Pretend Synchrony[*]

Synchronous Verification of Asynchronous Distributed Programs

KLAUS V. GLEISSENTHALL, University of California, San Diego, USA
RAMI GÖKHAN KICI, University of California, San Diego, USA
ALEXANDER BAKST, University of California, San Diego, USA
DEIAN STEFAN, University of California, San Diego, USA
RANJIT JHALA, University of California, San Diego, USA

We present *pretend synchrony*, a new approach to verifying distributed systems, based on the observation that while distributed programs must execute asynchronously, we can often soundly treat them as if they were synchronous when verifying their correctness. To do so, we compute a *synchronization*, a semantically equivalent program where all sends, receives, and message buffers, have been replaced by simple assignments, yielding a program that can be verified using Floyd-Hoare style Verification Conditions and SMT. We implement our approach as a framework for writing verified distributed programs in Go and evaluate it with four challenging case studies— the classic two-phase commit protocol, the Raft leader election protocol, single-decree Paxos protocol, and a Multi-Paxos based distributed key-value store. We find that pretend synchrony allows us to develop performant systems while making verification of functional correctness *simpler* by reducing manually specified invariants by a factor of 6, and *faster*, by reducing checking time by three orders of magnitude.

CCS Concepts: • **Theory of computation** → **Program verification**; **Program analysis**; **Distributed computing models**; *Concurrency*; • **Software and its engineering** → Software notations and tools;

1 INTRODUCTION

Asynchronous distributed systems are viciously hard to get right. Network delays, variations in execution time and message loss may trigger behaviours that were neither intended nor anticipated by the programmer. To eliminate this subtle source of errors, programmers painstakingly construct workloads and stress tests to tickle the relevant schedules [Desai et al. 2015; Killian et al. 2007; Yang et al. 2009]. These efforts are doomed to come up short. Due to unbounded data domains and the fact that programs are often parameterized, *i.e.*, the number of participating nodes is not known at compile time, the number of possible schedules is infinite, ensuring that tests, at best, scratch the surface of all possible behaviors. A more principled approach to ensuring correctness is to formally verify their intended properties through mathematical proof. However, verification is easier said than done. Due to the *asynchronous* nature of communication, the programmer must supply the prover with invariants that explicitly enumerate possible schedules by splitting cases over the joint states of all participants. Such invariants are complicated and hard to divine. Worse, asynchronous invariants must reason about the contents of (unbounded) message buffers which renders even *checking* the correctness of candidate invariants undecidable.

Lipton's reduction method [Lipton 1975] offers a tantalizing path towards taming asynchrony. Intuitively, it provides a basis for *moving* every receive up to its matching send operation thereby

[*] This work was supported by NSF grants CCF-1422471, CCF-1223850, CNS-1514435, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and generous gifts from Microsoft Research and Cisco.

Authors' addresses: Klaus v. Gleissenthall, Computer Science and Engineering, University of California, San Diego, USA, gleissen@cs.ucsd.edu; Rami Gökhan Kıcı, Computer Science and Engineering, University of California, San Diego, USA; Alexander Bakst, Computer Science and Engineering, University of California, San Diego, USA; Deian Stefan, Computer Science and Engineering, University of California, San Diego, USA, deian@cs.ucsd.edu; Ranjit Jhala, Computer Science and Engineering, University of California, San Diego, USA, jhala@cs.ucsd.edu.

fusing the pair of asynchronous operations into a single synchronous assignment. Several authors have explored how this idea can be used to simplify reasoning *e.g.*, about atomicity to permit correctness proofs of shared memory programs [Flanagan and Qadeer 2003; Hawblitzel et al. 2015b], and to reason about deadlocks in message passing systems [Bakst et al. 2017]. Unfortunately, several challenging problems conspire to thwart the application of reduction – specifically, fusing sends and receives – in the distributed setting.

(1) ***Broadcasts:*** We must account for sets of processes broadcasting to and hence receiving from other sets of processes. For example, systems like Paxos [Lamport 2001] or Raft [Ongaro and Ousterhout 2014] have sets of proposers broadcasting to sets of acceptors. These broadcasts defeat reduction: how do we fuse sets of sends to sets of receives when the actual order depends on different interleavings at run-time?
(2) ***Drops:*** We must account for messages being dropped on the network, or equivalently, delayed for arbitrarily long periods of time. These drops prevents reduction: how do we fuse a receive with a send whose payload may not even make it across the network?
(3) ***Rounds:*** Finally, distributed executions are often conceptually structured into iterations of multiple rounds of execution. For example, we might have multiple rounds (or ballots) in a leader election protocol, or different rounds of communication triggered by subsequent key-lookups in a distributed store. These rounds stymie reduction: how do we fuse sends and receives only within a single round even though delayed messages may actually be received and interfere with future rounds?

Our key insight is that by careful language design we can ensure that idiomatic and efficient distributed programs are imbued with enough semantic structure to enable reduction, thereby simplifying formal verification. In particular,

(1) *Symmetric process identifiers* suffice to name the individual processes that comprise the sets participating in broadcasts. We can make symmetry explicit by using the scalarset datatype [Norris IP and Dill 1996] which only allows equality comparison between process identifiers. Crucially, even though broadcasts may be received in different orders, symmetry ensures that the orders are equivalent modulo a permutation of process identifiers thereby allowing us to reduce multi-process broadcasts via a standard focus-and-blur (or instantiate-and-generalize) operation from the theory of abstract interpretation [Sagiv et al. 1999].
(2) *Typed channels* as in Scala actors or Go, help ensure that at any moment, a process is expecting messages of a single type, and that at most one message of a given type in-flight between a pair of processes. We observe that this property lets us treat message drops simply as a receive-with-timeout operation that returns a `None` message. This directly lets us soundly reduce sends and receives to assignments of `Maybe` messages.
(3) *Round identifiers* allow us to make the notion of rounds explicit in the program syntax, specifically to structure repeated computations as iterations over an unbounded set of `Rounds`. We can then use this structure to define a notion of independence called *round non-interference* which, inspired by classic work on loop parallelization [Bernstein 1966], intuitively ensures that the different rounds are communication-closed [Tzilla Elrad 1982]. This allows us to exploit the symmetry over `Rounds` to simplify the problem of proving an assertion over *all* rounds, into proving the assertion over a single, arbitrarily chosen round, which can be done using Lipton's reduction.

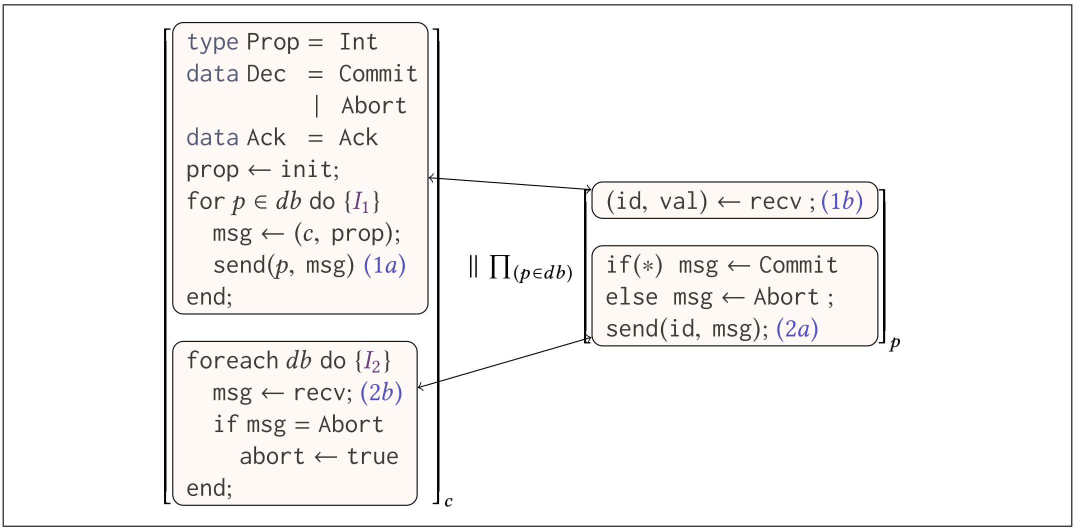
In this paper we use the above insights to develop *pretend synchrony*, an algorithmic framework for implementing formally verified distributed systems. Our framework is realized via the following novel contributions.

- ***Library:*** Our first contribution is a *library* of types and communication primitives that the programmer can use to implement an asynchronous, distributed system. To specify correctness requirements, the programmer also annotates loops iterating over (unbounded) sets of processes with synchronous assertions. Our library's design ensures that communication is structured into typed channels that enable reduction even in the presence of message drops, broadcasts, and multiple rounds of interaction – essential features of real-world distributed systems (§ 3).
- ***Compiler:*** Our second contribution is an algorithm that compiles programs into their synchronizations. Following [Bakst et al. 2017], we structure this compiler as a set of local rewrite rules each of which transforms the input program into a new program comprising a synchronized prefix and a suffix that still needs to be rewritten. Crucially, we introduce novel rewrite rules that exploit the communication structure explicated by our library to enable syntax-directed reduction in the presence of message drops, broadcasts and multiple rounds of communication. We identify a class of programs called Stratified Pairwise Communication Protocols (SPCP) for which the compiler is *complete*, *i.e.*, the compiler is guaranteed to compute a synchronization, if one exists (§ 4).
- ***Verifier:*** Our third contribution is to show how to use synchronization as a basis for verification. We prove a *soundness* result that each rewrite preserves the halting states of the program (Theorem 4.4). We then implement a *verifier* that traverses the synchronized program in a syntax-directed fashion to compose the code and assertions to emit Owickie-Gries style [Owicki and Gries 1976] verification conditions whose validity – determined via SMT – together with the soundness result, implies the correctness of the original source (§ 6).
- ***Evaluation:*** Our final contribution is an implementation of pretend synchrony as GOOLONG: a framework for writing verified distributed programs in GO. We use GOOLONG to develop four case studies: the classic two-phase commit protocol, the Raft leader election protocol, single-decree Paxos protocol, and a Multi-Paxos based distributed key-value store. To demonstrate that our library allows idiomatic implementations, we build a key-value store atop Multi-Paxos and show its performance to be competitive with other implementations. To demonstrate that our synchronizing compiler facilitates correctness verification, we show that all these protocols have simple and intuitive synchronous invariants that GOOLONG checks automatically. To demonstrate that pretend synchrony simplifies verification, we also implement the all case studies, except for the key-value store, in DAFNY [Leino 2010], and verify them using the classical approach as used by the state-of-the-art IRONFLEET system [Hawblitzel et al. 2015a]. We find that pretend synchrony reduces the number of manually specified invariant annotations by a factor of 6. In our experience, the synchronous assertions verified by GOOLONG proved essential in determining the many asynchronous invariants required for verification by DAFNY. Moreover, by eschewing complex quantified invariants (over the contents of message buffers), pretend synchrony shrinks the time taken to check the programs by three orders of magnitude – from over twenty minutes with DAFNY to under two seconds (§ 7).

2 OVERVIEW

We begin by motivating pretend synchrony at a high-level, and then present a series of small examples that illustrate its essential ingredients.

Network and Failure Model Our formal development and benchmarks assume a network where messages may be arbitrarily dropped or re-ordered. We model such messages via non-blocking receives that can non-deterministically time out and return a `None` value. Further, we assume a



```
[ type Prop = Int
  data Dec  = Commit
            | Abort
  data Ack  = Ack
  prop ← init;
  for $p \in db$ do {$I_1$}
    msg ← ($c$, prop);
    send($p$, msg) ($1a$)
  end;

  foreach $db$ do {$I_2$}
    msg ← recv; ($2b$)
    if msg = Abort
      abort ← true
  end; ]_c

|| $\prod_{(p \in db)}$

[ (id, val) ← recv ; ($1b$)

  if(∗) msg ← Commit
  else msg ← Abort ;
  send(id, msg); ($2a$) ]_p
```

Fig. 1. Two phase commit: first round. Each for loop is annotated with an invariant {I}. We use _ to denote an irrelevant variable.

standard crash-recovery failure model [Lamport 2001] wherein a process can crash, but where the process must save its state in persistent storage, and resume execution from that persisted state, or stay silent. Silence is equivalent to all the processes messages being dropped by the network. However, to simplify presentation in this overview, we will assume that all receives are blocking, and that messages cannot be dropped, but may be arbitrarily re-ordered.

