 lists the code of the snapshot object (`scan_spec` is explained later). Each memory cell holds a timestamped value (a value along with an integer timestamp). For simplicity, we consider a memory with just two cells. The arbitrary-size case is considered in Section 2.4.

The code uses a programming language with regular procedure calls and parallel composition, where each access to the shared memory is encapsulated

```
procedure scan() returns (snapshot: [int]StampVal) {
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    seq-reduce {
      par-reduce {
        (call r1[1] := read_f(1)) par
        (call r1[2] := read_f(2))
      }
      par-reduce {
        (call r2[1] := read_s(1)) par
        (call r2[2] := read_s(2))
      }
      if (r1 == r2) {
        snapshot := r1;
        return;
      }
    }
  }
}
```

```
right action read_f(i: int) returns
      (out: StampVal) {
  var k: int;
  var v: Value;
  if (*) {
    assume k < mem[i]->ts;
    out := StampVal(v, k);
  } else {
    out := mem[i];
  }
}

left action read_s(i: int) returns
      (out: StampVal) {
  var k: int;
  var v: Value;
  if (*) {
    assume k > mem[i]->ts;
    out := StampVal(v, k);
  } else {
    out := mem[i];}
}
```

Fig. 2: An abstraction `scan`. Compared to the original, the two memory reads call the abstracted actions `read_f` and `read_s`, resp. In these actions, $*$ is non-deterministic choice and local variables are initially assigned arbitrary values. The annotations `seq-reduce` and `par-reduce` are related to our reduction technique.

into a so-called *action*. Actions are assumed to execute atomically in a single indivisible step. In this case, we have two actions, `read(i)` reads the `i`-th memory cell, and `write(i,v)` updates the `i`-th memory cell with value `v` and a timestamp incremented by 1 from its current timestamp.

The procedure `scan` consists of a "spin" loop that exits when two consecutive reads of the entire memory yield identical results. A read of the memory is parallelized, each memory cell is read in a different thread. This is written using the `par` keyword in between the two calls to the action `read` (actions are called in the same way as procedures). The meaning of s `par` s' for two statements s and s' is that s and s' are executed in two different threads which are joined before executing the next statement. This ensures that the two reads of the entire memory do not overlap in time. For an expert reader, this corresponds to the fork-join model.

The procedure `write(i,v)` is omitted; it simply calls the homonymous action.

Specification: Linearizability. Our goal is to show that this object is *linearizable*, i.e., each concurrent invocation of `scan` or `write` seems to take effect instantaneously at some point between the call and the return. That is, each concurrent execution of multiple invocations corresponds to a *linearization*–a valid sequence of those invocations where every `scan` returns the memory state resulting from all preceding `write` operations.

The aforementioned sequential semantics of `scan` is defined by the action `scan_spec` in Figure 1, which assumes that the return value equals some instantaneous read of the memory (recall that actions execute in a single indivisible step). Linearizability can be reduced to showing that `scan` is a refinement of `scan_spec`, in a sense that will be made precise later. Note that `scan_spec` may block and this is sound because linearizability does not imply any notion of progress by itself.

Linearizability Proof. The work of Attiya et al.[5] shows that any "unreduced" linearizability proof requires prophecy variables, where an "unreduced" proof is one that attempts to establish a linearization for every possible execution of any number of invocations. Linearizability is equivalent to a standard notion of *trace inclusion* between the concurrent object and an atomic (sequential) specification where every invocation performs a single indivisible step. Traces are sequences of call and return events storing input and return values. Trace inclusion is known to be equivalent to the existence of a composition of a forward and backward simulation relations [37]. Attiya et al. [5] show that there exists no forward simulation from this snapshot object to the corresponding atomic (sequential) specification, which implies the need for using backward simulations (a forward simulation corresponds to a proof using so-called fixed linearization points). Backward simulations are known to correspond to using prophecy variables in a deductive verification context [1,37].

Next, we present a proof using our reduction technique which avoids the use of notoriously challenging prophecy variables. After a step of abstraction, the reads will become movers and they can be reordered to form an atomic section which is a "direct" refinement of `scan_spec`.

This goes beyond previous reduction techniques since the `scan` implementation nests parallel composition and sequential composition of statements.

2.2 Using Abstraction to Enable Reduction

The actions `read` and `write` do not commute for obvious reasons. To enable reduction, we introduce two abstractions of `read`: a right-mover abstraction `read_f` and a left-mover abstraction `read_s`, which are listed on the right of Figure 2. In general, an abstraction of an action over-approximates its effect on the global state and the set of possible return values.

A `read` abstraction commutes to the right of a write if it can return a value with an older timestamp than the one in memory, meaning any value it returns before the write remains valid afterward. We introduce this behavior via a nondeterministic choice: `read_f` can either return the timestamped value in memory, or an arbitrary timestamped value provided that the timestamp is strictly smaller than the timestamp in memory. The left mover abstraction `read_s` is very similar except that the returned timestamp in the "arbitrary" case should be strictly higher than the timestamp in memory. We note that designing such abstractions requires creativity, as in any other deductive proof system. The advantage here is that they lead to more ergonomic proofs—both more succinct and less tedious. Moreover, soundness of mover abstractions is a local property, as it concerns only pairs of actions, and the induced reduction significantly simplifies subsequent reasoning.

Figure 2 lists an abstraction of the `scan` procedure which calls the abstract actions `read_f` and `read_s` during the first and second read of the memory, resp. The occurrences of `seq-reduce` and `par-reduce` are explained below and should be ignored for now. This is a sound abstraction in the sense that any concurrent execution of the original snapshot implementation is also possible when `scan` is replaced by this abstract version. Soundness is a straightforward consequence of the fact that `read_f` and `read_s` over-approximate the behavior of `read`.

In the following, we show that this abstraction of `scan` is a refinement of the atomic action `scan_spec`, which concludes the linearizability proof.

2.3 Reducing Parallel Statements

We prove that it is sound to treat all reads in the abstract `scan` from Figure 2 as executing atomically, without interference from other threads. The first reduction step removes the use of parallel composition, represented by the annotation `par-reduce`. The first occurrence of `par-reduce` relies on `read_f(2)` being a right mover and thus,

```
(call r1[1] := read_f(1)) par (call r1[2] := read_f(2))
```

can be rewritten to

```
call r1[1] := read_f(1); call r1[2] := read_f(2)
```

where the parallel composition `par` has been replaced by sequential composition `;`. This fixes an order between the two actions, but interference is still allowed in between the two calls (i.e., after `read_f(1)` completes but before `read_f(2)` starts).

Indeed, for any interleaving where `read_f(2)` executes before `read_f(1)`, the right moverness of `read_f(2)` implies that it can be soundly swapped to the right of all actions that execute until `read_f(1)` and `read_f(1)` itself (here, soundness means preserving the final state and all return values of actions or procedures).

Dually, the second occurrence of `par-reduce` relies on `read_s(1)` being a left mover in order to reduce the parallel composition `par` to sequential composition.

```
seq-reduce { // -> atomic {
  call r1[1] := read_f(1);
     // right
  call r1[2] := read_f(2);
     // right
  call r2[1] := read_s(1);
     // left
  call r2[2] := read_s(2);
     // left
  if (r1 == r2) {
    snapshot := r1;
    return;
  }
}
```

Fig. 3: A reduced loop iteration.

The result of reducing the two parallel statements is shown on the left, in Figure 3. For simplicity, we write just the loop iteration. The sequence of reads is now a sequence of right movers followed by a sequence of left movers and we can use Lipton's reduction in order to rewrite it as an atomic section (the conditional and the assignments that follow the reads are accessing local variables and can be reordered in any direction, to the left in this case). Invoking this reduction principle is done via the keyword `seq-reduce`. The final reduced form of `scan` will group all reads and the if conditional inside an atomic section, marked using the keyword `atomic`. It is now quite straightforward to show that `scan` refines the atomic action `scan_spec`:

- every iteration where the conditional fails has no effect on the return value,
- if the conditional succeeds, then for every cell, the timestamps returned by `read_f` and `read_s` are identical. This indicates that both reads accessed the current memory state, and the values were not chosen arbitrarily. Specifically, if `read_f` had returned a timestamp smaller than the one in memory,

```
procedure scan() returns (snapshot:
     [int]StampVal){
  var r1: [int]StampVal;
  var r2: [int]StampVal;
  while (true) {
    seq-reduce {
      call r1 := collect_f(n); // right
      call r2 := collect_s(n); // left
      if (r1 == r2) {
        snapshot := r1;
        return;
      }
    }
  }
}
```

```
right procedure collect_f(n: int)
     returns (r: [int]StampVal) {
  var out: StampVal;
  if (n == 0) { return; }
  else {
    par-reduce {
      (call r := collect_f(n-1)) par
      (call out := read_f(n))
    }
    r[n] := out;}
}
left procedure collect_s(n: int) returns
     (r: [int]StampVal) {
  var out: StampVal;
  if (n == 0) { return; }
  else {
    par-reduce {
      (call out := read_s(n)) par
      (call r := collect_s(n-1))
    }
    r[n] := out;}
}
```

Fig. 4: Applying reduction on an abstraction of the `scan` procedure for an unbounded size memory.

then `read_s` could not have returned the same timestamp, as it only returns timestamps strictly greater than those currently in memory (which increase monotonically). The action `read_s` cannot return a timestamp greater than the one in memory for similar reasons.

2.4 The Unbounded Memory Case: Reductions for Structured Code

Figure 4 lists a reduction proof for an extension of the previous `scan` implementation to an unbounded size memory. Memory reads are performed inside two recursive procedures `collect_f` and `collect_s` which use the corresponding `read_f` and `read_s` actions to read memory cells.

Notably, this demonstrates an extension of Lipton's reduction to structured programs, code that contains procedure calls. For compositionality, we introduce a moverness type for procedures, and use that moverness type in a similar way to Lipton's reduction. After a parallel reduction step (explained below), `collect_f` and `collect_s` are typed as right and left procedures, respectively. This enables a reduction step that yields an atomic section encompassing an entire iteration of the outer `scan` loop, as in the previous case.