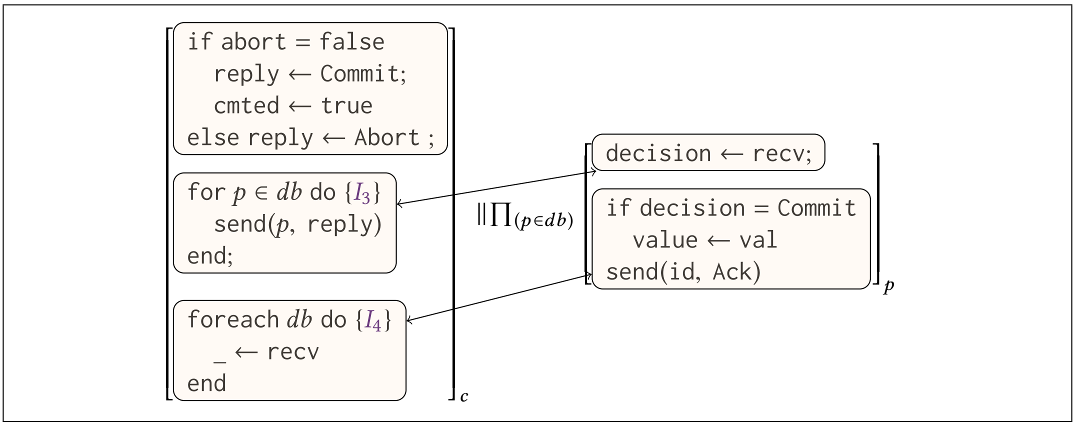
2.1 Motivation: Pretend Synchrony

Two-Phase Commit Figures 1 and 2 show the classic two phase commit (2PC) protocol [Lampson and Sturgis 1976]. In this protocol, a coordinator node c tries to commit a value to a number of database nodes db. We use $[P]_p$ to denote a single process p executing a program P and $\prod_{(p \in ps)} [P]_p$ to say that P is executed by a set of processes ps. We use $\|$ to denote parallel composition and assume that, initially, both `cmted` and `abort` are set to `false`. The protocol is made up of two rounds. In the first round, shown in Figure 1, the coordinator loops over all nodes to send them its proposal value. Each database node then nondeterministically chooses to either commit or abort the transaction and sends its choice to the coordinator. If at least one of the nodes chose to abort, the coordinator aborts the entire transaction by setting the appropriate flag. In the protocol's second round, the coordinator broadcasts its decision to either commit or abort to the database nodes which reply with an acknowledgement. Finally, if the coordinator decided to commit the transaction, each database node sets its value to the previously received proposal.

Correctness To prove correctness of our implementation of 2PC, we want to show that, if the protocol finished and the coordinator decided to commit the transaction, all database nodes have indeed chosen the proposed value, *i.e.*, the following must hold *after* the protocol:

$$\forall p \in db \ : \ c.\texttt{cmted} = \texttt{true} \Rightarrow p.\texttt{value} = c.\texttt{prop}$$

Asynchronous Invariants are Complicated Let us first consider how to prove this property in an *asynchronous* setting. We follow the proof from [Sergey et al. 2018]. Consider the coordinator's first loop in Figure 1 and let *done* denote the set of all database nodes for which the send at location



```
if abort = false
  reply ← Commit;
  cmted ← true
else reply ← Abort ;

for p ∈ db do {I_3}
  send(p, reply)
end;

foreach db do {I_4}
  _ ← recv
end
```

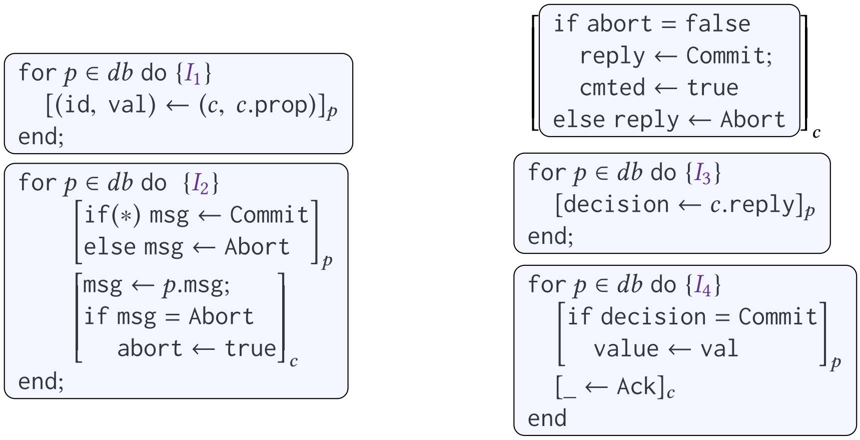
```
decision ← recv;

if decision = Commit
  value ← val
send(id, Ack)
```

Fig. 2. Two phase commit: second round.

(1a) has been executed, so far. In order to rule out messages appearing "out of thin air", we need to assert that whenever $p \notin done$, then there are no messages from c to p, and p has not yet executed the corresponding receive at location (1b). If, $p \in done$, we need to case split over the location of p due to the asynchronous nature of communication. Either, *1)* there is an in-flight message from c to p that contains c's process id and proposal value and p is waiting to receive the message, or *2)* p received c's message, set its `val` variable to `prop` and decided to either commit or abort the transaction, but did not respond yet, or *3)* p responded, relaying its decision to c. We need a similar case split for c's second loop in Figure 1: if $p \notin done$ then either *1)* there is a pending message from c to p containing c's process id and proposal, or *2)* p has chosen to commit or abort but has not yet sent a response, or *3)* p has sent its response consisting of either a commit or abort message. Finally, if $p \in done$, then p must have finished the first part of the protocol and `val` must be set to c's proposal. The invariant for the second part of the protocol consists of a similar case split.

Asynchrony Makes Verification Undecidable While avoiding such case splits — over the joint state of the coordinator, the database nodes and the message buffer— would be desirable in of itself, there is a more fundamental issue with proving correctness in an asynchronous setting: directly including the message buffer into the system state by modeling it as an array requires *nested array reads* which makes even *checking* a candidate invariant undecidable [Bradley et al. 2006]

Pretend Synchrony In this paper, we identify a new approach which builds on the following observation: even though the program behaves asynchronously when *executed*, we can soundly pretend that communication is synchronous when *reasoning* about the program. Consider, for example, the send marked with (1a) in Figure 1 and assume that in the current iteration coordinator c sends to some process p. Since in our execution model, messages are indexed by the *type* of values that are being sent across the network, trivially, the message can only be received at a *single receive*, namely the receive of p at the location marked with (1b) in Figure 1 (the receive in Figure 2 is expecting a value of type `Decision`). Moreover, the send is *non-interfering* with respect to all sends between c and *other* processes in db. Thus, without affecting the validity of our correctness property, we can, via Lipton's theory of movers [Lipton 1975], *pretend* the proposal is received *directly after* it is sent. A similar argument can be made for p send at location (2a) in Figure 1. Even though p's send matches multiple receives (*i.e.*, any receive at location (2b) in p's loop), the states resulting from picking a particular winner for a given iteration are *symmetric i.e.*, are equivalent modulo a *permutation* of process identifiers [Norris IP and Dill 1996]. Thus, as long as we preserve all possible iteration orders, we can match up the sends and receives without affecting correctness.



```
for $p \in db$ do $\{I_1\}$
    $[$(id, val) ← ($c$, $c$.prop)$]_p$
end;

for $p \in db$ do $\{I_2\}$
    $[$if(*) msg ← Commit
     else msg ← Abort$]_p$
    $[$msg ← $p$.msg;
     if msg = Abort
         abort ← true$]_c$
end;
```

(a) **first phase**

```
$[[$if abort = false
      reply ← Commit;
      cmted ← true
  else reply ← Abort$]]_c$

for $p \in db$ do $\{I_3\}$
    $[$decision ← $c$.reply$]_p$
end;

for $p \in db$ do $\{I_4\}$
    $[$if decision = Commit
         value ← val$]_p$
    $[$_ ← Ack$]_c$
end
```

(b) **second phase**

Fig. 3. Synchronization of two phase commit.

Synchronous Invariants are Simple Our method exploits this insight in the following way: instead of writing invariants for the asynchronous program, the user writes invariants *as if the program were synchronous*, effectively treating matching send and receive pairs as assignments. The synchronous invariants are given as annotations to for loops. Consider again Figure 1. For the first loop, we need the following invariant I_1 stating that, if the loop was executed for some process p (indicated by $p \in done$, where $done$ is an auxiliary variable referring to the set of p's that already executed the loop), then p must have been assigned c's proposal value.

$$I_1 \triangleq \lambda\, done.\ \forall p.\ p \in done \Rightarrow p.\texttt{val} = c.\texttt{prop}$$

Invariants I_2 and I_3 are trivial (*i.e.*, the same as I_1), as the respective loops do not modify any relevant values, and need not be supplied by the user. Finally, invariant I_4 states that whenever the loop has been executed for a process p, process p's value variable must be set to the proposal it received in the first round.

$$I_4 \triangleq \lambda\, done.\ \forall p. \begin{pmatrix} p \in done\ \land \\ c.\texttt{cmted} = \texttt{true} \end{pmatrix} \Rightarrow p.\texttt{value} = p.\texttt{val}$$

Together, these simple synchronous invariants – free of the case-splits needed to account for asynchrony – are sufficient to prove the correctness property for the 2PC.

Checking the Synchronous Invariant We verify the correctness of the synchronous invariant in two steps. First, we compute the *synchronization*; a semantically equivalent synchronous program, by iteratively applying a set of rewriting rules (§ 4). Second, we use the synchronization to generate and check verification conditions that ensure that the supplied invariants are inductive, interference-free and imply the desired correctness property (§ 6). In the first step, our implementation computes this synchronization completely automatically without relying on the user supplied invariants. Each rewriting step produces a new program that comprises a *synchronized prefix* and a suffix that still needs to be rewritten. This process is repeated until the suffix is reduced to skip. Figure 3 shows the synchronization that GOOLONG computes for 2PC. Intuitively, the synchronous version matches up the connected blocks from Figures 1 and 2. In the synchronized version, all protocol steps are executed one after the other: first, the coordinator assigns its proposal to all database nodes; then, each node decides to either commit or abort; then, the coordinator assigns its decision to each of the database nodes, and, finally, each node assigns the proposed value in case the coordinator

$$\left[\boxed{\begin{array}{l} \texttt{for } q \in qs \texttt{ do } \{I_5\} \\ \quad \texttt{send}(q,\ \texttt{ping}); \\ \quad \_ \leftarrow \texttt{recv}(q) \\ \texttt{end}\end{array}}\right]_p \parallel \prod_{(q\in qs)} \left[\boxed{\begin{array}{l} \texttt{v} \leftarrow \texttt{recv}; \\ \texttt{send}(p,\ \texttt{Ack})\end{array}}\right]_q$$

(a) **asynchronous**

$$\boxed{\texttt{for } q \in qs \texttt{ do } \{I_5\} \quad [\texttt{v} \leftarrow \texttt{ping}]_q \ ; \quad [\_ \leftarrow \texttt{Ack}]_p \texttt{ end}}$$

(b) **synchronous**

Fig. 4. Ex1 ping broadcast: reduction.

decided to commit. In the second step, Goolong uses the synchronization and the user supplied invariants to compute verification conditions that ensure that the program satisfies all assertions. Goolong checks the verification conditions by discharging them to an SMT-solver. Importantly, the verification conditions for checking the invariants in the synchronous setting fall into the array property fragment [Bradley et al. 2006] and hence, checking the invariant is decidable.

2.2 Main Ideas

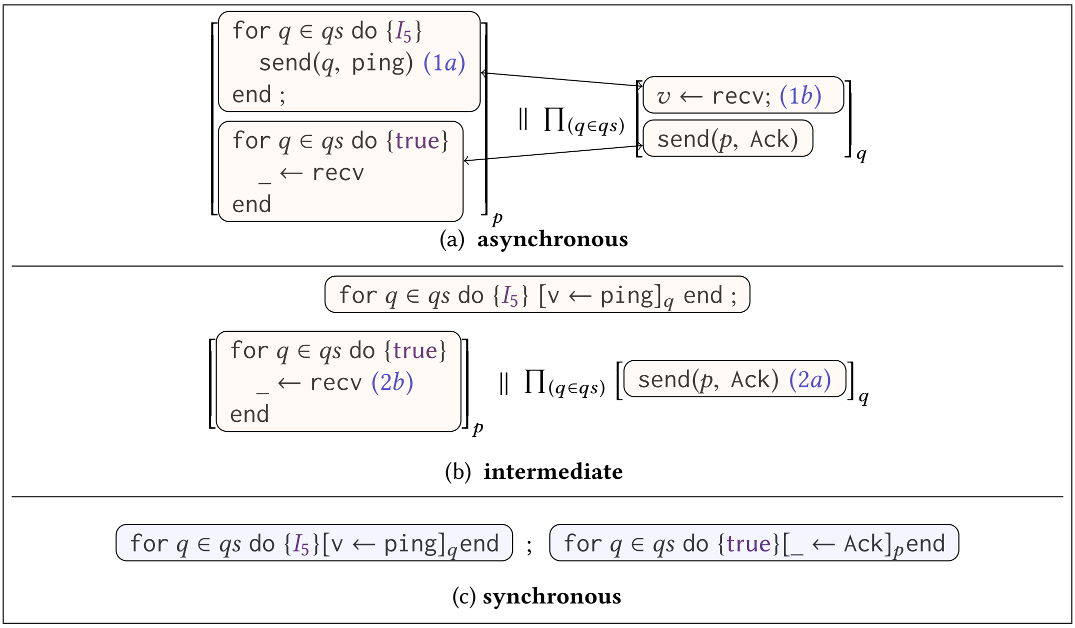
We now discuss the main ingredients behind pretend synchrony and illustrate how they enable synchronous verification of asynchronous distributed programs.

Ex1: Reduction Reasoning about programs as if they were synchronous greatly simplifies verification, but when can we synchronize a program without affecting the validity of the correctness property we want to verify? If we restrict ourselves to proving properties about halting states, then, for a given trace, we can always soundly synchronize a send by moving it up to its matching receive, via Lipton's theory of reduction [Lipton 1975]. Consider example Ex1 shown in Figure 4a. Process p loops over a statically unbounded set of processes qs. For each process q in qs, p sends a ping message and waits for q's acknowledgement. Each process in qs waits to receive a value, assigns the value to v, and finally answers with an acknowledgement. We want to prove that, after terminating, all processes in qs have set their variable v to ping, *i.e.*, we want $\forall q \in qs : q.\texttt{v} = \texttt{ping}$ to hold upon termination. For this, we first note that, for each $q \in qs$, q's receive can only be matched by a *single* send (the one in p). Moreover, the property we want to prove does not refer to intermediate program states such as the contents of the message buffer. Therefore, we can soundly rewrite the program into a simpler, synchronous version by transforming the send and receive pair into an assignment. Performing a similar step for q's send and p's receive yields the synchronous program shown in Figure 4. Then, the following invariant proves the intended property. This invariant states that, whenever the loop has iterated over a process, its v variable is set to ping.

$$I_5 \triangleq \lambda\, done.\ \forall\, q.\ q \in done \Rightarrow q.\texttt{v} = \texttt{ping}$$

Note that if we were to verify the original asynchronous program, this simple invariant would not be enough; instead, we would have to both case split on whether messages have yet been received or not and maintain an invariant about the messages buffer (*i.e.*, all messages from p to processes in qs contain the value ping).

Ex2: Symmetry We can move a send up to its matching receive, not only if a unique matching receive exists across all program traces, but also if all matching receives are *symmetric*, i.e., the states that result from picking a particular receive are equivalent up to a permutation of process



```
[ for q ∈ qs do {I_5}
    send(q, ping) (1a)
  end ;
  for q ∈ qs do {true}
    _ ← recv
  end ]_p   ∥ ∏(q∈qs) [ v ← recv; (1b)
                        send(p, Ack) ]_q
```

(a) **asynchronous**

```
for q ∈ qs do {I_5} [v ← ping]_q end ;
[ for q ∈ qs do {true}
    _ ← recv (2b)
  end ]_p   ∥ ∏(q∈qs) [ send(p, Ack) (2a) ]_q
```

(b) **intermediate**

```
for q ∈ qs do {I_5}[v ← ping]_q end ; for q ∈ qs do {true}[_ ← Ack]_p end
```

(c) **synchronous**

Fig. 5. Ex2 split ping broadcast: symmetry.

ids [Norris IP and Dill 1996]. Consider Ex2 in Figure 5a. Process p, first sends ping messages to all processes in q, and then, in a second loop, waits for their acknowledgements. As in the previous example, we want to show that, on termination, all processes in qs have set their v variable to ping. For this, consider the first loop of process p. As in the previous examples, p's send to some process q at location (1a) can only be received at a single receive, namely the receive of process q at location (1b). Hence, we can rewrite the original, asynchronous program into an intermediate program in Figure 5b. This intermediate program consists of a synchronous prefix corresponding to a synchronization of the first loop and a to-be-synchronized remainder.

Next, consider p's second loop. For a given loop iteration, p's receive at location (2b) can receive from multiple sends from processes in qs at location (2a). However, all the sends are *symmetric*. That is, the processes run the same code but differ in their process ids, and hence, picking an arbitrary winner for each iteration results in the same final states. This yields the synchronized program shown in Figure 5c. Even though Ex2 structures communication differently than Ex1, the synchronous correctness proof is unaffected. Consequently, we can reuse the same invariant, *i.e.*, I_5 to prove the desired correctness property.

Ex3: Multi-Cast Unfortunately, reduction and symmetry by themselves are not sufficient to verify realistic distributed systems. Consider Ex3 in Figure 6a that is a simplification of the "proposing" phase of the Paxos protocol [Lamport 2001]. Here, a set of processes ps is communicating with a set of processes qs. Each process p in ps loops over all processes in qs. For each process q in qs, p sends a pair consisting of its own process identifier and the value ping. Similarly, each process q in qs loops over all processes in ps. In each iteration, q receives an id and a value v from some process in ps and then sends an acknowledgment to the id. As in the previous examples, we would like to prove that, after the program has executed, each process in qs has set its local variable v to the value ping. Unfortunately, we cannot directly apply the reasoning from the last examples as the races in Ex3 are *not symmetric*! Consider, for instance, the receive of some process q in qs at location (b). For a given loop iteration, process q can receive from all processes in ps at location (a), however, these

```
$\prod_{(p \in ps)}$ [[ for $q \in qs$ do $\{I_5\}$
                          msg ← ($p$, ping);
                          send($q$, msg); $(a)$
                          _ ← recv($q$)
                        end ]]$_p$
|| $\prod_{(q \in qs)}$ [[ foreach $ps$ do
                             (id, v) ← recv; $(b)$
                             send(id, Ack)
                           end ]]$_q$
```

(a) **asynchronous**

```
[[ for $q \in qs$ do $\{I_5\}$
     send($q$, ($u$, ping));
     _ ← recv($q$)
   end ]]$_u$
|| $\prod_{(q \in qs)}$ [[ (id, v) ← recv;
                           send(id, Ack) ]]$_q$
```

(b) **intermediate synchronization**

```
$\prod_{(p \in ps)}$ [ for $q \in qs$ do $\{I_5\}$ ⟨[(id, v) ← ($p$, ping)]$_q$; [_ ← Ack]$_p$⟩ end ]
```

(c) **synchronous**

Fig. 6. Ex3 ping multi-cast.

processes may be in different loop iteration and hence, in general, picking one over the other might lead to different outcomes. To overcome this predicament, we focus our attention on the interaction between a single process in ps and an arbitrary iteration of the processes in qs. Even though the overall system has non-symmetric races, each interaction between a single process in p and all process in qs is symmetric. We therefore say that Ex3's races are *almost symmetric*, and exploit this insight in the following way: in order to compute the synchronization of the overall system, it is enough to synchronize the interaction between a single process in ps and all processes qs and to then repeat the single interaction in parallel, once for each process in ps. Importantly, in contrast to earlier examples, the synchronization is no longer a sequential program, but rather a parallel composition over processes ps. Figure 6b shows the interaction between a single process $u \in ps$ and an arbitrary iteration of each process in qs. Since this interaction is the same as in Ex2, we obtain the overall synchronization shown in Figure 6c, where we use $\langle P \rangle$ to mean that P is executed atomically. Finally, we note that, in spite of the substantially more complicated communication pattern, the intended property can still be proved through the simple synchronous invariant I_5.

Ex4: Message Drops A crucial difficulty in verifying distributed systems is the fact that messages can be dropped by the network. In such a setting, receives must be able to time out in order avoid waiting indefinitely for lost messages. Since it is, however, impossible to tell whether a message has in fact be dropped, or just delayed, this creates the possibility of receiving *stale messages* (*i.e.*, those meant for an earlier receive that timed out prematurely). As a result, a correctness proof in an asynchronous setting must show that stale messages do not affect the desired properties. Pretend synchrony's language restrictions enforce the following communication invariant that makes reasoning about message drops easy: at any given moment, a process expects messages of a single type, and at most one message of a given type is in flight between a given process pair, *i.e.*, there is only a single matching message from each possible sender. Given this restriction, we can synchronize a receive with timeout that expects a message from some fixed other process, without fear of receiving a stale message on the same channel, however, instead of assigning the received value straight away, we case split over whether the message has been dropped. We can generalize this reasoning to the case where we expect messages from a set of processes by invoking symmetry

```
⟦ for q ∈ qs do          ⟧
⟦   send(q, ping)        ⟧     ∥ ∏(q∈qs) ⟦ v ← recvTO ⟧q
⟦ end                    ⟧p
```

(a) **asynchronous**

```
for q ∈ qs do
  ⎡ if ∗ then      ⎤
  ⎢   v ← Just Ack ⎥
  ⎢ else           ⎥
  ⎣   v ← None     ⎦q
end
```

(b) **synchronous**

Fig. 7. Ex4 ping timeout.

reasoning, as before. Consider example Ex4 in Fig. 7, in which process p sends a ping message to each process in set qs. Each process in qs waits for a message using a non-blocking receive. This receive either assigns the received value wrapped in a maybe-type Just to variable v, or nondeterministically times out, in which case it assigns None. Figure 7b shows the synchronization of Ex4: The synchronization loops over all processes in qs and either assigns ping to q's local variable v, or p nondeterministically assigns Just Ack or None to w.

Ex5: Multi-Round Next, we discuss how to repeat protocols using rounds. Consider Ex5 in Figure 8 which repeats the protocol from Ex1 from Figure 4a, using rounds. For this, process p uses a loop rounds $r \in R$ do P end that iterates P once for each round identifier r in a set R. Each process in q repeats its protocol using a repeat statement, which loops forever. In each round, process p iterates over the processes in qs and sends a pair consisting of round number r and a ping to each q. It then waits for a reply from the same process, where it only accepts values in the current round r. Each process in qs waits for a value and a round number. When it receives a message, it binds the round number to r, assigns the received value to a map v that is indexed by the current round, and finally responds with an acknowledgement for round r. As in 2.2, we want to prove that variable v is set to ping upon termination of the protocol, however, now we want to show that this property holds for all rounds. More formally, we want to show that $\forall r \in R : \forall q \in qs : v[r] = \mathtt{ping}$ holds after the protocol finished. In order to prove this property, we first observe that Ex5 exhibits what we call *round non-interference*. Round non-interference requires that *1)* iterations corresponding to different rounds do not interfere with each other by sharing state, *i.e.*, by writing to or reading from the same variable and *2)* iterations corresponding to different rounds do not interfere by sending messages. Example Ex5 satisfies both of these conditions: iterations corresponding to different rounds do not share state since each process only accesses v indexed by its current round number r. Similarly, in Ex5 iterations corresponding to different rounds do not send messages to each other since p and all qs only send and receive messages for currently bound round number r. Since Ex5 exhibits round non-interference, we can prove that $\forall r \in R : \forall q \in qs : v[r] = \mathtt{ping}$ holds for all iteration, by showing that the property holds for a single, arbitrary iteration. (Theorem 5.4 in Section 5). Figure 8b show the instantiation of Ex5 to some arbitrary round r^*, which lets us to reuse invariant I_5 to prove the desired property.

Expressiveness To compute a synchronization, Goolong requires the input program only have almost symmetric races. Goolong automatically verifies this condition with a syntactic check (§ 3). When this check fails, Goolong produces an error witness consisting of the sends and receives participating in the race. We find that numerous protocols either naturally only contain almost symmetric races or can easily be refactored to ensure they do (most often this refactoring consists of making sure that messages that are independent of each other have different types). When a synchronization cannot be computed, Goolong provides feedback in the form of a counterexample

```
[ rounds r ∈ R do                     ]        [ repeat do                ]
[   for q ∈ qs do                     ]   ||   [   (r, v[r]) ← recv;      ]
[     send(q, (r, ping));             ] ∏(q∈qs)[   send(q, Ack, r)        ]
[     _ ← recv(q , r)                 ]        [ end                      ]_q
[   end                               ]
[ end                                 ]_p
```

(a) **multi-round**

```
[ for q ∈ qs do                       ]        [ (r*, v[r*]) ← recv;      ]
[   send(q, (r*, ping));              ]   ||   [ send(q, r*, Ack)         ]_q
[   _ ← recv(q, r*)                   ] ∏(q∈qs)
[ end                                 ]_p
```

(b) **arbitrary round**

Fig. 8. Ex5: multi-round version of Ex1.

to synchronization, which consists of a synchronized prefix and a remainder program that cannot be synchronized. We find that the counterexample often helps identify and eliminate the impediment to synchronization. In order for Goolong to compute a synchronization, the program must be *synchronizable*, that is, it must neither contain deadlocks nor spurious sends. We identify a class of programs called Stratified Pairwise Communication Protocols (SPCP) for which our method is *complete*, *i.e.*, it computes a synchronization if and only if the program is synchronizable (§ 4.5). Intuitively, this fragment consists of programs in which processes communicate pairwise, one after the other, by iterating over sets of processes. This fragment contains all the programs presented so far, *i.e.*, examples Ex1 to Ex4 and two-phase commit, and three out of our four case studies.

3 DISTRIBUTED MESSAGE PASSING PROGRAMS

Figure 9 shows the syntax of IceT, an imperative language for distributed message-passing programs. IceT serves as an intermediate language that Goolong automatically extracts from Go programs that were written using our library.

3.1 Syntax

Distributed Processes Each distributed process is associated with a unique identifier (ID), which serves as an address for sending messages. Identifiers are of the scalarset datatype [Norris IP and Dill 1996], *i.e.*, can only be compared for equality. We use `skip` to denote the empty process and use $[s]_p$ to denote a single process with ID p executing statement s. We write $p.x$ to refer to variable x of process p; when unambiguous, we will omit p and just write x.