The parallel reductions are now performed inside each of the two recursive procedures, and rely on similar arguments as above (the moverness of the `read_f` and `read_s` actions). After this reduction step, they contain no more parallel composition, and since all the actions they perform have the same moverness type, this type can be exported at the procedure level. The left moverness case requires that the procedure terminates *when executed in isolation*, which is obvious here because the parameter decreases and it is bounded below by 0.

Once both reduction steps have been applied, proving that the `scan` procedure is a refinement of `scan_spec` is similar to the bounded case presented above.

Formalizing the correctness of this reduction technique, which handles both structured code and parallel composition, is non-trivial. The next section in-

$$A \in \mathit{ActionName} \quad Q \in \mathit{ProcName} \quad X \in \mathit{ActionName} \cup \mathit{ProcName}$$

$$\begin{array}{rcl}
 & \mathit{Val} & \ni \text{☣} \\
v \in & \mathit{Var} & = \mathit{GVar} \cup \mathit{LVar} \\
g \in & \mathit{GStore} & = \mathit{GVar} \to \mathit{Val} \\
\ell \in & \mathit{LStore} & = \mathit{LVar} \rightharpoonup \mathit{Val} \\
\sigma \in & \mathit{Store} & = \mathit{Var} \rightharpoonup \mathit{Val} \\
\rho \in & \mathit{Gate} & = 2^{\mathit{Store}} \\
\tau \in & \mathit{Trans} & = 2^{\mathit{Store} \times \mathit{Store}} \\
\iota, o \in & \mathit{IOMap} & = \mathit{LVar} \rightharpoonup \mathit{LVar} \\
x \in & \mathit{LVar} & \\
I, O \in & 2^{\mathit{LVar}} & \\
M \in & \mathit{MoverType} & = \{B, R, L, N, \top\} \\
F \in & \{\mathit{true}, \mathit{false}\} &
\end{array}
\qquad
\begin{array}{rcl}
s \in \mathit{Stmt} & ::= & \texttt{skip} \mid \texttt{if}\ x\ s\ s \\
 & & \mid s\,;s \mid s\ \texttt{par}\ s \\
 & & \mid \texttt{call}\ (X, \iota, o) \\
 & & \mid \texttt{atomic}\ s \\
 & & \mid \texttt{par-reduce}\ s\ \texttt{par}\ s \\
 & & \mid \texttt{seq-reduce}\ s \\
\mathit{Action} & ::= & (I, O, \rho, \tau, M, F) \\
\mathit{ProcSig} & ::= & (I, O, M, F) \\
\mathit{as} & \in & \mathit{ActionName} \to \mathit{Action} \\
\mathit{ps} & \in & \mathit{ProcName} \to \mathit{ProcSig} \\
\mathcal{P} \in \mathit{Prog} & = & \mathit{ProcName} \to \mathit{Stmt}
\end{array}$$

Fig. 5: RedPL: Syntax

troduces a simple yet expressive programming language used to reason about correctness in the following sections.

3 RedPL: Syntax and Semantics

In this section we present our core programming language RedPL to formalize our approach to reduction. Our language is inspired by RefPL [31]. Figure 5 summarizes the syntax of RedPL.

Variables and stores. We assume there is a fixed set of *global variables* GVar and a fixed set of *local variables* LVar such that GVar and LVar are disjoint. The set of *variables* Var is the union of GVar and LVar. A *store* σ is a partial map from variables to *values*. We write $\sigma' \subseteq \sigma$ if σ is an extension of σ', $\sigma|_V$ for the restriction of σ to V, $\sigma - V$ for $\sigma|_{dom(\sigma)\setminus V}$, $\sigma[\sigma']$ for the store that is like σ' on $\mathrm{dom}(\sigma')$ and otherwise like σ, and $g \cdot \ell$ for the combination of *global store* g and *local store* ℓ.

Actions. RedPL models uninterrupted execution by a thread using atomic actions [11,30]. We assume there is a fixed set of actions with names from the set $\mathit{ActionName}$. All accesses to global variables are confined to actions. The action map as maps each $A \in \mathit{ActionName}$ to a tuple $\mathit{as}(A) = (I, O, \rho, \tau, M, F)$. The set of input variables I and the set of output variables O are each a subset of LVar. The gate ρ is a set of stores such that $\mathrm{dom}(\ell) = I$ for each $g \cdot \ell \in \rho$. The transition relation τ is a relation over stores such that $\mathrm{dom}(\ell) = I$ and $\mathrm{dom}(\ell') = O$ for each $(g \cdot \ell, g' \cdot \ell') \in \tau$. Executing the action from a store σ that does not satisfy the gate (i.e., $\sigma \notin \rho$) fails the execution. Otherwise, every transition (σ, σ') in τ describes a possible atomic state transition from σ (over $\mathit{GVar} \cup I$) to σ' (over $\mathit{GVar} \cup O$). The mover type M of the action is a member of the set $\mathit{MoverType}$ [19]; it captures succintly the nature of commutativity of this action compared to other actions defined by as. The failure type F is a Boolean value that indicates whether it is possible for this action to fail. If $F = \mathit{false}$, then the gate must include all possible stores over $\mathit{GVar} \cup I$.

Our formalization does not provide concrete syntax for the bodies of atomic actions, instead choosing to model them abstractly using a symbolic transition system. Our modeling approach is general and allows actions to be arbitrary and potentially failing computations over global, input, and output variables. Specifically, actions can model a variety of statements—asserts, assumes, (nondeterministic) assignments, choice, and sequencing.

Procedures. RedPL models preemptible concurrent execution using procedures. We assume there is a fixed set of procedures with names from the set $\mathit{ProcName}$ which is disjoint from $\mathit{ActionName}$. We split the specification of procedures into two maps—the signature ps and the program $\mathcal{P}$. An important aspect of our formalization is to transform procedure bodies while keeping their signature fixed. Splitting the specification of procedure behavior into the signature map and the program aids our formalization.

The signature map ps maps each $Q \in \mathit{ProcName}$ to a tuple (I, O, M, F). The set of input variables I and the set of output variables O are each a subset of LVar. When Q is called, its local store gets a binding for each variable in LVar. The mover type M is a member of the set $\mathit{MoverType}$. The failure type F is a Boolean indicating whether it is possible for the execution of the procedure to fail. Our type checker, described later, checks the consistency of M and F against the body of the procedure.

The program $\mathcal{P}$ maps each $Q \in \mathit{ProcName}$ to a statement s that is executed when Q is called. The primitive statement `skip` does nothing; it serves as a marker in the formal operational semantics explained later. The statement `if` $x\ s_1\ s_2$ looks up the value of x in the local store and continues to execute s_1 if the value is *true* or s_2 if the value is *false*. The statement $s_1\,;s_2$ executes s_1 followed by s_2. The statement s_1 `par` s_2 executes both s_1 and s_2 in parallel.

The statement `call` (X, ι, o) calls either an action or a procedure. Parameter passing is expressed using an *input map* ι from I to LVar, and an injective *output map* o from O to LVar. For both ι and o, the domain is callee's formals and the range is caller's actuals. Input variables are immutable, since they are not mapped to by output maps and the variables of a procedure are not modified anywhere else. An action call is the only way to access global variables or to modify either the global or the local store.

The statement `atomic`s executes s with preemptions disabled, i.e., the statement s is executed to completion before any other concurrent execution is scheduled. The statement `par-reduce`s_1 `par` s_2 expresses the programmer intention to reduce the parallel computation s_1 `par` s_2 to the sequential computation $s_1\,;s_2$. The statement `seq-reduce` s expresses the programmer intention to reduce the statement s to `atomic` s. A statement s is *atomic-free* if s does not have any occurrences of `atomic`. A statement s is *reduce-free* if s does not have any occurrences of `seq-reduce` or `par-reduce`. A program $\mathcal{P}$ is *atomic-free* if $\mathcal{P}(Q)$ is atomic-free for all $Q \in \mathrm{dom}(\mathcal{P})$.

Although RedPL does not have explicit support for loops and nondeterministic choice, both can be modeled. We can model a loop using a recursive procedure. We can model nondeterministic choice using the conditional statement `if` $x\ s\ s$ after assigning a nondeterministically chosen value to the local variable x (via an action).

$$
\begin{aligned}
f &::= (\ell, SC[s]) \\
t &::= \mathsf{Lf}\ f \mid \mathsf{Nd}\ f\ \bar{t} \\
\mathcal{T} &::= \{t, \dots, t\} \\
c &::= (g, \mathcal{T}) \mid \lightning
\end{aligned}
\qquad
\begin{aligned}
SC &::= \bullet_s \mid SC\,;s \mid \mathtt{in\text{-}atomic}\ SC \\
&\quad\ \mid \mathtt{in\text{-}seq\text{-}reduce}\ SC \\
TC &::= \bullet_t \mid \mathsf{Nd}\ f\ \bar{t}\,TC\bar{t} \\
PC &::= \{TC\} \uplus \mathcal{T} \\
LC &::= PC[\mathsf{Lf}\ (\bullet_\ell, SC)]
\end{aligned}
$$

$$
\textbf{(proc call)}\ (g, PC[\mathsf{Lf}\ (\ell, SC[\mathtt{call}\ (Q, \iota, o)])]) \to
$$
$$
(g, PC[\mathsf{Nd}\ (\ell, SC[\mathtt{call}\ (Q, \iota, o)])\ \mathsf{Lf}\ (\{v \mapsto \text{☣} \mid v \in LVar\}[\ell \circ \iota], \mathcal{P}(Q))])
$$

$$
\textbf{(return)}\ (g, PC[\mathsf{Nd}\ (\ell, SC[\mathtt{call}\ (Q, \iota, o)])\ \mathsf{Lf}\ (Q, \ell', \mathtt{skip})]) \to
$$
$$
(g, PC[\mathsf{Lf}\ (\ell[\ell' \circ o^{-1}], SC[\mathtt{skip}])])
$$