Programs We obtain programs from the sequential and parallel composition of single process statements. We allow grouping of consecutive statements of the same process, *i.e.*, for statements s_1 and s_2, we abbreviate $[s_1]_p; [s_2]_p$ to $[s_1; s_2]_p$. Let X be a finite set. Then, we write $\prod (x \in X).P$ and $\texttt{for } x \in X \texttt{ do } P$ to denote the *parallel* and *sequential* composition of all instantiations of P to values in X, respectively. More formally, let $X = \{x_0, x_1, \ldots, x_k\}$ and let $t[u/x]$ denote the substitution (without capture) of term u for variable x in term t. Then, we define

$$\prod (x \in X).P \triangleq P[x_0/x] \;\|\; P[x_1/x] \;\|\; \ldots \;\|\; P[x_k/x]$$
$$\texttt{for } x \in X \texttt{ do } P \texttt{ end} \triangleq P[x_0/x] \;;\; P[x_1/x] \;;\; \ldots \;;\; P[x_k/x]$$

$x, r, X, R \in$ Identifiers $\quad t \in$ MsgType

e ::=		**Expressions**
	c	*literal values*
	| $p.x$	*variable*
	| $f(\bar{e})$	*primitive*
s ::=		**Statements**
	$\mathtt{x} \leftarrow \mathtt{e}$	*assignment*
	| $\mathtt{send}(t, e, e, r)$	*send*
	| $\mathtt{x} \leftarrow \mathtt{recv}(\mathtt{w}, \mathtt{t}, \mathtt{r})$	*receive*
	| $\mathtt{x} \leftarrow \mathtt{recvTO}(\mathtt{w}, \mathtt{t}, \mathtt{r})$	*receive with timeout*
	| $\mathtt{assume}(e)$	*assume*
	| $\mathtt{assert}(e)$	*assert*
	| $\mathtt{x} \leftarrow \mathtt{pick}(\mathtt{X})$	*pick element*
w ::=		**Sender**
	$*$	*any sender*
	| e	*expression*
P ::=		**Program**
	$\mathtt{skip}$	*empty process*
	| $[s]_e$	*singleton process*
	| $P; P$	*sequential composition*
	| $P \parallel P$	*parallel composition*
	| $P \oplus P$	*non-deterministic choice*
	| $\prod (x \in X).P$	*parallel iteration*
	| $\mathtt{for}\ x \in X\ \{e\}\ \mathtt{do}\ P\ \mathtt{end}$	*sequential iteration with binder*
	| $\mathtt{foreach}\ X\ \mathtt{do}\ P\ \mathtt{end}$	*sequential iteration without binder*
	| $\mathtt{rounds}\ r \in R\ \mathtt{do}\ P\ \mathtt{end}$	*sequential iteration over protocol rounds*
	| $\mathtt{repeat}\ \mathtt{do}\ P\ \mathtt{end}$	*unbounded iteration over protocol rounds*
	| $\langle e \triangleright P \rangle$	*atomic action*

Fig. 9. Syntax of the IceT language.

We use foreach X do P to denote sequential iteration over a finite set X *without* binding of elements in X, *i.e.*, for a set X with k elements foreach X do P repeats program P k-times. Note that foreach-loops are the only way a process can create new process identifiers. We use rounds $r \in R$ do P end to denote iteration of a program P over a set of protocol rounds R, and repeat do P end to denote a program that repeats P indefinitely.

Normal Forms A program is in *normal form* if it consists of a parallel composition of sequences of statements from *distinct processes* that do not share variables, *i.e.*, in a normal form program, a process cannot be composed in parallel with itself. For two such programs P and Q, we let $P \circ Q$ denote the result of sequencing Q after P, process-wise. We assume that input programs are in normal form, however, our rewriting rules produce programs that might not be.

3.2 Semantics

Next, we describe the behavior of IceT programs.

Sends and Receives Processes communicate by sending and receiving messages. The statement send(t, p, e, r) asynchronously sends the value of an expression e (of type t) to process p, in round r. Dually, x ← recv(p, t, r) *blocks* until it receives a value of type t from process p in round r, and then assigns the received value to the variable x. A non-blocking receive x ← recvTO(p, t, r) waits for a message of type t from process p in round r. If a message with value v is received in time, the receive assigns Just v to x, however, it can time out non-deterministically, in which case it assigns None. Blocking and non-blocking receives are used in different settings: blocking receives require a reliable network where messages cannot be dropped, but only arbitrarily reordered (such as in Section 2.1); non-blocking receives are used with an unreliable network, where messages can be both arbitrarily reordered and dropped. Our benchmarks use the unreliable network setting. We let a receive from a set X denote a receive from any $x \in X$; a receive from $*$ represents a receive from *any* process. In contexts where message type and round are fixed, we omit them from the statements, *i.e.*, we use send(p, e) to denote a send of value of e to p and $x \leftarrow$ recv(p) to denote a receive from p. Finally, we use $x \leftarrow$ recv as an abbreviation for $x \leftarrow$ recv$(*)$.

Invariants In IceT, each for-loop is annotated with a loop invariant I_S. I_S is a *synchronous data invariant*, *i.e.*, a loop invariant over the synchronization. Synchronous data invariants are user supplied and used by our verification condition generation procedure (§ 6), which checks that the invariants indeed hold on the *synchronization*. In addition to the data invariant, our rewrite requires a *communication* invariant I_C that is checked during the rewrite to a synchronous program (§ 4). Communication invariants contain information on who to send to or receive from. We include these annotations for our completeness result, but we find that, in practice, these invariants are very simple, *i.e.*, either true or stating that some variable contains a certain process ID. For all our examples, Goolong computes communication invariants completely automatically.

Atomic Actions and Pick The language includes atomic actions $\langle e \triangleright P \rangle$, which are *not* written by the user – instead, they are computed by our rewriting rules (§ 4). An action $\langle P \triangleright e \rangle$ comprises a program P that executes *atomically* and an *annotation* e that is derived from the supplied synchronous data invariants, and used to generate verification conditions (§ 6). We sometimes omit e if it equals true. Our rewriting rules also generate statements of the form x ← pick(X), in which variable x is assigned an arbitrary element of X, however, this statement is not exposed in the surface language.

Semantics The semantics of IceT is standard, however, we provide a full formalization the supplementary material. Program configurations C are either triples of the form $C \triangleq (\sigma, msgs, P)$ consisting of a store σ, a multi-set of in-flight messages $msgs$, and program P or a distinguished configuration crash used to represent assertion failures. Importantly, all processes are halted in the crash configuration, which allows us to encode local assertion violations as halting properties. Stores map a variable and ID to a value. Message multi-sets are maps of type (ID × ID × MsgType × Round × Values) ⟶ Int. Sends add messages to the set while receives and message drops remove them. We write $C \rightarrow C'$ to mean that configuration C' is reachable from C by executing some sequences of singleton processes statements.

Properties A property φ is a first-order formula (over some background theory): a store σ satisfies φ (we write $\sigma \models \varphi$) if and only if φ is true when its free variables are given their valuation from σ. Satisfaction is lifted to configurations in the obvious way. Assume statements assume(φ) reduce to skip when φ is true and diverge (loop forever) otherwise. Assert statements assert(φ) reduce to skip when φ is true and otherwise transition to the crash configuration, where every process is halted.

Halting Properties and Invariance In this paper we are concerned with verifying *halting properties* of programs, that is, properties over halted states, *i.e.*, states in which no process can take any further step. Examples of halting properties range from local assertion safety to global properties

$$P \triangleq [\mathtt{send}(t, c, v)\ l_0]_a \parallel [\mathtt{send}(t, c, v)\ l_1]_b \parallel [\mathtt{recv}(*, t)\ \{l_0, l_1\}]_c$$

$$Q \triangleq \Big[\mathtt{for}\ \_ \in as\ \mathtt{do}\ \_ \leftarrow \mathtt{recv}(*, t)\ \{l_0\}\ \mathtt{end}\Big]_b \parallel \prod (a \in as).\, [\mathtt{send}(t, b, v)\ l_0]_a$$

Fig. 10. Examples: Symmetric Races

$$\begin{array}{rcll} \textsc{msgs} & \in & (\mathrm{ID} \times \mathrm{ID} \times \mathsf{MsgType} \times \textsc{Round} \times \mathsf{Exp}) \to \textsc{Int} & \textbf{Message Sets} \\ \textsc{Asrt} & ::= & \emptyset \mid \textsc{Asrt} \cup \{\mathtt{unfold}(p, x, ps)\} & \textbf{Assertions} \\ \Gamma & ::= & (\textsc{msgs}, \textsc{Asrt}) & \textbf{Contexts} \end{array}$$

Fig. 11. Syntax for context Γ.

such as deadlock freedom or agreement as in two-phase commit (see Equation 2.1). We write $P \models_H \varphi$ to denote that φ holds in all halted configurations reachable from the initial configuration. For our rewrites, we sometimes need to ensure that properties are preserved under interference from parallel processes that share variables. We write $P \models_I \varphi$ to denote that φ is invariant under P, *i.e.*, that if φ holds initially, then φ holds after any number of execution steps.

3.3 Almost Symmetric Races

Our method requires programs to only contain *almost symmetric races*. In order to check this property for a given program, we first annotate each syntactic send statement with a unique *tag*. For each syntactic receive, we then compute the *set of tags* it can receive from. In order to determine whether a program only contains almost symmetric races, we require that, for each syntactic receive, the set of tags it can receive from must consist of *at most one syntactic send* statement. While, in general, it is hard to compute the set of send-tags a given syntactic receive can receive messages from, we use the *types* of the sent and received messages to compute a sound over-approximation by adding all sends with the correct type, round, and sender specification to the set (*e.g.*, in § 2.1 proposals, decisions and acknowledgements have different types, thereby ensuring that they can be received in the correct place, only). This provides a simple syntactic check.

Eliminating Wildcard Receives If a program only contains almost symmetric races, we can easily eliminate wildcard receives by replacing them with either a receive from the unique corresponding process or the symmetric set of IDs.

Example 3.1. Consider programs P and Q shown in Figure 10. Sends are annotated with tags in red and receives are annotated with sets of tags shown in blue. The receive in P violates the property of almost symmetric races because it could return a message sent by either a or b. In Q, the receive in c can receive a message from any of the processes in as, but each message originates at the single send statement tagged with l_0. As a result, the receive statement $\mathtt{recv}(*, t)$ can be replaced by $\mathtt{recv}(as, t)$.

Ensuring that a program only contains almost symmetric races, together with our language restrictions, provides us with the following crucial guarantee: whenever a process is expecting a message of a fixed type and round, from a fixed other process, there can be at most one such message. This property follows from the definition of symmetric races and the fact that our language restrictions ensure that any two iterations of the same (`foreach`- or `for`-) loop send to different processes.

4 REWRITE RULES

In this section, we describe our rewriting rules for producing synchronizations.

$$\textsc{R-Send}\quad \frac{\Delta, \Sigma \models x = q \qquad q \text{ is a PID} \qquad m \text{ fresh} \qquad \Gamma' = \Gamma \cup \{(p, q, t, r, m)\}}{\Gamma, \Delta, \Sigma, [\mathtt{send}(x, n, t, r)]_p \rightsquigarrow \Gamma', \left(\Delta; [\mathtt{m} \leftarrow \mathtt{n}]_p\right), \Sigma, \mathtt{skip}}$$

$$\textsc{R-Recv}\quad \frac{\Delta, \Sigma \models x = p \qquad p \text{ is a PID} \qquad (p, q, t, r, m) \in \Gamma \qquad \Gamma' = \Gamma - \{(p, q, t, r, m)\}}{\Gamma, \Delta, \Sigma, [\mathtt{y} \leftarrow \mathtt{recv(t, x, r)}]_q \rightsquigarrow \Gamma', \left(\Delta; [y \leftarrow p.m]_q\right), \Sigma, \mathtt{skip}}$$

$$\textsc{R-RecvTO}\quad \frac{\Gamma, \Delta, \Sigma, [\mathtt{y} \leftarrow \mathtt{recv(t, x, r)}]_q \rightsquigarrow \Gamma', \left(\Delta; [y \leftarrow p.m]_q\right), \Sigma, \mathtt{skip}}{\Gamma, \Delta, \Sigma, [\mathtt{y} \leftarrow \mathtt{recvTO(t, x, r)}]_q \rightsquigarrow \Gamma', \left(\Delta; [y \leftarrow \mathtt{Just}\ p.m]_q \oplus [y \leftarrow \mathtt{None}]_q\right), \Sigma, \mathtt{skip}}$$

$$\textsc{R-Choice}\quad \frac{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma, (\Delta; \Delta_A), \Sigma, \mathtt{skip} \qquad \Gamma, \Delta, \Sigma, B \rightsquigarrow \Gamma, (\Delta; \Delta_B), \Sigma, \mathtt{skip}}{\Gamma, \Delta, \Sigma, A \oplus B \rightsquigarrow \Gamma, (\Delta; \Delta_A \oplus \Delta_B), \Sigma, \mathtt{skip}} \qquad \textsc{R-False}\quad \frac{\Delta \models \mathit{false}}{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma, \Delta, \Sigma, \mathtt{skip}}$$

$$\textsc{R-Context}\quad \frac{\Gamma, \Delta, \Sigma, A \rightsquigarrow \Gamma', \Delta', \Sigma', A'}{\Gamma, \Delta, \Sigma, A \circ B \rightsquigarrow \Gamma', \Delta', \Sigma', A' \circ B}$$

Fig. 12. Proof Rules (Basic Statements).

Symbolic States Each rewriting rule defines a relation between a pair of *symbolic states*, whose syntax is summarized in Figure 11. A symbolic state comprises a *context* (Γ); an already *synchronized* part of the program, consisting of a sequential prefix Δ and parallel context Σ; and a program (P) in normal form, which is yet to be rewritten. Both Δ and Σ are IceT programs *without* sends or receives. A context Γ consists of a symbolic set of messages $\textsc{msgs}$ and a set of assertions $\textsc{Asrt}$. $\textsc{msgs}$ is a multi-set of in-flight messages. $\textsc{Asrt}$ contains assertions about process identifiers and is populated during the rewrite. Finally, we sometimes use Γ to refer to its individual parts, *i.e.*, for context $\Gamma \triangleq (\textsc{msgs}, \textsc{Asrt})$ we write $\Gamma \cup \{msg\}$ to mean $\textsc{msgs} \cup \{msg\}$ and $\Gamma \vdash a$ to mean $a \in \textsc{Asrt}$.

Rewriting Rules Each rewriting rule defines a judgment of the form $\Gamma, \Delta, \Sigma, P \rightsquigarrow \Gamma', \Delta', \Sigma', P'$. The goal of each step is to move parts of program P into the synchronization (Δ, Σ) such that eventually we can rewrite P to skip. We now describe the main rules of our method, starting with rules for loop-free programs, programs that contain communication between a single process and a group of processes, and finally communication between sets of processes.

4.1 Loop-Free Programs

Figure 12 shows the rules for rewriting loop-free programs.

Message Passing Rule R-Send handles message sends. The rule rewrites a send from process p to some x into skip if x evaluates to a process ID q. The rule checks that x corresponds to some ID q through the condition $\Delta, \Sigma \models x = q$, where we write $\Delta, \Sigma \models \varphi$ to say that formula φ is valid after executing Δ and despite interference from Σ. More formally, $\Delta, \Sigma \models \varphi$ holds if and only if $\Delta \models_H \varphi$ and $\Sigma \models_I \varphi$ hold. The rule takes a "snapshot" of n's value by adding an assignment of n to a fresh variable m to the prefix Δ. This is necessary as n's value might change due to later

$$\mathrm{Ex1}_{\textsc{Async}} \triangleq \begin{bmatrix}\texttt{send}(q,\ \texttt{ping});\\ \_ \leftarrow \texttt{recv}(q)\end{bmatrix}_p \parallel \begin{bmatrix}\texttt{v} \leftarrow \texttt{recv}(p);\\ \texttt{send}(p,\ \texttt{Ack})\end{bmatrix}_q$$

(a) **Asynchronous Program**

$$\Big[\texttt{skip};\ \_ \leftarrow \texttt{recv}(q)\Big]_p \parallel \Big[\texttt{v} \leftarrow \texttt{recv}(p);\ \texttt{send}(p,\ \texttt{Ack})\Big]_q$$

(b) **After R-Send**

$$\Big[\texttt{skip};\ \_ \leftarrow \texttt{recv}(q)\Big]_p \parallel \Big[\texttt{skip};\ \texttt{send}(p,\ \texttt{Ack})\Big]_q$$

(c) **After R-Recv**

$$\mathrm{Ex1}_{\textsc{Sync}} \triangleq [\texttt{v} \leftarrow \texttt{ping}]_q\ ;\ [\_ \leftarrow \texttt{Ack}]_p$$

(d) **Target Synchronous Program**

Fig. 13. Rewrite of Ex1 (Basic Statements).

assignments. It then updates msgs by adding a message for m between p and q. Note that the message is *symbolic i.e.*, it contains identifier m rather than m's value. Rule R-Recv rewrites a receive from x into `skip`, if x corresponds to a ID p. R-Recv removes a matching message m from msgs and sequences the corresponding assignment after Δ. R-RecvTO performs the same task as R-Recv, however, it case-splits over whether the receive timed out.

Branching and Context IceT handles branching through nondeterministic choice, where $A \oplus B$ denotes a choice between A and B. The rule R-Choice rewrites a program with branching by rewriting each branch independently, *i.e.*, to rewrite $A \oplus B$, both A and B must be rewritten to `skip`. The resulting synchronous prefix consists of a choice between the prefixes of the individual branches. For simplicity, we require the branches to produce the initial symbolic context Γ, however, the rule can be generalized by allowing the branches to produce a different context under the condition that the contexts of both branches are equivalent modulo a renaming of fresh variables. Rule R-False rewrites unreachable program points to `skip`. Rule R-Context allows rewriting a program A independently of statements that are executed after, or in parallel. Our method additionally contains rule R-Congruence (omitted for brevity), which allows rewriting trivially equivalent programs. For instance, the rule allows rewriting $(\texttt{skip}; P)$ to P, for any program P.

Example 4.1. Consider example Ex1 from § 2.2 which we reproduced in Figure 13a and where we have replaced wildcard receives with receives from the respective processes. The goal is to rewrite $\mathrm{Ex1}_{\textsc{Async}}$ to `skip` producing the synchronization $\mathrm{Ex1}_{\textsc{Sync}}$ in Figure 13d. The initial context is $(\emptyset, \emptyset)$. We first apply R-Context to select the send statement in p's program, and then apply rule R-Send to reduce the send to `skip`, producing the program in Figure 13b and updating Γ with message $(p, q, \top, r, \texttt{ping})$, where we omitted the fresh "snapshot" variable introduced by R-Send for presentation. Next, we apply R-Context with R-Recv to yield the program in Figure 13c with context $(\emptyset, \emptyset)$ and prefix $[\texttt{v} \leftarrow \texttt{ping}]_q$. Applying the congruence $\texttt{skip}; s \equiv s$ and repeating the same sequence of rules yields the target $\mathrm{Ex1}_{\textsc{Sync}}$.

4.2 Single Process to Group of Processes

Next, we present our method for rewriting loops over symmetric sets of processes. At a high-level, in order to reason about an unbounded collection of processes, we (1) *focus* on a single (arbitrarily chosen) process, (2) *synchronize* the interactions with that process, and (3) *generalize* to the entire set. This strategy inspired by techniques such as temporal case splitting from Model

Checking [McMillan 1999], or equivalently, unfold & fold or focus & adoption for Linear Type systems [Fahndrich and DeLine 2002].

R-Loop

$$\frac{\begin{array}{l}(1)\ \ u, x\ \mathtt{fresh}\\ (2)\ \ \Gamma_0 \triangleq \Gamma \cup \{\mathtt{unfold}(u, x, ps)\}\ \ \text{and}\ \Delta_0 \triangleq \mathtt{assume}(I_C)\\ (3)\ \Delta, \Sigma \models I_C\ \ \text{and}\ \ (\Delta_0; \langle \Delta^u \rangle), \Sigma \models I_C\\ \Gamma_0, \Delta_0, \Sigma, [A]_u \parallel B[x/p] \rightsquigarrow \Gamma, (\Delta_0; \Delta^u), \Sigma, \mathtt{skip}\end{array}}{\begin{array}{l}\Gamma, \Delta, \Sigma, \prod (p \in ps).\ \left[A\right]_p \parallel \left[\mathtt{for}\ p \in ps\ \{I_S\}\ \mathtt{do}\ B\ \mathtt{end}\right]_q \rightsquigarrow\\ \Gamma, \left[\mathtt{for}\ p \in ps\ \mathtt{do}\ \langle I_S \triangleright \Delta^u[p/u] \rangle\ \mathtt{end}\right], \Sigma, \mathtt{skip}\end{array}}$$

Fig. 14. Proof Rules (Iteration over sets of processes).

Iterating Over Symmetric Process Identifiers We formalize this strategy in our rewrite rule R-Loop shown in Figure 14. R-Loop rewrites the interaction between a set of symmetric processes ps and a single process q that iterates over ps. In order to rewrite the entire loop, R-Loop requires showing that a *single interaction* between q and an arbitrary process $u \in P$ can be rewritten. For this, the rule picks a fresh process identifier u and a fresh loop variable x in condition (1). It then adds a permission to communicate with process u (*i.e.*, to "unfold" u from set P) to the context Γ, and strengthens the prefix Δ by assuming a communication invariant I_C in condition (2). Intuitively, permissions ensure that each iteration communicates with a single process, while communication invariants contain information about who to send to and receive from. Finally, the rule requires invariant I_C to be inductive in condition (3). If the rewrite succeeds, the rule extends the synchronous prefix by repeating the synchronization that was computed for a single iteration, once for every process in ps. Importantly, since each iteration communicates with a single process only, it can be executed atomically. Note that the context Γ after the rewrite is required to be the same as the initial context. This ensures that messages cannot be leaked across iterations.

Unfolded Processes Figure 15 contains the rules for unfolding process u from a set ps. Rule R-Send-Unfold binds u to x allowing the process that iterates over P to initiate communication by sending to u using its loop variable. Rule R-Recv-Unfold unfolds u through a receive, *i.e.*, it transforms a receive from an arbitrary process in ps to a receive from u. Informally, this can be thought of as an application of symmetry, as described in Section 2.2. It then havocs the loop variable x by assigning

R-Send-Unfold

$$\frac{\Gamma \vdash \mathtt{unfold}(u, x, ps) \qquad \Gamma' \triangleq \Gamma - \{\mathtt{unfold}(u, x, ps)\}}{\Gamma, \Delta, \Sigma, \mathtt{send}(t, x, n) \rightsquigarrow \Gamma', (\Delta\ ;\ \mathtt{assume}(x = u)), \Sigma, \mathtt{send}(t, x, n)}$$

R-Recv-Unfold

$$\frac{\Gamma \vdash \mathtt{unfold}(u, x, ps) \quad \Gamma' \triangleq \Gamma - \{\mathtt{unfold}(u, x, ps)\}}{\Gamma, \Delta, \Sigma, \mathsf{y} \leftarrow \mathsf{recv}(\mathsf{ps}, \mathsf{t}) \rightsquigarrow \Gamma', (\Delta\ ;\ \mathsf{x} \leftarrow \mathsf{pick}(\mathsf{ps})), \Sigma', \mathsf{y} \leftarrow \mathsf{recv}(\mathsf{u}, \mathsf{t})}$$

Fig. 15. Proof Rules (unfolding).

$$\text{Ex2}_{\textsc{Async}} \triangleq \left[\begin{array}{l}\texttt{for } q \in qs \texttt{ do}\\ \quad \texttt{send}(q,\ \texttt{ping});\\ \quad \_ \leftarrow \texttt{recv}(q)\\ \texttt{end}\end{array}\right]_p \parallel \prod(q \in qs)\left[\begin{array}{l}\texttt{v} \leftarrow \texttt{recv}(p);\\ \texttt{send}(p,\ \texttt{Ack})\end{array}\right]_q$$

(a) **Asynchronous Program**

$$\left[\begin{array}{l}\texttt{send}(x,\ \texttt{ping});\\ \_ \leftarrow \texttt{recv}(x)\end{array}\right]_p \parallel \left[\texttt{v} \leftarrow \texttt{recv}(p);\ \texttt{send}(p,\ \texttt{Ack})\right]_u$$

(b) **Obligation after applying R-Loop**

$$\text{Ex2}_{\textsc{Sync}} \triangleq \texttt{for } q \in qs \texttt{ do } [\texttt{v} \leftarrow \texttt{ping}]_q\ ;\ [\_ \leftarrow \texttt{Ack}]_p \texttt{ end}$$

(c) **Target Synchronous Program**

Fig. 16. Rewrite of Ex2 (Iteration over sets of processes).

an arbitrary process in ps. Note that both rules consume the unfold permission, *i.e.*, only a single process can be unfolded in a given iteration.

Example 4.2. Figure 16a shows Ex2 from § 2.2. We want to rewrite $\text{Ex2}_{\textsc{Async}}$ into its synchronization $\text{Ex2}_{\textsc{Sync}}$ shown in Figure 16c. For brevity, we omit the synchronous invariant. We start from the initial context $\Gamma_0 \triangleq (\emptyset, \emptyset)$ and apply R-Loop which picks an arbitrary iteration of the loop, generating the rewrite obligation shown in Figure 16b. As the new obligation occurs in a context Γ where $\Gamma \vdash \texttt{unfold}(u, x, qs)$, we can apply R-Send-Unfold which consumes the permission and binds x to u. The rewrite of the body does not depend on any loop-carried state, so $I_C \triangleq \texttt{true}$ suffices to rewrite the program to $\texttt{skip}$ with prefix $\left([\texttt{v} \leftarrow \texttt{ping}]_u; [\_ \leftarrow \texttt{Ack}]_p\right)$. Finally, since this rewrite matches the precondition in rule R-Loop, we can rewrite $\text{Ex2}_{\textsc{Async}}$ to $\texttt{skip}$ with prefix $\text{Ex2}_{\textsc{Sync}}$ shown in Figure 16c.

4.3 Group of Processes to Group of Processes

Finally, we turn to pairs of (symmetric) groups of processes.

Set to Set The rule R-Focus shown in Figure 18 handles rewriting the parallel composition of two *sets* of IDs, ps and qs. Analogous to R-Loop, the rule works by (a) *focusing* on an arbitrary element $u \in ps$, (b) *rewriting* the interaction of u with the members of qs, and (c) *generalizing* the interactions to all the members of ps. As before, the rule picks a fresh pid u in condition (1), adds a permission to communicate with process u to the context Γ, and strengthens the prefix Δ by assuming a communication invariant I_C in (2). The rule then requires that the interaction between u and an arbitrary iteration of each process in qs can be rewritten. If the rewrite succeeds, the rule extends the synchronous context Σ by repeating the synchronization Δ^u in parallel, once for each $p \in ps$ in condition (3). Finally, in condition (4), the rule requires that the communication invariant is inductive and preserved under interference from parallel processes.