$$
\textbf{(fork)}\ (g, PC[\mathsf{Lf}\ (\ell, SC[s_1\ \mathtt{par}\ s_2])]) \to
$$
$$
(g, PC[\mathsf{Nd}\ (\ell, SC[s_1\ \mathtt{par}\ s_2])\ \mathsf{Lf}\ (\ell, s_1)\ \mathsf{Lf}\ (\ell, s_2)])
$$

$$
\textbf{(join)}\ (g, PC[\mathsf{Nd}\ (\ell, SC[s_1\ \mathtt{par}\ s_2])\ \mathsf{Lf}\ (\ell_1, \mathtt{skip})\ \mathsf{Lf}\ (\ell_2, \mathtt{skip})]) \to
$$
$$
(g, PC[\mathsf{Lf}\ (\ell[\ell_1|_{mod(s_1)}][\ell_2|_{mod(s_2)}], SC[\mathtt{skip}])])
$$

$$
\textbf{(action call)}\ \frac{as(A) = (\_, \_, \rho, \tau) \quad (g\cdot(\ell \circ \iota), g'\cdot\hat{\ell}) \in \rho \circ \tau \qquad \ell' = \ell[\hat{\ell} \circ o^{-1}]}{(g, LC[\ell][\mathtt{call}\ (A, \iota, o)]) \to (g', LC[\ell'][\mathtt{skip}])}
$$

$$
\textbf{(action fail)}\ \frac{as(A) = (\_, \_, \rho, \_, \_, \_) \qquad g\cdot(\ell \circ \iota) \notin \rho}{(g, LC[\ell][\mathtt{call}\ (A, \iota, o)]) \to \lightning}
\qquad
\textbf{(branch)}\ \frac{s' = \begin{cases} s_1 & \ell[x] = \mathit{true} \\ s_2 & \ell[x] = \mathit{false} \end{cases}}{(g, LC[\ell][\mathtt{if}\ x\ s_1\ s_2]) \to (g, LC[\ell][s'])}
$$

$$
\textbf{(skip)}\ (g, LC[\ell][\mathtt{skip}\,; s]) \to (g, LC[\ell][s]) \qquad \textbf{(stop)}\ (g, \{\mathsf{Lf}\ (\_, \mathtt{skip})\} \uplus \mathcal{T}) \to (g, \mathcal{T})
$$

$$
\textbf{(atomic enter)}\ (g, LC[\ell][\mathtt{atomic}\ s]) \to (g, LC[\ell][\mathtt{in\text{-}atomic}\ s])
$$

$$
\textbf{(atomic exit)}\ (g, LC[\ell][\mathtt{in\text{-}atomic}\ \mathtt{skip}]) \to (g, LC[\ell][\mathtt{skip}])
$$

$$
\textbf{(par-reduce enter)}\ (g, PC[\mathsf{Lf}\ (\ell, SC[\mathtt{par\text{-}reduce}\ s_1\ \mathtt{par}\ s_2])]) \to
$$
$$
(g, PC[\mathsf{Nd}\ (\ell, SC[\mathtt{par\text{-}reduce}\ s_1\ \mathtt{par}\ s_2])\ \mathsf{Lf}\ (\ell, s_1)\ \mathsf{Lf}\ (\ell, s_2)])
$$

$$
\textbf{(par-reduce exit)}\ (g, PC[\mathsf{Nd}\ (\ell, SC[\mathtt{par\text{-}reduce}\ s_1\ \mathtt{par}\ s_2])\ \mathsf{Lf}\ (\ell_1, \mathtt{skip})\ \mathsf{Lf}\ (\ell_2, \mathtt{skip})]) \to
$$
$$
(g, PC[\mathsf{Lf}\ (\ell[\ell_1|_{mod(s_1)}][\ell_2|_{mod(s_2)}], SC[\mathtt{skip}])])
$$

$$
\textbf{(seq-reduce enter)}\ (g, LC[\ell][\mathtt{seq\text{-}reduce}\ s]) \to (g, LC[\ell][\mathtt{in\text{-}seq\text{-}reduce}\ s])
$$

$$
\textbf{(seq-reduce exit)}\ (g, LC[\ell][\mathtt{in\text{-}seq\text{-}reduce}\ \mathtt{skip}]) \to (g, LC[\ell][\mathtt{skip}])
$$

Fig. 6: RedPL: Operational semantics for program $\mathcal{P}$

3.1 Semantics

Figure 6 presents the operational semantics of RedPL as a transition relation $\rightarrow$ over *configurations*. Each configuration is either a failure ↯ or a pair $(g, \mathcal{T})$ comprising a global store g and a finite multiset $\mathcal{T}$ of threads. Each thread is a tree (which generalizes a call stack); new leaf nodes (Lf) are created via the `call` and `par` statements. Both of these statements block the caller in an internal node Nd until the leaf nodes are finished. Each tree node contains a *frame* (ℓ, s), where ℓ is the local store and s is the remaining statement to execute.

In the definition of $\rightarrow$ we use several evaluation contexts that have a unique hole $\bullet$ which marks the evaluation position; filling the hole is denoted by $\cdot[\cdot]$. SC is the statement context, which is a statement with a hole $\bullet$. $SC[s]$ is a statement with s in evaluation position. In addition to a hole $\bullet$, there are three other statement contexts. The context $SC\,;s$ finishes evaluating SC before moving on to s. The context `in-atomic` SC is introduced when `in-atomic` is entered. The context `in-seq-reduce` SC is introduced when `in-seq-reduce` is entered. PC is a multiset of thread trees with a hole $\bullet$ in one of the trees. We have additional conditions on this multiset PC: (1) Trees that do not contain a hole $\bullet$ do not have the `in-atomic` statement in them. (2) If t is the tree with the hole $\bullet$, and if there is any `in-atomic` statement in t, then it must be on the unique path from root of t to the hole. These conditions ensure that an atomic statement executes without interference. The hole in a PC may be filled with an arbitrary tree. LC is a specialization of PC in which the hole is filled with a leaf with holes inside it for a local store and the next statement to be executed.

$$
\begin{aligned}
\mathit{mod}(\texttt{skip}) &= \varnothing \\
\mathit{mod}(\texttt{call}\ (X, \iota, o)) &= \mathrm{img}(o) \\
\mathit{mod}(s_1\,; s_2) &= \mathit{mod}(s_1) \cup \mathit{mod}(s_2) \\
\mathit{mod}(s_1\ \texttt{par}\ s_2) &= \mathit{mod}(s_1) \cup \mathit{mod}(s_2) \\
\mathit{mod}(\texttt{atomic}\, s) &= \mathit{mod}(s) \\
\mathit{mod}(\texttt{par-reduce}\, s_1\ \texttt{par}\ s_2) &= \mathit{mod}(s_1) \cup \mathit{mod}(s_2) \\
\mathit{mod}(\texttt{seq-reduce}\, s) &= \mathit{mod}(s) \\
\mathit{mod}(\texttt{if}\ x\ s_1\ s_2) &= \mathit{mod}(s_1) \cup \mathit{mod}(s_2)
\end{aligned}
$$

Fig. 7: The *mod* function

Figure 6 provides the semantics for a fixed program $\mathcal{P}$ organized as a collection of rules, one for each kind of statement in the hole. The rule **(proc call)** for executing `call` (Q, ι, o) from a context with local store ℓ creates a new leaf and initializes its frame with a local store where the input variables of Q get their values from $\ell \circ \iota$ ($\circ$ means function or relation composition) and the rest of the variables are initialized to the default value ☣. The rule **(return)** for returning from a call updates the caller's local store with the values in the callee's local store using the output map o.

The rule **(fork)** for executing s_1 `par` s_2 creates two leaf nodes for executing s_1 and s_2, each node getting a copy of the local store of the parent. The parent is blocked until both children nodes have finished executing. Then, the modified variables from each child node are written back to the parent's local store in the rule **(join)**. The type checker for RedPL (described in Section 5) checks that the local variables modified by s_1 are not accessed by s_2 and vice-versa. This check ensures that the updates to the local store of the parent from the local stores of the children nodes are conflict-free. The *mod* function, shown in Figure 7, approximates the set of local variables that are modified by a statement.

Atomic actions execute directly in the context of the caller. If the current store does not satisfy the gate of an executed action, the execution stops in the *failure configuration* $\lightning$ (rule **(action fail)**). Otherwise, the execution takes a step according to the transition relation of the action (rule **(action call)**). We refer to transitions in which an atomic action executes as an *action* transition, and all other transitions as *local* transitions. For any action transition t_A of an action A called with input mapping ι_A from a leaf node with local store ℓ in the frame, we define the input parameter binding of the transition as $\textit{in-bind}(t_A) = \ell \circ \iota_A$.

Rule **(branch)** executes a conditional statement. Rule **(skip)** moves the evaluation context forward. Rule **(stop)** removes a finished tree from the multiset of trees. Rule **(atomic enter)** enters an atomic section and rule **(atomic exit)** exits it. Rules **(par-reduce enter)** and **(par-reduce exit)** are similar to **(fork)** and **(join)**, respectively. Rules **(seq-reduce enter)** and **(seq-reduce exit)** enter and exit a `seq-reduce` block.