Example 4.3. We now examine Ex4 from § 2.2 which we reproduce in Figure 17a. We again omit synchronous invariants for readability. We start the rewrite by applying R-Focus followed by an application of R-Loop which creates an obligation to rewrite the program shown in Figure 17b to $\texttt{skip}$ in a context Γ where $\Gamma \vdash \texttt{unfold}(q^u, x, qs)$ and $\Gamma \vdash \texttt{unfold}(p^u, \_, ps)$, with communication invariant true. This allow us to first apply R-Recv-Unfold in order to transform the receive from the *set* ps into a receive from the *process* p^u and then R-Send-Unfold to bind x to q^u allowing

$$\prod_{(p\in ps)} \left[\begin{array}{l} \texttt{for } q \in qs \texttt{ do} \\ \quad \texttt{send}(q,\ (p,\ \texttt{ping})); \\ \quad \_ \leftarrow \texttt{recv}(q) \\ \texttt{end} \end{array}\right]_p \parallel \prod_{(q\in qs)} \left[\begin{array}{l} \texttt{foreach } ps \texttt{ do} \\ \quad (\texttt{id},\ \texttt{v}) \leftarrow \texttt{recv}(ps); \\ \quad \texttt{send}(\texttt{id},\ \texttt{Ack}) \\ \texttt{end} \end{array}\right]_q$$

(a) **Asynchronous Program**

$$\left[\begin{array}{l} \texttt{send}(x, (p^u, \texttt{ping})); \\ \_ \leftarrow \texttt{recv(x)} \end{array}\right]_{p^u} \parallel \left[\begin{array}{l} (\texttt{id}, \texttt{v}) \leftarrow \texttt{recv}(ps); \\ \texttt{send(id, Ack)} \end{array}\right]_{q^u}$$

(b) **Obligation after applying R-Focus, R-Loop**

$$\Delta_u(p^u, q^u) \triangleq \left\langle [(\texttt{id}, \texttt{v}) \leftarrow (p^u, \texttt{ping})]_{q^u}; [\_ \leftarrow \texttt{Ack}]_{p^u} \right\rangle$$

(c) **Synchronization computed by R-Atomic**

$$(\textstyle\prod (p \in ps).\texttt{for } q \in qs \texttt{ do } \Delta_u(p, q) \texttt{ end})$$

(d) **Target Synchronous Program**

Fig. 17. Rewrite of Ex4 (Communicating process sets).

R-Focus

$$\frac{\begin{array}{l} (1)\ \ u \text{ fresh} \\ (2)\ \ \Gamma_0 \triangleq \Gamma \cup \{\texttt{unfold}(u, \_, ps)\} \text{ and } \Delta_0 \triangleq \texttt{assume}(I_C) \\ (3)\ \Sigma' = \left(\Sigma \parallel \prod (p \in ps).\Delta^u[p/u]\right) \\ (4)\ \Delta, \Sigma' \vDash I_C \ \text{ and } \ (\Delta_0; \Delta^u), \Sigma' \vDash I_C \\ \Gamma_0, \Delta_0, \texttt{skip}, [A]_u \parallel \prod (q \in qs). [B]_q \rightsquigarrow \Gamma_0, (\Delta_0; \Delta^u), \texttt{skip}, \texttt{skip} \end{array}}{\Gamma, \Delta, \Sigma, \prod (p \in ps). \left[A\right]_p \parallel \prod (q \in qs). \left[\texttt{foreach } ps \texttt{ do } B\right]_q \rightsquigarrow \Gamma, \Delta, \Sigma', \texttt{skip}}$$

Fig. 18. Proof Rules (Rewriting a single process in a set).

process p^u to send to process q^u. Rewriting the resulting program to skip yields the synchronous prefix shown in Figure 17c, and finally, closing the open obligations yields Figure 17d.

4.4 Rewrite Soundness

Our soundness theorem states that each rewriting step preserves the reachability of halted states.

Definitions For a synchronization (Δ, Σ), context Γ and program P, we write $C \in [\![\Delta, \Sigma, \Gamma, P]\!]$ if C is a configuration whose buffer is consistent with Γ (i.e., evaluating Γ on C's store yields C's buffer), whose store is reachable by executing $(\Delta; P \parallel \Sigma)$ from some initial state and whose program consists of P in parallel with some unexecuted part of Σ. Let $C \approx C'$ when $C = C' = \texttt{crash}$ or C and C' have equal stores.

Theorem 4.4 (Rewrite rule soundness). *Let A be a program in normal form that only contains almost-symmetric races, and assume $\Gamma_A, \Delta_A, \Sigma_A, A \rightsquigarrow \Gamma_B, \Delta_B, \Sigma_B, B$ and $C_A \in [\![\Delta_A, \Sigma_A, \Gamma_A, A]\!]$ and $C_A \to C_H$ where C_H is halted. Then there is $C_B \in [\![\Delta_B, \Sigma_B, \Gamma_B, B]\!]$ and C'_H s.t. $C_B \to C'_H$ and $C_H \approx C'_H$.*

Proof. By induction over the derivation of $\Gamma_A, \Delta_A, \Sigma, A \rightsquigarrow \Gamma_B, \Delta_B, \Sigma, B$, where we prove a generalized version of the above statement showing that the property holds in the presence of an arbitrary extension E as long as the resulting program only has symmetric races. We include the full proof and its supporting lemmas in our supplementary material. □

4.5 Completeness: Stratified Pairwise Communication

We now define a class of programs called *Stratified Pairwise Communication Protocols (SPCP)* and show that our rewrite method always produces a synchronization for programs in this class. Figure 19 shows a set of inference rules that inductively define the set SPCP. SPCP contains all our examples, and benchmarks, except for Paxos. For an IceT program P, we let talksto(P) denote the set of processes that P sends to, or receives from (we assume P has no wildcard receives). Rule R-Base defines the base case: a program consisting of two processes is an SPCP program if both processes are loop and race free, and the processes only communicate with each other. Rule R-Broadcast transforms a base-case protocol between processes p and q by repeating the interaction, once for each process q in a fresh set qs. For this, the rule makes use of a predicate choose which chooses a process to interact with in the current iteration. This process can either be an arbitrary process from qs, or the process bound in the loop variable. That is, we define choose(q, qs, id) as either choose$(q, qs, id) \triangleq$ `id ← q` or choose$(q, qs, id) \triangleq$ `id ← recv(qs)`. Similarly, rule R-Multicast transforms an interaction between a single process p and set of processes qs by repeating it once for each process in a fresh set ps, however, each process in qs picks an arbitrary process in ps to interact with. Finally, rule R-Compose transforms two SPCP programs into a new SPCP through sequential process-wise composition.

Example 4.5. The examples in § 2 as well as Ex6 shown below are SPCP programs.

$$\mathrm{Ex6} \triangleq \prod (p \in ps). \begin{bmatrix} \begin{array}{l} \mathtt{for\ q \in qs\ do} \\ \quad \mathtt{send(q, p);} \\ \mathtt{end;} \\ \mathtt{for\ \_ \in qs\ do} \\ \quad \mathtt{\_ \leftarrow recv(qs)} \\ \mathtt{end;} \\ \mathtt{send(m, \_)} \end{array} \end{bmatrix}_p \parallel \prod (q \in qs). \begin{bmatrix} \begin{array}{l} \mathtt{foreach}\ ps\ \mathtt{do} \\ \quad \mathtt{id \leftarrow recv(ps)\ ;} \\ \quad \mathtt{send(id, \_)} \\ \mathtt{end\ ;} \end{array} \end{bmatrix}_q \parallel \begin{bmatrix} \begin{array}{l} \mathtt{for\ p \in ps\ do} \\ \quad \mathtt{\_ \leftarrow recv(ps)} \\ \mathtt{end} \end{array} \end{bmatrix}_m$$

Synchronizability An IceT program P is called *synchronizable* if and only if all halting states of P are valid final states (*i.e.*, the program does not deadlock) and have empty message buffers (*i.e.*, there are no spurious sends). We use $\emptyset$ to denote both the empty store and the empty buffer. Then, P is synchronizable if, whenever$(\emptyset\ , \emptyset,\ P) \rightarrow C_H$, and C_H is halted, then $C_H \triangleq (\_,\ \emptyset,\ \mathtt{skip})$.

Proposition 4.6. *If $P \in SPCP$, then P is synchronizable if and only if there exists a derivation* $\emptyset, \mathtt{skip}, \mathtt{skip}, P \rightsquigarrow \emptyset, \Delta, \Sigma, \mathtt{skip}$, *for some synchronization* (Δ, Σ).

5 REPEATING PROTOCOLS USING ROUNDS

We now discuss our mechanism for repeating protocols in rounds. We first introduce our language primitives and their semantics, and then show conditions under which we can reduce the problem of proving a halting property of the form $P \models_H \forall r \in \textsc{Round} : \varphi(r)$ stating that some property φ holds for all rounds r of program P to the problem proving that φ holds for an arbitrary round r^*.

Protocol Rounds Let Round denote a data type representing a protocol round. Rounds can be compared for equality, but not for ordering. Our language contains the following two iteration primitives over rounds. Primitive `rounds` $r \in R$ `do` P `end` repeats program P once for each round

R-Base

$$\frac{\begin{array}{l} P \triangleq ([A]_p \parallel [B]_q) \\ P \text{ loop-free and race-free} \\ \mathsf{talksto}(A) \subseteq \{q\} \text{ and } \mathsf{talksto}(B) \subseteq \{p\} \end{array}}{P \in \mathrm{SPCP}}$$

R-Broadcast

$$\frac{qs \text{ fresh} \qquad \left([A]_p \parallel [B]_q\right) \in \mathrm{SPCP}}{\left(\left[\begin{array}{l} \mathtt{for}\ q \in qs\ \mathtt{do} \\ \quad \mathtt{choose}(q, qs, id); \\ \quad A[id/q] \\ \mathtt{end} \end{array}\right]_p \parallel \prod (q \in qs).[B]_q\right) \in \mathrm{SPCP}}$$

R-Multicast

$$\frac{ps \text{ fresh} \qquad \left([A]_p \parallel \prod (q \in qs).[\mathtt{id} \leftarrow \mathtt{recv(p)}\ ;\ B]_q\right) \in \mathrm{SPCP}}{\left(\prod (p \in ps).\,[A]_p \parallel \prod (q \in qs).\mathtt{foreach}\ ps\ \mathtt{do}\ [\mathtt{id} \leftarrow \mathtt{recv(ps)}\ ;\ B]_q\right) \in \mathrm{SPCP}}$$

R-Compose

$$\frac{P \in \mathrm{SPCP} \qquad Q \in \mathrm{SPCP}}{P \circ Q \in \mathrm{SPCP}}$$

Fig. 19. Definition of Stratified Pairwise Communication Protocols (SPCP) Programs.

$r \in R$, where r is bound in P and primitive `repeat do` P `end` iterates P indefinitely. Values of type Round can only be created or modified by `rounds`. Furthermore, in each iteration of a `repeat` loop, we require that the loop binds a single round number through a receive, as first action. For each process P occurring under `rounds`, we transform P into a program where each local variable x of value type Val is turned into a map from Round to Val.

Example 5.1. Consider again Figure 8a from Section 2.2. Figure 8a shows the multi-round variant of example Ex2 from Section 2.2. Process p iterates over a set of rounds R using `rounds` and each process in qs repeats its protocol using `repeat`. The local variables v of each process $q \in qs$ have been transformed into a map from Round to v. Each $q \in qs$ receives the current round number and uses it to index v. Ex5 satisfies our syntactic requirement, as each $q \in qs$ binds its round number through a receive, as a first action.

Round Non-Interference Our reduction exploits the fact that sequential rounds of a single process can be transformed into equivalent parallel rounds, if the sequential rounds are independent of each other. We define a notion of independence called *round non-interference* which is inspired by classic work on loop parallelization [Bernstein 1966]. Let a memory location denote either a variable, or a map at a given index. Let A be a loop body, and $A(r)$ be A instantiated to round r. We say that two loop iterations $A(r)$ and $A(r')$ exhibit round non-interference if

1) $A(r)$ does not write to a location that is read by $A(r')$ and $A(r')$ does not read a location written by $A(r)$

2) $A(r)$ does not write to a location that is also written to by $A(r')$

3) $A(r)$ and $A(r')$ only send and receive messages using their currently bound round.

We say that a loop satisfies round non-interference if any two of its iterations do, and say that a program satisfies round non-interference if all its loops do. Note that in particular, we can ensure that properties *1)* and *2)* hold by allowing process to only access variables indexed by its current round. Intuitively, if a loop with body A satisfies round non-interference, $A(r_0)\ ;\ A(r_1)\ ;\ \ldots\ ;\ A(r_n)$ is equivalent to $A(r_0) \parallel A(r_1) \parallel \ldots \parallel A(r_n)$ with respect to halting states, as distinct iterations do not share memory and communication is contained within rounds. Using the above reasoning, we can parallelize all protocol iterations in a program P and group together the iterations belonging to a given round r. Let $\mathtt{group}(\mathsf{P}, \mathsf{r})$ denote the result of this process for a program P and round r.

Proposition 5.2. *For all programs P, rounds r and r', and properties φ if P exhibits round non-interference and satisfies our syntactic requirements, then, if*

$$\mathtt{group}(\mathsf{P}, \mathsf{r}) \models_H \varphi(r) \textit{ and } \mathtt{group}(\mathsf{P}, \mathsf{r}')) \models_H \varphi(r') \textit{ then } \mathtt{group}(\mathsf{P}, \mathsf{r}) \parallel \mathtt{group}(\mathsf{P}, \mathsf{r}') \models_H \varphi(r) \wedge \varphi(r') \ .$$

Example 5.3. Example Ex5 exhibits round non-interference: each iteration only accesses variables indexed by the current round and processes only send and receive using the current round number.

Reduction We now show that, in order to prove a safety property φ of a multi-round program with round non-interference, it suffices to prove the property for an arbitrary single round. To state our theorem, let $P@r^*$ denote a transformation of program P that erases all occurrences of `repeat` and `rounds`, and instantiates the currently bound round to r^*.