4 Commutativity of Atomic Actions

We present basic concepts that will be used towards the end of this section to define a well-formed action map. Intuitively, an action map *as* is well-formed if the mover type of each atomic action is correct w.r.t. the entire pool of atomic actions in *as*. For the next few definitions, we fix two actions $X = (I_X, O_X, \rho_X, \tau_X, \_, \_)$ and $Y = (I_Y, O_Y, \rho_Y, \tau_Y, \_, \_)$,

Weakest liberal precondition The weakest liberal precondition $\textit{wlp}(X, \rho_Y)$ is the set of all triples (g, ℓ_X, ℓ_Y) such that X does not fail in $g{\cdot}\ell_X$ and executing X from $g{\cdot}\ell_X$ always leads to a global store g' where gate of Y holds on $g'{\cdot}\ell_Y$.

$$\begin{aligned}\textit{wlp}(X, \rho_Y) = \{(g, \ell_X, \ell_Y) \mid\ & g{\cdot}\ell_X \in \rho_X \wedge \\ & \big(\forall g', \ell'_X : (g{\cdot}\ell_X, g'{\cdot}\ell'_X) \in \tau_X \implies g'{\cdot}\ell_Y \in \rho_Y\big)\}\end{aligned}$$

A consequence of this definition is that if a state satisfies the gate of X but not $\textit{wlp}(X, \rho_Y)$, then there is a way to execute X and get to a state where gate of Y does not hold. This consequence, noted formally below, is used heavily in the proof of soundness of our reduction theorem.

$$\begin{aligned}\forall g, \ell_X, \ell_Y : g{\cdot}\ell_X \in \rho_X \ \wedge\ (g, \ell_X, \ell_Y) \notin \textit{wlp}(X, \rho_Y) \implies \\ \exists \hat{g}, \hat{\ell} :\ (g{\cdot}\ell_X, \hat{g}{\cdot}\hat{\ell}) \in \tau_X \ \wedge\ \hat{g}{\cdot}\ell_Y \notin \rho_Y\end{aligned}$$

Commutes We say X commutes with Y, denoted by $\textit{commutes}(X, Y)$, if executing X followed by Y leads to a state that is also possible by executing Y before X.

$$\begin{aligned}\forall g, g', \bar{g}, \ell_X, \ell_Y, \ell'_X, \ell'_Y : & (g, \ell_X, \ell_Y) \in \textit{wlp}(X, \rho_Y) \wedge (g, \ell_Y, \ell_X) \in \textit{wlp}(Y, \rho_X) \\ & \wedge (g{\cdot}\ell_X, \bar{g}{\cdot}\ell'_X) \in (\rho_X \circ \tau_X) \wedge (\bar{g}{\cdot}\ell_Y, g'{\cdot}\ell'_Y) \in (\rho_Y \circ \tau_Y) \\ & \implies \exists \hat{g} :\ (g{\cdot}\ell_Y, \hat{g}{\cdot}\ell'_Y) \in (\rho_Y \circ \tau_Y) \wedge (\hat{g}{\cdot}\ell_X, g'{\cdot}\ell'_X) \in (\rho_X \circ \tau_X)\end{aligned}$$

Success preservation We say X *preserves the success* of Y, denoted by *preserves-success*(X, Y), if whenever gate of Y and gate of X hold from a state, then any transition of X leads to a state that also satisfies gate of Y.

$$\forall g, \ell_X, \ell_Y :\ g{\cdot}\ell_X \in \rho_X \wedge g{\cdot}\ell_Y \in \rho_Y \implies (g, \ell_X, \ell_Y) \in \textit{wlp}(X, \rho_Y)$$

Failure preservation We say X *preserves the failure* of Y, denoted by *preserves-failure*(X, Y), if whenever X does not fail from a state but Y does, there exists a transition of X from that state that leads to the failure of Y. Equivalently, if X does not fail from a state and cannot make a transition that leads to the failure of Y, then Y does not fail from that state.

$$\forall g, \ell_X, \ell_Y : \ (g, \ell_X, \ell_Y) \in \mathit{wlp}(X, \rho_Y) \implies g{\cdot}\ell_Y \in \rho_Y$$

Well-formed action map. We have 5 types: right-mover ($\mathbf{R}$), left-mover ($\mathbf{L}$), both-mover ($\mathbf{B}$), non-mover ($\mathbf{N}$), top ($\top$). These types form a lattice under a partial order defined as: $\mathbf{B} \sqsubseteq a \sqsubseteq \mathbf{N} \sqsubseteq \top$ where $a \in \{\mathbf{R}, \mathbf{L}\}$. Let the join operator for this partial order be denoted by $\sqcup$. An action map *as* is *well-formed* if for all action $A \in \mathit{ActionName}$, if $\mathit{as}(A) = (I_A, O_A, \rho_A, \tau_A, M_A, F_A)$, then the following conditions are satisfied:

1. $M_A \neq \top$.
2. If $M_A \sqsubseteq \mathbf{L}$, then for all $X \in \mathit{ActionName}$:

$$\begin{array}{ll} \texttt{(L1)} & \mathit{preserves\text{-}success}(X, A) \\ \texttt{(L2)} & \mathit{preserves\text{-}failure}(A, X) \\ \texttt{(L3)} & \mathit{commutes}(X, A) \end{array}$$

3. If $M_A \sqsubseteq \mathbf{R}$, then for all $X \in \mathit{ActionName}$:

$$\begin{array}{ll} \texttt{(R1)} & \mathit{preserves\text{-}success}(A, X) \\ \texttt{(R2)} & \mathit{commutes}(A, X) \end{array}$$

4. If $F_A = \mathit{false}$, then $\rho_A = \{\, g{\cdot}\ell_A \mid g \in \mathit{GStore} \wedge \ell_A \in \mathit{LStore} \wedge \mathrm{dom}(\ell_A) = I_A \,\}$.

There are important differences between our conditions for a well-formed action map and those in prior work on Civl [28]. Our conditions are weaker and we were able to identify these in the process of writing the proof for the soundness of our reduction techniques (Theorem 1).

First, our definition of *commutes*(X, Y) is weaker. The corresponding condition for Civl only assumes ρ_X and ρ_Y in the initial state while verifying that X and Y can be reordered. In contrast, our condition makes the stronger assumption $\mathit{wlp}(X, \rho_Y)$ and $\mathit{wlp}(Y, \rho_X)$ in the initial state. Intuitively, our check requires reordering X and Y only for those initial states from which it is impossible to fail for any ordering of X and Y. The following example illustrates this difference. Let S be a shared set, and define actions A and B by $A(j) : \ S := S \setminus \{j\}$, $B(i) :$ `assert` $i \in S$ where the assertion is the gate of B. Under our commutativity condition, A is a right mover and B is a left mover, but this does not hold under Civl's original condition.

Second, Civl performs two separate checks, backward preservation and nonblocking, for left movers. Backward preservation requires a left mover A to demonstrate for every action X that if ρ_X holds after a step of A then ρ_X also holds before the step. The nonblocking check requires A to either fail or take a step from every initial state. Instead, we have a single failure preservation check that requires less of A than the combination of backward preservation and nonblocking requirements. Intuitively, our check requires A to take a step only when trying to preserve a failure of the action X. For example, if X does not have any failing behavior, failure preservation would hold trivially, but nonblocking check for A could still be nontrivial.

5 Types for Reduction

In this section, we exploit mover types of atomic actions to check that the application of `par-reduce` and `seq-reduce` in a program are applicable to achieve sound reduction. We achieve this goal by defining two helper functions on statements—*may-fail* and *mover-type*. The function *may-fail* propagates the failure types of actions to statements by using a conservative static analysis. The function *mover-type* lifts mover types of actions to statements using an effect system [19]. Together, these functions allow us to define a well-typed program which implies that applications of reduction in the program are valid.

Failure typing for statements We compute *may-fail*(s) for any statement s by conservatively propagating the failure types of atomic actions.

$$
\begin{aligned}
\textit{may-fail}(\texttt{skip}) &= \textit{false}\\
\textit{may-fail}(\texttt{call}\ (A,\iota,o)) &= \textit{may-fail}(A)\\
\textit{may-fail}(\texttt{call}\ (Q,\iota,o)) &= \textit{may-fail}(Q)\\
\textit{may-fail}(s_1\,;s_2) &= \textit{may-fail}(s_1) \lor \textit{may-fail}(s_2)\\
\textit{may-fail}(s_1\ \texttt{par}\ s_2) &= \textit{may-fail}(s_1) \lor \textit{may-fail}(s_2)\\
\textit{may-fail}(\texttt{atomic}\ s) &= \textit{may-fail}(s)\\
\textit{may-fail}(\texttt{par-reduce}\ s_1\ \texttt{par}\ s_2) &= \textit{may-fail}(s_1) \lor \textit{may-fail}(s_2)\\
\textit{may-fail}(\texttt{seq-reduce}\ s) &= \textit{may-fail}(s)\\
\textit{may-fail}(\texttt{if}\ x\ s_1\ s_2) &= \textit{may-fail}(s_1) \lor \textit{may-fail}(s_2)
\end{aligned}
$$

Fig. 8: The *may-fail* function.

Mover typing for statements Given a well-formed action map *as* and a procedure signature map *ps*, we define *mover-type* function, which assigns a mover type to a statement. To define this function, we first define the sequential composition of mover types in the table below. Using this table, we can define *mover-type* recursively as follows:

$$
\begin{aligned}
\textit{mover-type}(\texttt{skip}) &= \mathbf{B}\\
\textit{mover-type}(s_1\ \texttt{par}\ s_2) &= \top\\
\textit{mover-type}(\texttt{call}\ (A,\iota,o)) &= \textit{mover-type}(A)\\
\textit{mover-type}(\texttt{call}\ (Q,\iota,o)) &= \textit{mover-type}(Q)\\
\textit{mover-type}(\texttt{atomic}\ s) &= \textit{mover-type}(s)\\
\textit{mover-type}(\texttt{seq-reduce}\ s) &= \textit{mover-type}(s)\\
\textit{mover-type}(\texttt{par-reduce}\ s_1\ \texttt{par}\ s_2) &= \textit{mover-type}(s_1; s_2)\\
\textit{mover-type}(s_1\,;s_2) &= \textit{mover-type}(s_1); \textit{mover-type}(s_2)\\
\textit{mover-type}(\texttt{if}\ x\ s_1\ s_2) &= \textit{mover-type}(s_1) \sqcup \textit{mover-type}(s_2)
\end{aligned}
$$

;	B	L	R	N	⊤
B	B	L	R	N	⊤
R	R	N	R	N	⊤
L	L	L	⊤	⊤	⊤
N	N	N	⊤	⊤	⊤
⊤	⊤	⊤	⊤	⊤	⊤

Fig. 9: The *mover-type* function.

For example, consider the following seq-reduce statement from the example in the overview. The mover type assigned to the seq-reduce block will be **N**.

```
seq-reduce {
    par-reduce {(call r1[1] := read_f(1)) par (call r1[2] := read_f(2))};
    par-reduce {(call r2[1] := read_s(1)) par (call r2[2] := read_s(2))};
}
```

Each call to `read_f` within the first par-reduce has **R** type. The par-reduce statement then gets the mover type of $\mathbf{R};\mathbf{R} = \mathbf{R}$ from the table. Similarly, the second par-reduce contains calls to `read_s`, which are each **L**, and the par-reduce statement gets the mover type of $\mathbf{L}; \mathbf{L} = \mathbf{L}$. Now, the statement inside seq-reduce composes the two par-reduce blocks sequentially: the first has type $\mathbf{R}$, and the second has type $\mathbf{L}$. The sequential composition $\mathbf{R}; \mathbf{L}$ results in an $\mathbf{N}$ type, which is then assigned to the entire seq-reduce block.

The rules above imply that if s is nested inside s' and $\textit{mover-type}(s) = \top$ then $\textit{mover-type}(s') = \top$. Since $\textit{mover-type}(s_1\ \texttt{par}\ s_2) = \top$, if $s_1\ \texttt{par}\ s_2$ is nested inside a statement s, then $\textit{mover-type}(s) = \top$. We will return to this observation when we discuss the rules for well-typed programs below.

Well-typed programs We define a predicate $\textit{well-typed}(s, l)$ where $s \in \textit{Stmt}$ and $l \in 2^{LVar}$. The predicate $\textit{well-typed}(s, l)$ is checking two aspects of a statement. First, it checks that s only accesses the local variables it is allowed to. This check is relevant because if s contains a nested occurrence of two statements s_1 and s_2 executing in parallel, then local variables modified by these two statements must be disjoint from each other. In fact, we check a stronger, yet simpler to check, property that local variables modified by s_1 are neither read nor written by s_2, and vice-versa. This requirement is used only to simplify the formalization. One could consider fresh copies of local variables instead (note that the number of arms in a parallel construct is constant). In our implementation, the arms of the par construct can only be procedure calls, so this restriction just translates to requiring their input and output parameters to be disjoint, which is both natural and easy to check. Second, we check that applications of `par-reduce` and `seq-reduce` have the appropriate mover and failure types on their target statements. This part of the check uses the previously defined functions *mover-type* and *may-fail*.

$$\begin{aligned}
\textit{well-typed}(\texttt{skip}, l) &= \textit{true}\\
\textit{well-typed}(\texttt{atomic}\ s, l) &= \textit{well-typed}(s, l)\\
\textit{well-typed}(\texttt{call}\ (X, \iota, o), l) &= \mathrm{img}(\iota) \subseteq l \wedge \mathrm{img}(o) \subseteq l\\
\textit{well-typed}(\texttt{if}\ x\ s_1\ s_2, l) &= \textit{well-typed}(s_1, l) \wedge \textit{well-typed}(s_2, l) \wedge x \in l\\
\textit{well-typed}(s_1\, ;\, s_2, l) &= \textit{well-typed}(s_1, l) \wedge \textit{well-typed}(s_2, l)\\
\textit{well-typed}(s_1\ \texttt{par}\ s_2, l) &= \textit{well-typed}(s_1, l - \textit{mod}(s_2)) \wedge\\
& \textit{well-typed}(s_2, l - \textit{mod}(s_1))\\
\textit{well-typed}(\texttt{seq-reduce}\ s, l) &= \textit{well-typed}(s, l) \wedge \textit{mover-type}(s) \sqsubseteq \mathbf{N}\\
\textit{well-typed}(\texttt{par-reduce}\ s_1\ \texttt{par}\ s_2, l) &= \textit{well-typed}(s_1, l - \textit{mod}(s_2)) \wedge\\
& \textit{well-typed}(s_2, l - \textit{mod}(s_1)) \wedge\\
& (\textit{mover-type}(s_1) \sqsubseteq \mathbf{L} \vee\\
&\ \ \textit{mover-type}(s_2) \sqsubseteq \mathbf{R} \wedge \neg\textit{may-fail}(s_2))
\end{aligned}$$

Fig. 10: The *well-typed* function

Most rules in the definition above are straightforward. We take a closer look at the rules for parallel and sequential reduction, focusing on the mover-related checks. `par-reduce` s_1 `par` s_2 checks that one of two cases apply: either s_1 is a left mover or s_2 is a right mover and must not fail. Intuitively, in both cases, all the code in s_1 may be commuted before all the code in s_2. For instance, here is an example illustrating why the right mover statement s_2 is not allowed to fail. Let `x` be a shared integer variable, and let `read` and `inc` be actions, where `read` is a right-mover that first asserts `x > 0` (this corresponds to its gate) and then

```
var x: int;

right action read() returns (out: int) {
  assert x > 0;
  assume out <= x;
}

action inc() {
  x := x + 1;
}
```

```
procedure Q {
  par-reduce {
    call inc() par (call read())
  }
}

procedure Q’ {
  call inc();
  call read();
}
```

Fig. 11: Example illustrating need for right movers do not fail condition

reads the value of `x` (exactly or less), and `inc` is a non-mover that increments `x` by 1 (it is not a left-mover because it does not satisfy failure preservation, and it is not a right-mover because it does not commute to the right of a read). Let Q be a procedure in the original program P, and assume that it gets reduced to Q’ in the reduced program P’ by an application of par-reduce (as shown in the snippet above). Assuming an initial state where `x = 0`, P can fail (if `read` executes first) but P' has no execution that fails (since `inc` always executes before `read`). This is problematic since the reduction "hides" failures which goes against sound reasoning. Hence we require that right movers do not fail. This is what the helper function *may-fail* checks.

`seq-reduce` s checks that $\textit{mover-type}(s) \sqsubseteq \mathbf{N}$ which implies that any execution through s is of a form $\mathbf{R}^*{\cdot}\mathbf{N}?{\cdot}\mathbf{L}^*$, and therefore s can be converted to an atomic section [19]. In this case, the statement s must not have any unreduced parallel statement (whose mover type is $\top$) nested inside it. But it is possible for s to have a reduced parallel statement (whose mover type could be different from $\top$) nested inside it. This flexibility is important in practice and is particularly useful for our case studies described in Section 8.

A program $\mathcal{P}$ is *well-typed* if for all $Q \in \text{dom}(\mathcal{P})$, if $\textit{ps}(Q) = (\_, \_, M, F)$ then the following hold: (1) $\textit{well-typed}(\mathcal{P}(Q), \textit{LVar})$, (2) $\textit{mover-type}(\mathcal{P}(Q)) \sqsubseteq M$, and (3) $\textit{may-fail}(\mathcal{P}(Q)) \Rightarrow F$. The well-typed predicate is computed separately for each procedure Q in the program using the signatures of all procedures and actions called by Q. First, the body of Q is checked to be well-typed w.r.t. the set of all local variables. Second, the mover type of the body of Q must be stronger than the annotated mover type M of Q. This check ensures that the type checking of procedures that call Q will succeed even if Q was inlined at the call site. In essence, this means that the mover type of any procedure is valid regardless of the specific execution taken in completing a call to that procedure. Third, the failure type of Q is checked to be a conservative approximation of the failure type of the body of Q. We use the notion of well-typed programs in Section 6 to state the soundness theorem of our reduction technique.

6 Reduction for RedPL Programs

In this section, we give a meaning to the `seq-reduce` and `par-reduce` annotations in RedPL programs, and state the related soundness theorem. Soundness is stated in terms of a *refinement* relation between programs that we define hereafter.

A configuration $(g, \mathcal{T})$ is *initial* if it contains an arbitrary number of threads that are about to execute a well-typed statement, i.e., for all $t \in \mathcal{T}$, there exist ℓ and s such that $t = Lf(\ell, s)$, *well-typed*(s, LVar), and s is atomic-free and reduce-free. A configuration $(g, \mathcal{T})$ is *final* if $\mathcal{T} = \varnothing$, i.e., all threads have finished executing successfully. The failure configuration $\lightning$ is also final.

Given two well-typed programs $\mathcal{P}$ and $\mathcal{P}'$, we say $\mathcal{P}$ refines $\mathcal{P}'$ (denoted $\mathcal{P} \preccurlyeq \mathcal{P}'$) if the following two properties hold for all initial configurations $(g, \mathcal{T})$:
(P1) If there is an execution of $\mathcal{P}$ that fails from the initial configuration $(g, \mathcal{T})$, then there also is an execution of $\mathcal{P}'$ that fails from the same initial configuration:

$$(g, \mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} \lightning \implies (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning$$

(P2) If there exists an execution of $\mathcal{P}$ starting from the initial configuration $(g, \mathcal{T})$ that reaches the final configuration $(g', \varnothing)$, then there also exists an execution of $\mathcal{P}'$ from the same initial configuration, that either reaches the same final configuration or results in a failure:

$$(g, \mathcal{T}) \xrightarrow{\mathcal{P}}{}^{*} (g', \varnothing) \implies (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} (g', \varnothing) \vee (g, \mathcal{T}) \xrightarrow{\mathcal{P}'}{}^{*} \lightning$$

The refines relation is transitive, i.e., if $\mathcal{P}_1 \preccurlyeq \mathcal{P}_2$ and $\mathcal{P}_2 \preccurlyeq \mathcal{P}_3$, then $\mathcal{P}_1 \preccurlyeq \mathcal{P}_3$.