Theorem 5.4. *For all programs P and properties φ, where r^* is a fresh round variable, if P satisfies round non-interference and satisfies our syntactic requirements, then*

$$\textit{if } P@r^* \models_H \varphi(r^*) \quad \textit{then} \quad P \models_H \forall r \in \textsc{Round} : \varphi(r)$$

Proof. For a proof by contradiction, we assume that there exists a halting state of P that does not satisfy $\varphi(\hat{r})$, in some round $\hat{r} \in R$, where P iterates over R. Since P satisfies round non-interference, P is equivalent to a parallel composition over $\mathtt{group}(\mathsf{P}, \mathsf{r})$, for all $r \in R$. By proposition 5.2 we get that $\mathtt{group}(\mathsf{P}, \hat{\mathsf{r}})$ must not satisfy $\varphi(\hat{r})$. By another application of our equivalence, we get that $P@\hat{r}$ does not satisfy $\varphi(\hat{r})$. However, since r^* was left unconstrained, $P@r^*$ must also violate the property which contradicts our assumption. □

Example 5.5. Consider again Ex5. As in 2.2, we want to prove that variable v is set to `ping` upon termination, however, now we want to show that this property holds in all rounds. More formally, we want to show that $\forall r \in R : \forall q \in qs : v[r] = \mathtt{ping}$ holds upon termination. Since Ex5 exhibits round non-interference, we can invoke Theorem 5.4 which requires us to show that $\forall q \in qs : v[r^*] = \mathtt{ping}$ holds for Ex5@r^* shown in Figure 8b and allows us to reuse invariant I_5.

6 GENERATING VERIFICATION CONDITIONS

Given a program A and a property φ, we can verify that $A \models_H \varphi$ by first rewriting A to produce the synchronization (Δ, Σ), and then verifying that $(\Delta \parallel \Sigma)\,;\mathtt{assert}(\varphi) \not\twoheadrightarrow \mathsf{crash}$. Let φ be a formula over a subset of the processes of A and assume that A can be rewritten to (Δ, Σ), *i.e.*, $-, \mathtt{skip}, \mathtt{skip}, A \rightsquigarrow -, \Delta, \Sigma, B$. Then, by Theorem 4.4, the original program is safe (*i.e.*, $A \models_H \varphi$) if the *rewritten* program is safe (*i.e.*, $(\Delta \parallel \Sigma)\,;\mathtt{assert}(\varphi) \not\twoheadrightarrow \mathsf{crash}$) and φ does not mention state in B.

Verification Conditions To check if $\Delta; \mathtt{assert}(\varphi) \not\twoheadrightarrow \mathsf{crash}$, we compute its verification condition (VC) by computing its *weakest precondition* and checking to see if it is implied by the initial state. The procedure $\mathsf{wp}(s, \varphi)$ defines the weakest precondition computation: if s is a program and φ is a formula, then $\mathsf{wp}(s, \varphi)$ is a formula describing the initial states that guarantees that s only terminates in states satisfying φ. The definition of $\mathsf{wp}(s, \varphi)$ is deferred omit for brevity.

Assignments and Loops As IceT programs contain unbounded sets of processes, variables are modeled as maps from ID to values. Thus, $\mathsf{wp}(p.x \leftarrow e, \varphi)$ substitutes x with the *map* $x[p \leftarrow e]$ in φ. Determining the weakest precondition of a loop requires a user-supplied loop invariant. The $\mathtt{for}\ x \in X\ \{I\}\ \mathtt{do} \ldots$ case uses a user-supplied invariant parameterized by *done*, an auxiliary variable that contains the elements of X that have already been processed.

Parallel Composition We define the case for parallel processes $\prod (p \in P).A$ as an analog of the standard case for loops. First, let the function $\mathsf{actions}(A)$ collect the labeled atomic actions of A. Next, we define an auxiliary variable pc_p (the program counter) that maps each process p in P to its current program location (*i.e.*, the current atomic block). The final component is an inductive invariant. The invariant is constructed from the annotations on the actions in $\mathsf{actions}(A)$: if $\langle a_l \triangleright B_l \rangle_l \in \mathsf{actions}(A)$, then the invariant includes the conjunct $\mathsf{pc}_p[P] = l \Rightarrow B$ where B is the *disjunction* of all annotations for control locations that can transition to l, *i.e.*, all B_m such that $\langle a_m \triangleright B_m \rangle_m \in \mathsf{actions}(A)$ and l is one of the immediate successors of m in the control flow of A. We define $\mathsf{wp}(\prod (p \in P).A, Q)$ as three main assertions, requiring that 1) the invariant I holds initially; 2) each transition preserves the invariant; and 3) when every process in P has terminated (*i.e.*, has its program counter set to l_{exit}), I implies Q.

Soundness and Decidability Theorem 6.1 expresses the correctness of $\mathsf{wp}(\Delta \parallel \Sigma, \mathsf{true})$.

THEOREM 6.1 (VCs). *If* $\mathsf{Valid}(\mathsf{wp}(\Delta \parallel \Sigma, \mathsf{true}))$ *then* $\Delta \not\twoheadrightarrow \mathsf{crash}$.

In general, even if the formulas asserted and assumed in a program are decidable, the naive model of message sets as arrays requires nested array reads, yielding undecidable verification conditions. However, when $-, \mathtt{skip}, \mathtt{skip}, A \rightsquigarrow -, \Delta, \Sigma, B$, the program $\Delta \parallel \Sigma$ does not contain message sets. Hence, if the formulas asserted and assumed in A are in the array property fragment [Bradley et al. 2006], then $\mathsf{wp}(\Delta \parallel \Sigma, \mathsf{true})$ is as well.

PROPOSITION 6.2 (VC DECIDABILITY). *Assume* $-, \mathtt{skip}, \mathtt{skip}, A \rightsquigarrow -, \Delta, \Sigma, B$, *and* $\psi = \mathsf{wp}(\Delta \parallel \Sigma, \mathsf{true})$, *the local states of A do not contain nested reads and all synchronous invariants, asserts and assumes are in the array property fragment. Then, checking the validity of ψ is decidable.*

7 EVALUATION

To evaluate pretend synchrony, we implement it in a tool GOOLONG[1] that takes as input a Go program A and halting property φ and (1) computes a *synchronization*, and, should one exist, (2) *verifies* the synchronization satisfies φ, thereby proving the safety of A (§ 6). We implement the rewriting step by interpreting our rewriting rules (§ 4) as a PROLOG predicate. To implement the verification step, we compute the weakest precondition of the rewritten program (§ 6) and then use Z3 to prove validity of the VC. We have used GOOLONG to develop four challenging case studies: the classic two-phase commit protocol, the Raft leader election protocol, single-decree Paxos protocol and a Multi-Paxos based distributed key-value store that employs our protocol rounds. In our tests, the key-value store outperformed other verified stores [Drăgoi et al. 2016; Taube et al. 2018a] while staying within 3x of an unverified state-of-the-art implementation [Moraru et al. 2013].

GOOLONG Library To be able to extract ICET programs from programs written in Go, we wrote a library that exposes ICET language primitives (*e.g.*, sequential iteration, symmetric processes, `send` and `recvTO` etc) in Go. In order to soundly extract protocols (in the form of ICET terms) from larger programs, the GOOLONG library keeps track of protocol specific variables: protocol variables are created explicitly using the `NewVar()` primitive and can only be read or updated through the library's `Get` and `Set` functions. All assignments from non-protocol variables are treated as non-deterministic. GOOLONG offers send and receive primitives for common types like `int`, however, implementing realistic protocols often requires custom data-types. To support such customization, GOOLONG allows the user to write their own sends/receives and marshaling for arbitrary data-structures, however, the user must provide a translation to ICET terms. GOOLONG's networking core uses TCP/IP connections and is based on the EPaxos implementation from [Moraru

[1] https://github.com/gleissen/goolong

Benchmark	Goolong						Dafny				
	#LG	#LI	#A	#I	RW (s)	Chk (s)	#LI	#A	#I	#H	Chk (s)
Two-Phase Commit	102	49	2	3	0.17	0.04	55	8	30	62	12.81
Raft Leader Election	138	44	17	6	0.19	0.18	40	20	50	73	301.68
Single-Decree Paxos	504	65	23	14	0.25	1.51	69	50	72	63	1141.35
Total	744	158	42	23	0.61	1.73	164	78	152	198	1455.84
Multi-Paxos KV	847	100	21	14	0.24	1.64	-	-	-	-	-

Table 7.1. **#LG** is the number of lines of Go code (not including proofs), **#LI** is the number of lines of extracted IceT code, **#A** is the number of lines of annotations (*e.g.*, updating ghost variables), **#I** is the number of lines of loop invariants, **RW** is the time Goolong took to compute a synchronization, **#H** is the number of lines of code needed to implement the message passing and concurrency semantics (which are not natively supported by Dafny), and **#Chk** is the time it took to generate and verify the VCs with Z3. All experiments were run on a 2.3 GHz Intel Core i7 CPU with 16 GB memory.

et al. 2013]. We write invariants as a comment, between delimiters `{-@` and `-@}`. Finally, Goolong extends `go/ast` with a pass that traverses the abstract-syntax tree of the program, and emits an IceT program that Goolong then verifies. We implemented two-phase commit, Raft leader election, single-decree Paxos and a Multi-Paxos based key-value store using our library.

Evaluation We evaluate Goolong by answering three questions: (Q1) Is Goolong *expressive* enough to enable the specification and verification of complex real-world systems? (Q2) Does Goolong *eliminate* asynchrony-related invariants to lighten the proof annotation burden? (Q3) Does Goolong enable *efficient* verification? To answer these questions, we used the state-of-the-art Dafny verifier [Leino 2010] to check the original *asynchronous* versions of all but the last of our benchmarks. In our implementation, we followed the methodology of [Hawblitzel et al. 2015a], *e.g.*, grouping statements into atomic blocks to minimize process-local interleavings. Table 7.1 summarizes the results of our comparison.

Q1: Expressiveness To demonstrate that Goolong is able to handle complex systems, we use it to verify the following four case studies: (1) *Two-Phase Commit* (as discussed in 2.1). We verify that if the transaction is committed then each participant has accepted the coordinator's proposal. (2) *Raft Leader Election* from [Ongaro and Ousterhout 2014]. We verify that no two candidates can become leader in the same term. (3) *Single-Decree Paxos* from [Lamport 2001]. We verify agreement, *i.e.*, if two proposers decide on a value, it must be the same value. (4) *Distributed Key-Value Store*, a database implementing a distributed key-value store that achieves consensus using the Multi-Paxos Protocol. We verify that the database nodes agree on the sequence of the values issued to the database.

System	Throughput (req/ms)
Goolong	118.5
PSync	32.4
Ivy-Raft*	13.5
IronKV*	~ 30

Table 7.2. Comparison of verified key value store implementations.

Q2: Proof Burden Our results in Table 7.1 show that pretend synchrony drastically simplifies proofs by reducing the number of required invariants by a factor of 6. This difference is caused by the automatic reduction computation, *i.e.*, the rewriting step yields a synchronous program composed of coarse-grained atomic blocks which simplify the invariants needed for verification. In contrast, with Dafny, we had to specify extra invariants that describe *1)* the contents of in-flight messages and *2)* invariants that case-split over the joint global state of all participating nodes which makes proofs longer and harder, as discussed in § 2.

Q3: Verification Time Table 7.1 shows that pretend synchrony dramatically reduces verification time, by three orders of magnitude. The reduction in invariants contributes to this speedup, but most of it likely stems from the elimination of message buffers and associated quantified VCs which slow down SMT solvers by triggering excessive axiom instantiations [Leino and Pit-Claudel 2016].

Distributed Key-Value Store To show that our ideas can be applied to real world applications, we implement a Multi-Paxos key-value store atop our Single-Decree Paxos implementation. Our system consists of a statically unbounded number of replicas running *proposer*, *acceptor* and *listener* from Multi-Paxos. Clients execute PUT and GET commands; requests are batched dynamically (up to 5000 requests) at server-side. Each replica maintains a *log* of instances, where each instance consists of a batch of commands, and the instance number is used as round identifier. New instances are added to the log after receiving a dedicated commit message which the leader sends after each successfully completed Paxos round. Each instance performs a full Paxos round, *i.e.*, we do not perform optimizations such as skipping the proposal phase when there is a stable leader (see § 8). Acceptors (and proposers) maintain the same data-structures as in Single-Decree Paxos, however, each acceptor indexes its bookkeeping by the instance number, *i.e.*, it keeps a map from instance number to highest received ballot, etc. To ensure round non-interference, we place the proposer inside a dedicated function. Importantly, this function does not maintain state, ensuring that individual calls are indeed independent. Similarly, the acceptor is executed in a separate GO-routine and does not share state with any other process, guaranteeing its independence. To prove agreement, we compute a hash of each instance at runtime; our proof then ensures that all replicas agree on the same instance hash. Because of our rounds mechanism, the proof from Single-Decree Paxos can be reused with only minor changes that account for acceptors indexing their bookkeeping. Similar to Single-Decree Paxos, GOOLONG verifies agreement in less than two seconds, as shown in Table 7.1.

Performance Measurements We compare our system's performance to other verified implementations. For this, we run it on three separate Amazon EC2 t2.micro instances in the same availability zone using a client executing a series of PUT or GET queries. Table Table 7.2 show the throughput compared to Psync [Drăgoi et al. 2016], Ivy [Taube et al. 2018b], and IronFleet [Hawblitzel et al. 2015a]. While we were able to run Psync, we were unable to run Ivy and IronKV and instead we provide their published results as a (albeit not directly comparable) reference. Compared to [Moraru et al. 2013], our system has a slowdown between 1.5-3x in different configurations.