The notation $s[s_2/s_1]$ denotes the result of replacing all occurrences of statement s_1 with statement s_2 inside the statement s. Similarly, the notation $\mathcal{P}[s_2/s_1]$ denotes a new program in which, for every procedure Q, all occurrences of s_1 in the body of Q, as defined in $\mathcal{P}$, are replaced with s_2:

$$\mathcal{P}[s_2/s_1] = \{Q \mapsto \mathcal{P}(Q)[s_2/s_1] \mid Q \in \mathit{ProcName}\}$$

A statement s is *terminating* in program $\mathcal{P}$ if $\mathcal{P}$ does not have any infinite executions from any configuration $(g, \mathtt{Lf}(\ell, s))$ such that *well-typed*(s, LVar). A well-typed program $\mathcal{P}$ is *terminating* if for all $Q \in \mathrm{dom}(\mathcal{P})$ such that *mover-type*$(Q) \sqsubseteq \mathbf{L}$, we have $\mathcal{P}(Q)$ is terminating in $\mathcal{P}$.

Theorem 1. *Let $\mathcal{P}_s$ be an atomic-free and well-typed program. Let*

$$\mathcal{P}_i = \mathcal{P}_s[s_1\,;s_2 \;/\; \texttt{par-reduce}\; s_1 \;\texttt{par}\; s_2]$$
$$\mathcal{P}_r = \mathcal{P}_i[\texttt{atomic}\; s \;/\; \texttt{seq-reduce}\; s]$$

Then, the programs $\mathcal{P}_i$ and $\mathcal{P}_r$ are well-typed. Furthermore, if $\mathcal{P}_i$ is terminating, then: (1) $\mathcal{P}_s \preccurlyeq \mathcal{P}_i$, and (2) $\mathcal{P}_i \preccurlyeq \mathcal{P}_r$. Therefore, $\mathcal{P}_s \preccurlyeq \mathcal{P}_r$.

[Proof Sketch] First, we establish that $\mathcal{P}_s \preccurlyeq \mathcal{P}_i$, thereby showing that it is sound to sequentialize the concurrent behavior inside `par-reduce`. We prove refinement properties (P1) and (P2) separately. The top-level strategy is to rewrite an execution of $\mathcal{P}_s$ into an execution of $\mathcal{P}_i$ such that, for each application of the `par-reduce` rule, the statement s_1 executes before s_2, using an induction on the number of unreduced `par-reduce` applications. Note that $\mathcal{P}_i$ eliminates all occurrences of `par-reduce`, including those nested inside `seq-reduce`. As a consequence, there is no parallelism within any `seq-reduce` application.

Second, we establish that $\mathcal{P}_i \preccurlyeq \mathcal{P}_r$, thereby showing that it is sound to define atomic sections for all code blocks inside `seq-reduce`. This step also proceeds by induction on the number of unreduced `seq-reduce` applications. We show this by rewriting an execution of $\mathcal{P}_i$ into an execution of $\mathcal{P}_r$ in which each code block inside `seq-reduce` is of the form $\mathbf{R}^*\mathbf{N}?\mathbf{L}^*$.

See the full version of the paper[20] for more details.

7 Implementation

We have implemented our proof rule in Civl [29] verifier for layered concurrent programs [28]. Our implementation covers every aspect of our formalization except for the side conditions on termination of left-mover procedures and absence of failures in right-mover statements. For the examples reported in Section 8, the verification of these side conditions was done manually.

Civl is an extension of the Boogie verifier [7] for sequential programs. Similar to Boogie, the implementation of Civl is broadly split into a type checker and a verification-condition generator. The type checker handles basic type analysis and checks in addition that layer annotations on variables, yield invariants, actions, and procedures are consistent with each other. It also checks that the mover type of each procedure is consistent with the mover type inferred from the body of the procedure. The verification-condition generator in Civl eliminates all concurrency features from the input program and produces a collection of sequential procedures annotated with specfications. These sequential procedures encode checks related to mover types of atomic actions, refinement checks for each procedure, and noninterference checks related to yield invariants. The sequential procedures are processed by the standard Boogie flow that converts each procedure to a logical constraint and checks it using an SMT solver.

Our implementation modifies and extends Civl as follows. First, we modified the verification conditions generated for checking mover types of actions according to the rules laid out for well-formed action map in Section 4. The revised rules are more general and therefore applicable in more scenarios. The revised failure preservation check for left movers provide an easier mental model while debugging unsuccessful proofs.

Second, we added mover types to procedures and implemented the checking of these types against procedure bodies, as described in Section 5. Our type checker also accounts for mover types of procedures in determining the degree of program transformation allowed between successive program layers. The ability to reduce programs in a fully nested manner, as described in Section 6, is important to enable a single layer to perform a large chunk of the proof, thus reducing the proof overhead of layers. We allow procedures annotated with mover types to be summarized using preconditions and postconditions. Atomic code fragments with calls to such procedures can now be analyzed without inlining these procedures. Our implementation also handles loops directly in the same manner as recursive procedures.

Finally, we allow parallel calls to be reduced using the parallel reduction technique introduced in this paper. We provide parallel execution of procedure calls rather than parallel execution of statements. This design choice simplified parameter passing between the caller and the callees. The modified local variable analysis described in this paper is unnecessary and replaced by a simple check that the output variables used across all arms of a parallel call are all distinct from each other. Our implementation includes, in the same framework, the proof rule for synchronizing asynchronous calls reported earlier [30].

Our formalization of RedPL in Section 3 makes explicit every application of `par-reduce` and `seq-reduce` in the source program. These annotations are not explicitly declared in a Civl program; instead, our type checker automatically infers information equivalent to them.

8 Evaluation

We evaluate the implementation described above on a diverse set of challenging case studies: a parallelized snapshot object (Section 2), the classic message-passing simulation of shared memory by Attiya, Bar-Noy, and Dolev [4] (ABD), an implementation of the FLASH cache coherence protocol [32], and a version of the Two-Phase Commit protocol. These implementations naturally decompose into procedures and make significant use of dynamic thread creation. Message passing is modeled in the style of RPC: broadcasting and waiting for responses is expressed as a parallel composition of procedure calls, each modifying the receiver's state and returning an acknowledgment.

Example	**#LOC** Total	**#LOC** Impl & Spec	**Time** sec
Snapshot	119	82	0.4
ABD	389	206	1.4
Coherence	608	401	5
2PC	146	111	1.8

Fig. 12: Evaluation metrics.

The evaluation shows that each case study involves substantial nesting of parallel and sequential reductions, leveraging both left and right mover types. This approach helps avoid the need for complex invariants that would otherwise arise from fine-grained interleavings. As common in Civl, the proofs are decomposed into a sequence of refinement steps, some of these steps being "abstraction" steps that are not related to reduction (they introduce non-deterministic abstractions or ghost variables). Also, these proofs rely on using the other features of Civl, e.g., inductive invariants and permission-based reasoning. The latter is particularly useful to enable commutativity reasoning. Many times, commutativity between actions is implied by the distinctness of (some of) their inputs and this is encoded using permissions.

Figure 12 presents quantitative metrics from our evaluation. We report the total lines of code for each proof, along with a separate count for the implementation and specification components. We also report that the wall-clock time required for executing each proof. The ratio of proof annotation lines to the combined lines of implementation and specification ranges from 0.31 (for Two-Phase Commit) to 0.88 (for ABD). Also, all these proofs can be completed in few seconds. In addition, the Civl repository contains approximately 50 further examples, including larger benchmarks (on the order of hundreds to thousands of lines) where the seq-reduce rule applies. We focused on the four case studies in the paper because they incorporate both seq-reduce and par-reduce, and therefore most clearly illustrate the interaction and use of the two rules.

In the following, we give more details about each case study, except for the snapshot object that we already described in Section 2. We present the implementation and the specification we prove, and the use of reduction. The proof files are available in the supplementary material.

8.1 The ABD register

```
type TimeStamp // a set with a total order and a
     lower bound
TS: TimeStamp // global timestamp used to order
     operations
value_store: Map TimeStamp Value
procedure ReadClient(pid) returns (val) {
  old_ts := Begin(pid)
  ts, val := Read(pid, old_ts)
  End(pid, ts);
}
procedure WriteClient(pid, val) {
  old_ts := Begin(pid)
  ts := Write(pid, old_ts)
  End(pid, ts);
}
```

```
action Begin(pid) returns (ts) {
  ts = TS;
}
action Read(pid, old_ts) returns
     (ts, val) {
  assume old_ts <= ts
  assume ts in value_store
  val := value_store[ts]
}
action Write(pid, val) returns (ts) {
  assume old_ts < ts
  assume ts not in value_store
  value_store[ts] := val
}
action End(pid, ts) {
  TS := max(TS, ts)
}
```

Fig. 13: A linearizable specification for ABD. The TS global variable models a clock which is read when operations start and advances when they end.

The ABD algorithm implements a read-write register (shared memory) on top of message passing. It provides two operations Read and Write on a register that is replicated across n replicas for fault tolerance. Less than half of the replicas can crash. The operations are invoked in parallel by a collection of clients.

Each replica stores a timestamped value (timestamp, value) where timestamp comes from a totally ordered set with a lower bound (e.g., natural numbers). Both Read and Write operations have two phases: the QueryPhase and the UpdatePhase. In the QueryPhase, they send Query messages to all replicas in parallel, wait for a quorum (at least half) of replies, and retrieve the reply (t, v) with the maximum timestamp t among the responses. Then they enter the UpdatePhase, where they send Update messages to all replicas. The Read operation sends an Update(t, v) message, while the Write operation sends an Update(t+1, v') where v' is the new value to be written. They both wait for a quorum of acknowledgements before returning. When receiving an Update(t, v) message, a replica updates its store to (t, v) if t is greater than its current timestamp, and replies with an acknowledgment regardless of whether it updates. Upon receiving a Query message, it replies with its current copy.