8 LIMITATIONS AND FUTURE WORK

Pretend synchrony, like other techniques, has limitations and drawbacks. We discuss some of these limitations below.

General Loops and Races Our method does not support general, unstructured loops. Often, when loops are used to invoke a protocol multiple times, they can be abstracted as rounds. But, we cannot model protocols that depend on arbitrary loop carried state. One of the observations we made during this work is that distributed core protocols often only have structured loops (e.g., loops over sets of processes). Nevertheless, our approach is not applicable to arbitrary protocols. We leave the exploration of other classes of systems to future work. Similarly, pretend synchrony only supports a limited form of network non-determinism in the form of almost symmetric races (§ 3.3). GOOLONG checks this condition and returns an error witness, in case of a violation. Again, we find that many (consensus) protocols naturally fit this assumption.

Round Non-Interference Our notion of round non-interference does not allow maintaining state between rounds or comparing the ordering of round identifiers. This rules out some common optimizations. For example, in Multi-Paxos [Lamport 2001], a stable leader can skip the "proposal" phase and only execute the second, "acceptance" phase of the algorithm. This optimization requires maintaining state between different instances in the acceptor. Fortunately, this optimization is not crucial for the performance of our key-value store: we employ server-side batching where each protocol run allows agreeing on *many* commands at once, thereby diminishing the effect of protocol messages. More significantly, not being able to compare round identifiers also rules out some protocols: for instance Stoppable Paxos [Lamport et al. 2008] requires comparing the ordering of rounds to determine whether to accept a proposal. It would be interesting to explore whether our round non-interference condition can be relaxed to allow such protocols.

Pairwise Communication and Superfluous Sends Goolong enforces pairwise communication between processes, *i.e.*, (sets of) processes may talk to several other (sets of) processes, however, this communication has to happen, one-after-the-other, one interlocutor at a time (§ 4.5). In particular, Goolong enforces that, in each loop iteration, each process talks to exactly one other process. Similarly, Goolong rules out superfluous messages: each message sent must be received (unless it is dropped by the network). While these restrictions may not hold in general, we find them to be reasonable for distributed (core) protocols.

Temporal/Liveness Properties Currently, Goolong only proves safety properties. Since synchronization restricts the ordering of events and removes message buffers, one has to be careful when proving temporal/liveness properties. It would be interesting to explore which class of temporal logic properties is preserved under synchronization, where [Chaouch-Saad et al. 2009; Cohen and Lamport 1998] seem to provide a promising starting point.

Topologies Finally, at present, our technique does not handle systems that impose parameterized *topologies* such as rings, as these systems do not expose the symmetries our work exploits. This rules out protocols such as Chord [Stoica et al. 2001]. It would be interesting to explore whether pretend synchrony can be extended to these settings.

9 RELATED WORK

Verification of Distributed Systems Several recent papers focus on proving functional correctness of distributed systems and algorithms. Verdi [Wilcox et al. 2015; Woos et al. 2016] has been used to prove linearizability for Raft, while Dafny [Leino 2010] is used by [Hawblitzel et al. 2015a] to machine-verify correctness (including liveness) of other distributed system implementations. Disel [Sergey et al. 2018; Wilcox et al. 2017] aims to enable the user to *compose* protocols in an effort to modularize proofs of correctness. Similarly, [Taube et al. 2018a] aims to modularize verification of distributed algorithms by splitting verification conditions into subsets each of which uses a decidable fragment of first-order logic, were different subsets may use different fragments. [Padon et al. 2017] presents a method of (manually) transforming general, first-order verification conditions into effectively propositional logic, and applies it to verifying several Paxos variants. Finally, in [Padon et al. 2016] the user guides the system towards finding an inductive invariant by examining counterexamples. However, unlike our approach, all the above perform asynchronous reasoning which significantly complicates the invariants.

Reductions Lipton's theory of reductions has been used to identify groups of program statements that appear to act atomically with respect to other threads [Elmas et al. 2009; Flanagan and Qadeer 2003]. Our work exploits the insights from [Lipton 1975] in order to rewrite programs into their synchronous equivalents. Our use of left-movers is closely related to that of, *e.g.*, persistent sets and partial-order reduction [Abdulla et al. 2014; Flanagan and Godefroid 2005; Godefroid et al. 1996] which computes representative traces in *acyclic* state spaces. The above methods work, however, in an *explicit-state* fashion, and, unlike our approach cannot be used to compute coarse grained synchronizations which are the key to verifying functional correctness. The work in [Desai et al. 2014] is similar to ours in that our rules explore traces where buffers are small (by moving receives right after sends). However, we check parametrized programs with unbounded data, while [Desai et al. 2014] is concerned with finite state programs. [Bouajjani et al. 2018] introduces the notion of k-synchronizability: a program is k-synchronous if it is equivalent wrt. movers to a program in which all buffers are bounded by k, however the technique is not applicable to the parametrized systems we consider. Canonical sequentialization [Bakst et al. 2017] uses Lipton's movers to derive a set of rewrite rules that reduce an asynchronous program into an equivalent sequential program. However, the method is limited to checking for deadlocks as opposed to correctness. Furthermore, it does not account for the complexities of real-world distributed systems, *e.g.*, message drops, broadcasts, or rounds, and hence is inapplicable to our benchmarks (save 2PC). [Kragl et al. 2018] uses reduction to simplify reasoning about asynchronous function calls in a shared memory setting.

Parameterized Verification Counter abstraction [Pnueli et al. 2002] is a classic way to exploit symmetry. [D'Osualdo et al. 2013, 2012] convert distributed Erlang programs into a vector addition systems which can be checked for *coverability* properties, thus excluding *e.g.*, deadlock-freedom. [Farzan et al. 2014] and [Gleissenthall et al. 2016] automatically infer counting arguments in the form of counting automata or by synthesizing descriptions of sets and referring to their cardinalities, respectively. However, we require tracking the *contents* of messages which is challenging for counter-based approaches. There has been much work on inferring universally quantified invariants [Bjørner et al. 2013; Gleissenthall et al. 2016; Gurfinkel et al. 2016; Hoenicke et al. 2017; Monniaux and Alberti 2015; Sanchez et al. 2012]. By removing message buffers, pretend synchrony enables the application of those methods to asynchronous programs.

Specialized Models Many techniques seek to exploit convenient programming models or language features to simplify verification. [Drăgoi et al. 2014] develops a logic based on the Heard-Of (HO) model (a synchronous execution model with benign faults) and [Drăgoi et al. 2016] implements a DSL for developing and verifying systems in this model. [Marić et al. 2017] develops a HO language that is expressive enough to implement consensus algorithms, while producing small cutoff bounds for verifying parameterized systems. However, our approach *does not* depend upon the runtime to provide a synchronized semantics (which may get in the way of performance). Instead, Goolong constructs and reasons about a semantically equivalent, synchronized version of the given program. In recent, concurrent work [Dragoi and Widder 2018] explores how to reduce asynchronous protocols to the HO model, based on the notion of communication-closed rounds [Elrad and Francez 1982], which is similar, in spirit, to our notion of round non-interference. [Konnov et al. 2017, 2015] encode *threshold-based* distributed algorithms into counter systems and uses acceleration, which is closely related to Lipton-style reductions, to simplify the SMT-discharged proofs. The threshold restriction renders this approach inapplicable to our benchmarks. [Lange et al. 2018] infers behavioural types from Go source code and uses a model checker to analyse safety and liveness properties. Session Types [Charalambides et al. 2016; Deniélou et al. 2012; Honda 1993; Honda et al. 2012, 2008; Lange et al. 2018] project a user-provided global protocol into a set of types for the various local processes of a program. Unlike us, however, they require the user to specify the global protocol and are not concerned with functional correctness.

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 373–384.

Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. *PACMPL* 1, OOPSLA (2017), 110:1–110:27. https://doi.org/10.1145/3133934

Arthur J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Trans. Electronic Computers* 15, 5 (1966), 757–763. https://doi.org/10.1109/PGEC.1966.264565

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS*.

Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *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 II*. 372–391. https://doi.org/10.1007/978-3-319-96142-2_23

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *VMCAI*.

Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. 2009. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *Reachability Problems, 3rd International Workshop, RP 2009, Palaiseau, France, September 23-25, 2009. Proceedings*. 93–106. https://doi.org/10.1007/978-3-642-04420-5_10

Minas Charalambides, Peter Dinges, and Gul Agha. 2016. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Science of Computer Programming* 115 (2016), 100–126.

Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, Proceedings*. 317–331. https://doi.org/10.1007/BFb0055631

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

Ankush Desai, Pranav Garg, and P. Madhusudan. 2014. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 709–725. https://doi.org/10.1145/2660193.2660211

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic testing of asynchronous reactive systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 73–83.

E. D'Osualdo, J. Kochems, and C.-H. L. Ong. 2013. Automatic Verification of Erlang-Style Concurrency. In *Proceedings of the 20th Static Analysis Symposium (SAS'13)*. Springer-Verlag.

Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. 2012. Soter: An Automatic Safety Verifier for Erlang. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions (AGERE! 2012)*. ACM, New York, NY, USA, 137–140. https://doi.org/10.1145/2414639.2414658

Cezara Drăgoi, Thomas A Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A logic-based framework for verifying consensus algorithms. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 161–181.

Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. 2016. PSYNC: A partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices* 51, 1 (2016), 400–415.

C. Dragoi and J. Widder. 2018. Reducing asynchrony to synchronized rounds. *ArXiv e-prints* (April 2018). arXiv:cs.PL/1804.07078

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2009. A Calculus of Atomic Actions. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 2–15. https://doi.org/10.1145/1480881.1480885

Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173. https://doi.org/10.1016/0167-6423(83)90013-8

Manuel Fahndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/512529.512532

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2014. Proofs that count. *ACM SIGPLAN Notices* 49, 1 (2014), 151–164.

Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *ACM Sigplan Notices*, Vol. 40. ACM, 110–121.

Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 338–349.

Klaus v Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. Cardinalities and universal quantifiers for verifying parameterized systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 599–613.

Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. 1996. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Vol. 1032. Springer Heidelberg.

Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. 2016. SMT-based verification of parameterized systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016.* 338–348. https://doi.org/10.1145/2950290.2950330

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015a. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 1–17.

Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. 2015b. Automated and Modular Refinement Reasoning for Concurrent Programs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. 449–465. https://doi.org/10.1007/978-3-319-21668-3_26

Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread modularity at many levels: a pearl in compositional verification. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 473–485. http://dl.acm.org/citation.cfm?id=3009893

Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR*.

Kohei Honda, Eduardo RB Marques, Francisco Martins, Nicholas Ng, Vasco T Vasconcelos, and Nobuko Yoshida. 2012. Verification of MPI programs using session types. In *European MPI Users' Group Meeting*. Springer, 291–293.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types.. In *POPL*.

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings.*

Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. 2017. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 719–734.

Igor Konnov, Helmut Veith, and Josef Widder. 2015. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *International Conference on Computer Aided Verification*. Springer, 85–102.

Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *CONCUR*.

Leslie Lamport. 2001. Paxos Made Simple. (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report. Microsoft Research.

Butler Lampson and Howard E. Sturgis. 1976. Crash Recovery in a Distributed Data Storage System. In *Technical report XEROX Palo Alto Research Center*.

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*. ACM, 1137–1148.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

K. R. M. Leino and C. Pit-Claudel. 2016. Trigger selection strategies to stabilize program verifiers. In *CAV*.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (Dec. 1975), 717–721. https://doi.org/10.1145/361227.361234

Ognjen Marić, Christoph Sprenger, and David Basin. 2017. *Cutoff Bounds for Consensus Algorithms.* Technical Report. CAV.

Kenneth L. McMillan. 1999. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*. 219–234. https://doi.org/10.1007/3-540-48153-2_17

David Monniaux and Francesco Alberti. 2015. A Simple Abstraction of Arrays and Maps by Program Translation. In *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, Sandrine Blazy and Thomas Jensen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–234. https://doi.org/10.1007/978-3-662-48288-9_13

Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 358–372.

C. Norris IP and David L. Dill. 1996. Better verification through symmetry. *Formal Methods in System Design* 9, 1 (1996), 41–75. https://doi.org/10.1007/BF00625968

Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: an axiomatic approach. In *Communications of the ACM*.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31. https://doi.org/10.1145/3140568

Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 614–630.

Amir Pnueli, Jessie Xu, and Lenore Zuck. 2002. Liveness with (0, 1, ∞)-counter abstraction. In *International Conference on Computer Aided Verification*. Springer, 107–122.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1999. Parametric Shape Analysis via 3-valued Logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/292540.292552

Alejandro Sanchez, Sriram Sankaranarayanan, César Sánchez, and Bor-Yuh Evan Chang. 2012. *Invariant Generation for Parametrized Systems Using Self-reflection*. Springer Berlin Heidelberg, Berlin, Heidelberg, 146–163. https://doi.org/10.1007/978-3-642-33125-1_12

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. In *POPL*.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/383059.383071

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018b. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 662–677. https://doi.org/10.1145/3192366.3192414

Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018a. Modularity for Decidability of Deductive Verification with Applications to Distributed Systemswith Applications to Distributed Systems. In *PLDI*.

Nissim Francez Tzilla Elrad. 1982. Decomposition of distributed programs into communication-closed layers. In *Science of Computer Programming*.

James R. Wilcox, Ilya Sergey, and Zachary Tatlock. 2017. Programming Language Abstractions for Modularly Verified Distributed Systems. In *SNAPL*.

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 357–368.

Doug Woos, James R Wilcox, Steve Anton, Zachary Tatlock, Michael D Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, 154–165.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. 213–228.