Specification: Linearizability. Our goal is to prove that this register implementation is linearizable, which we reduce to a refinement check. To capture the condition that each operation appears to take effect atomically between its call and return, we instrument operations to read a global clock at the start, which advances when they end. This clock is aligned with ABD timestamps: each Read returns a value with a timestamp greater than or equal to the clock at invocation, and each Write writes a value with a timestamp strictly greater than the clock at invocation. At the end of an operation, the clock advances as the timestamp read/written by that operation. The resulting specification is shown in Figure 13, where ReadClient and WriteClient wrap the respective operations with Begin and End actions for clock access. Each method contains a single "internal" step (a call to the action Read or Write), reflecting the requirement that they take effect instantaneously. The map value_store, updated by Write and accessed by Read, ensures that reads return previously written values.

```
right procedure {:layer 3} QueryPhase(i: int,
     old_ts: TimeStamp)
  returns (max_ts: TimeStamp, max_value: Value) {
  ...
  par-reduce {
    call max_ts, max_value :=
      QueryPhase(i + 1, old_ts)
    par call ts, value :=
      Query(i, old_ts)  // right
  }
  if (less_than(max_ts, ts)) {
    max_ts := ts; max_value := value;
  }
}
```

```
procedure Read(pid: ProcessId, old_ts:
     TimeStamp) returns (ts:
     TimeStamp, value: Value) {
  ...
  seq-reduce {
    call ts, value :=
     QueryPhase(0, old_ts);   // right
    call UpdatePhase(0, ts, value); //
     left
  }
}
```

Fig. 14: `QueryPhase` and `Read` procedures.

We prove that the "concrete" versions of `ReadClient` and `WriteClient` where the calls to the actions `Read` or `Write` are replaced by calls to the homonymous ABD procedures are a refinement of the abstract specification in Figure 13.

Applying reduction. The goal is to apply reduction to show that the ABD procedures `Read` and `Write` can be rewritten to execute within a *single* atomic section, which is then shown to refine the abstract `Read` and `Write` actions in Figure 13. To achieve this, we introduce an abstraction of the `Query` handler which is a right mover, which in turn ensures that the `Update` handler becomes a left mover. This abstraction enables parallel reductions in both the `QueryPhase` and `UpdatePhase`; for example, in the `QueryPhase` shown below, it allows the recursive procedure to be reduced by sequentializing all `Query` handlers (which initially happened in parallel). This sequentialization, in turn, enables a sequential reduction within the `Read` and `Write` procedures—illustrated below for `Read`—since the `Query` handlers (right movers) are followed by `Update` handlers (left movers).

The abstraction of `Query` allows it to return a timestamp lower than the current replica timestamp. However, it cannot return any arbitrarily low timestamp—it should be greater than or equal to the clock timestamp `old_ts` obtained in `Begin`. This reduction eliminates the need for an inductive invariant that tracks relationships between read and write operations across different stages of their query or update phases.

8.2 Cache coherence

We implement the FLASH cache coherence protocol [32] which is a directory-based MESI cache coherence protocol. The protocol manages consistency across multiple caches in a shared memory multiprocessor system using a centralized directory.

Figure 15 shows our model for the memory (`mem`), directory (`dir`) and a set of caches (`cache`). The memory is indexed by memory addresses (`MemAddr`), while each cache uses local cache addresses (`CacheAddr`) for indexing. Because the memory address space is larger than the cache address space, a hash function maps each memory address to a cache address, allowing multiple memory addresses to correspond to the same cache address. Each cache stores data in cache lines, which contain a memory address, a value, and a state (`Modified, Exclusive, Shared, Invalid`). The directory tracks the status of each memory address across all caches. If a memory address is held in the Modified or Exclusive state by a cache, the directory records this as `Owner(i)`, where `i` is the ID of

```
type MemAddr; type CacheAddr;
function Hash(MemAddr): CacheAddr;
datatype State
  {Modified(), Exclusive(), Shared(), Invalid()}
datatype CacheLine
  {CacheLine(ma: MemAddr, value: Value, state:
      State)}
datatype DirState
  {Owner(i: CacheId), Sharers(iset: Set CacheId)}
```

```
// Implementation state
var {:layer 0,2} mem: [MemAddr]Value;
var {:layer 0,2} dir:
      [MemAddr]DirState;
var {:layer 0,2} cache:
      [CacheId][CacheAddr]CacheLine;
// Specification state
var {:layer 1,3} absMem:
      [MemAddr]Value;
```

Fig. 15: State representation for cache coherence protocol

```
procedure dir_read_exc_req(i: CacheId, ma: MemAddr)
{
 ... // variable initialization
 seq-reduce {
   call dirState := dir_req_begin(ma); // right
   if (dirState is Owner) {
     call value := cache_invalidate_exc
                  (dirState->i, ma, Invalid())
     // non-mover
     call write_mem(ma, value); // both mover
   }
   else {
     call dp := invalidate_sharers(ma,
     dirState->iset); // left
     call value := read_mem(ma); // both mover
   }
   call cache_read_resp(i, ma, value, Exclusive());
     // left
   call dir_req_end(ma, Owner(i)); // left
 }
}
```

```
left procedure invalidate_sharers
(ma: MemAddr, victims: Set CacheId)
     {
  ...
  if (victims == Set_Empty()) {
    return;
  }
  victim := Choice(victims->val);
  victims' := Set_Remove(victims,
     victim);
  par-reduce { // left
   call
     cache_invalidate_shd(victim,
     ma, Invalid())
   par call invalidate_sharers(ma,
     victims')
  }
}
```

Fig. 16: Reduction applied at directory for exclusive state request

the owning cache. Otherwise, the directory records it as Sharers(iset), where iset is the set of cache IDs having the memory address in Shared state.

We implement 5 top-level operations on the cache:

- cache_read and cache_write which read and write a cache entry, respectively.
- cache_evict_req initiates eviction of a cache line.
- cache_read_shd_req and cache_read_exc_req initiate bringing a memory address into the cache in Shared and Exclusive mode, respectively.

We now detail the operation of cache_read_exc_req and the interactions between cache and directory. The cache initiates an exclusive request to the directory via dir_read_exc_req shown in Figure 16. If the directory state for the requested memory address is Owner, it sends an invalidate request to the owner by calling cache_invalidate_exc. The owner is then expected to change its state to Invalid and send the data back to the directory, which writes it to memory by calling write_mem. If the directory state is Sharers, it sends invalidate requests in parallel to all caches in the sharers list by invoking cache_invalidate_shd. The directory then reads the memory by calling read_mem and sends the response back to the orginal cache via cache_read_resp, and ending the request by calling dir_req_end.

Specification. We introduce an abstract memory, absMem. The goal is to show that the cache_write and cache_read operations are refinement of atomic actions that directly read and write from absMem. We hide directory and all cache operations that interact with it. This specification naturally captures the cache coherence property.

Applying reduction. After introducing the ghost variable `absMem` used in the specification, we define a number of abstractions of actions used in the implementation that become movers. The memory operations `read_mem` and `write_mem` are made both movers, the shared invalidate request `cache_invalidate_shd` is made a left mover, the response to a read request at a cache `cache_read_resp` is made a left mover, the actions `dir_req_begin` and `dir_req_end` for reading and updating the directory state are made right and left movers, respectively. These movers enable reduction at many sites in the implementation. In particular, they enable parallel reduction in the invalidate loop (`invalidate_sharers`) to sequentialize them, and subsequently, a sequential reduction on the entire body of the procedure `dir_read_exc_req` for bringing a memory address into the cache in Exclusive mode. This is made precise in Figure 16.

The reduction helps a refinement proof to hide the directory and all the caches so that the read and write operations at cache are refinements of the atomic operations over `absMem`.

8.3 Two-phase Commit (2PC)

Two-phase Commit is a classic distributed protocol used to implement concurrent transactions. A number of *coordinator* processes make a number of *replicas* agree on an order between concurrent transactions. Each transaction is associated with a start time and an end time, and it is submitted to a single coordinator. Two transactions conflict if their time intervals overlap. The goal is to ensure that all committed transactions are not conflicting pairwise.

A coordinator runs in two phases. In the vote phase, it sends vote requests to all replicas which reply with `YES` or `NO` (accept or not a transaction). A replica stores the set of pending transactions (which are not yet committed or aborted) for which it already voted `YES` in a so-called *locked set*, and it answers `YES` iff the incoming vote request concerns a transaction that does not conflict with some transaction in the locked set. In the finalize phase, if all replies are `YES`, the coordinator sends a commit request and otherwise, an abort request. If a replica receives an abort request, it removes the transaction from the locked set.

Specification. We add a ghost variable, `committed_transactions`, which keeps track of all transactions that have been committed. Before adding a transaction to this set (in the coordinator's code), we assert that it does not conflict with any previously committed transaction.

Applying reduction. To enable reduction, we abstract the vote request handler to allow it to non-deterministically respond with `NO` without modifying the state. This abstraction makes the vote request handler a right mover, while the abort request handler becomes a left mover. The commit request handler is a both mover, since it does not change the state. To illustrate, consider two successive vote requests handled by the same replica (requests at different replicas commute, as they access disjoint state). If the transactions conflict, one handler might return `YES` and the other `NO`. Without the abstraction, reordering these handlers isn't sound: if the second executes first, it might respond `YES`, which breaks commutativity. However, with the abstraction, the second handler can non-deterministically return NO, allowing the reordered execution where the first still responds YES. Similar reasoning applies to other combinations of handlers.

As in previous cases, the abstraction enables a combination of parallel reduction and sequential reduction. The parallel reduction is used to sequentialize the two phases, as exemplified below for the vote requests, and then, the sequentialization enables showing that the whole computation for a transaction can be executed within an atomic section.

```
right procedure vote_all(xid: TransactionId, i:
     ReplicaId)
  returns (votes: [ReplicaId]Vote) {
  ...
  if (1 <= i) {
    vr = VoteRequest(xid, i);
    par-reduce {
     (call votes := vote_all(xid, i-1))
     par (call out := vote(vr)); } // right
    }
    votes[i] := out;
}}
```

```
procedure TPC(xid: TransactionId) {
  ...
  seq-reduce {
    call votes := vote_all(xid, n);
      // right
    // locally calculate decision
      based on votes
    call finalize_all(decision,
      xid); // left
  }
  ...
}
```

Fig. 17: `vote_all` and `TPC` procedures.

9 Related Work

We review works concerning the use of commutativity reasoning in proving correctness of concurrent or distributed systems.

Commutativity reasoning in deductive verification. Lipton's reduction theory [35] introduced the concept of *movers* to define a program transformation that creates bounded-size atomic blocks. This work assumes a simple programming language without procedure calls and a fixed number of threads. QED [11] expanded the scope of Lipton's theory by introducing iterated application of reduction and abstraction over atomic actions. Also, atomic sections are allowed to contain loops but no procedure calls or dynamic thread creation. Civl [23] builds upon the foundation of QED, adding invariants [38,25], refinement layers and permission-based reasoning via a linear type system [28], and pending asyncs [30,27]. Pending asyncs can be viewed as threads restricted to executing a single atomic step and which cannot be joined. They are used to summarize asynchronous procedure calls and define a reduction scheme where asynchronous procedure calls are transformed to synchronous ones [30]. This reduction scheme is based on proving that the asynchronously called procedure can be summarized to a left-mover pending async. This idea has been extended to sequentializing an asynchronous program that creates an unbounded number of pending asyncs via an induction principle [27]. In this work, we introduce more flexible reduction schemes that improve scalability. These schemes support greater compositionality by allowing atomic sections to include both sequential and parallel procedure calls. Additionally, they expand the capabilities of reduction by enabling both left- and right-mover-based commutative reorderings.

Anchor [15] applies reduction to a low-level object-oriented language, where mover annotations are assigned to read and write accesses to object fields. It introduces a type system that enables proving the atomicity of entire procedures, which builds on Lipton's reduction. In contrast, our work is set in a more abstract language, supports compositional reduction reasoning about procedures, and accounts for parallel composition. [16] investigates the integration of reduction with rely-guarantee reasoning, which falls outside the scope of this work.

CSPEC [8] takes an approach similar to Civl but mechanizes all metatheory within the Rocq theorem prover [39] for flexibility and sound extensibility. Armada [36] also has flexible and mechanized metatheory whose usefulness is demonstrated by implementing a variety of program transformations, including those catering to fine-grained concurrency and weak memory models. IronFleet [22] embeds TLA-style state-machine modeling [33] into the Dafny verifier [34] to refine high-level distributed systems specifications into low-level executable implementations. Their proofs embed reduction reasoning into Dafny in a rather ad-hoc manner.

Movers have also been used to define an equivalence-preserving transformation that eliminates buffers in message-passing programs [6,40]. These works define a restricted class of programs and prove that reasoning about the set of *rendezvous* executions of these programs, where messages are delivered instantaneously, is complete, i.e., any other execution is equivalent to a rendezvous execution, up to reordering of mover actions. For instance, [40] introduces some number of heuristics which are based on syntax in order to reduce a given program. Those heuristics do not apply to our case studies, and it is hard to imagine an extension where they would become applicable. For instance, reduction is sometimes enabled by abstracting actions (message handlers) and this cannot be handled via syntactical arguments. Two-phase commit (2PC) is a canonical benchmark in this line of work and has been verified many times in a variety of systems, including automated ones[10,40]. In our 2PC, replicas use nontrivial logic to determine their votes, which is not the case for the versions used in these systems. In those previous works, replicas vote *Yes* or *No* nondeterministically, which significantly simplifies the correctness argument: all message handlers are left movers without requiring any abstraction [30]. In contrast, in our version of 2PC, some message handlers are right movers and some are left movers, after devising appropriate abstractions.

Commutativity reasoning in algorithmic verification. In the context of algorithmic verification, commutativity reasoning manifests in the so-called partial-order reduction techniques [21,18,2,26] which mostly concern finite-state systems or executions of bounded length.

In the context of automated proof synthesis for infinite-state programs, most existing work focuses on programs with a bounded number of threads [9,12,13]. The work in [14] proposes an instrumentation scheme for parameterized programs, where an unbounded number of threads execute the same code. This scheme enables the representation of sound reductions in such settings. Additionally, they formalize a notion of reduction usefulness, suggesting that a suitable reduction can lead to proofs requiring fewer or simpler ghost variables.

Acknowledgments. This work is partially partially supported by the French National Research Agency (project SCEPROOF).

10 Data Availability Statement

The implementation and all artifacts required to reproduce the results of this paper are publicly available. The source code of the Civl extension and the underlying Boogie infrastructure can be obtained from

https://github.com/boogie-org/boogie/tree/master/Source

The examples and benchmarks used in the evaluation are available in the Civl test suite at

https://github.com/boogie-org/boogie/tree/master/Test/civl

The experiments rely on the versions of Boogie and Z3 retrieved by the build scripts in the repository. These scripts contain the exact commands used to run the benchmarks and to produce the results reported in the paper. Reproduction can therefore be achieved by checking out the repository and following the documented build and evaluation instructions.

References

1. Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:10.1016/0304-3975(91)90224-P.
2. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014. doi:10.1145/2535838.2535845.
3. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.
4. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.
5. Hagit Attiya and Constantin Enea. Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs that Use Concurrent Objects. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: https://doi.org/10.4230/LIPIcs.DISC.2019.2, doi:10.4230/LIPICS.DISC.2019.2.
6. Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. Verifying distributed programs via canonical sequentialization. *Proc. ACM Program. Lang.*, 1(OOPSLA):110:1–110:27, 2017. doi:10.1145/3133934.
7. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. doi:10.1007/11804192_17.

8. Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 306–322. USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/chajed.
9. Duc-Hiep Chu and Joxan Jaffar. A Framework to Synergize Partial Order Reduction with State Interpolation. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2014. doi:10.1007/978-3-319-13338-6_14.
10. Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. *SIGPLAN Not.*, 51(1):400–415, January 2016. doi:10.1145/2914770.2837650.
11. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 2–15. ACM, 2009. doi:10.1145/1480881.1480885.
12. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Sound sequentialization for concurrent program verification. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 506–521. ACM, 2022. doi:10.1145/3519939.3523727.
13. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Stratified Commutativity in Verification Algorithms for Concurrent Programs. *Proc. ACM Program. Lang.*, 7(POPL):1426–1453, 2023. doi:10.1145/3571242.
14. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Commutativity Simplifies Proofs of Parameterized Programs. *Proc. ACM Program. Lang.*, 8(POPL):2485–2513, 2024. doi:10.1145/3632925.
15. Cormac Flanagan and Stephen N. Freund. The Anchor verifier for blocking and non-blocking concurrent software. *Proc. ACM Program. Lang.*, 4(OOPSLA):156:1–156:29, 2020. doi:10.1145/3428224.
16. Cormac Flanagan and Stephen N. Freund. Mover Logic: A Concurrent Program Logic for Reduction and Rely-Guarantee Reasoning. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*, volume 313 of *LIPIcs*, pages 16:1–16:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2024.16, doi:10.4230/LIPICS.ECOOP.2024.16.
17. Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting Purity for Atomicity. *IEEE Trans. Software Eng.*, 31(4):275–291, 2005. doi:10.1109/TSE.2005.47.
18. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005. doi:10.1145/1040305.1040315.

19. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 338–349. ACM, 2003. doi:10.1145/781131.781169.
20. Namratha Gangamreddypalli, Constantin Enea, and Shaz Qadeer. Reduction for Structured Concurrent Programs, 2026. URL: https://arxiv.org/abs/2601.13341, arXiv:2601.13341.
21. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. doi:10.1007/3-540-60761-7.
22. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015. doi:10.1145/2815400.2815428.
23. Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and Modular Refinement Reasoning for Concurrent Programs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 449–465. Springer, 2015. doi:10.1007/978-3-319-21668-3_26.
24. Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
25. Cliff B. Jones. Specification and Design of (Parallel) Programs. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 321–332. North-Holland/IFIP, 1983.
26. Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:10.1145/3498711.
27. Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 227–242. ACM, 2020. doi:10.1145/3385412.3385980.
28. Bernhard Kragl and Shaz Qadeer. Layered Concurrent Programs. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 79–102. Springer, 2018. doi:10.1007/978-3-319-96145-3_5.
29. Bernhard Kragl and Shaz Qadeer. The Civl verifier. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 143–152. IEEE, 2021. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_23, doi:10.34727/2021/ISBN.978-3-85448-046-4_23.

30. Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the Asynchronous. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.21, doi:10.4230/LIPICS.CONCUR.2018.21.
31. Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for Structured Concurrent Programs. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 275–298. Springer, 2020. doi:10.1007/978-3-030-53288-8_14.
32. Jeffrey Kuskin, David Ofelt, Mark A. Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy. The Stanford FLASH Multiprocessor. In David A. Patterson, editor, *Proceedings of the 21st Annual International Symposium on Computer Architecture. Chicago, IL, USA, April 1994*, pages 302–313. IEEE Computer Society, 1994. doi:10.1109/ISCA.1994.288140.
33. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: http://research.microsoft.com/users/lamport/tla/book.html.
34. K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.
35. Richard J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM*, 18(12):717–721, 1975. doi:10.1145/361227.361234.
36. Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 197–210. ACM, 2020. doi:10.1145/3385412.3385971.
37. Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.*, 121(2):214–233, 1995. URL: https://doi.org/10.1006/inco.1995.1134, doi:10.1006/INCO.1995.1134.
38. Susan S. Owicki and David Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commun. ACM*, 19(5):279–285, 1976. doi:10.1145/360051.360224.
39. The Coq Development Team. The Coq Proof Assistant, version 8.11.0, 2020. doi:10.5281/zenodo.3744225.
40. Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proc. ACM Program. Lang.*, 3(POPL):59:1–59:30, 2019. doi:10.1145/3290372.