Communication-closed asynchronous protocols

Andrei Damian[3], Cezara Drăgoi[1], Alexandru Militaru[3], and Josef Widder[2]

[1] INRIA, ENS, CNRS, PSL
[2] TU Wien
[3] Politehnica University Bucharest

Abstract. Fault-tolerant distributed systems are implemented over asynchronous networks, so that they use algorithms for asynchronous models with faults. Due to asynchronous communication and the occurrence of faults (e.g., process crashes or the network dropping messages) the implementations are hard to understand and analyze. In contrast, synchronous computation models simplify design and reasoning. In this paper, we bridge the gap between these two worlds. For a class of asynchronous protocols, we introduce a procedure that, given an asynchronous protocol, soundly computes its round-based synchronous counterpart. This class is defined by properties of the sequential code.
We computed the synchronous counterpart of known consensus and leader election protocols, such as, Paxos, and Chandra and Toueg's consensus. Using Verifast we checked the sequential properties required by the rewriting. We verified the round-based synchronous counter-part of Multi-Paxos, and other algorithms, using existing deductive verification methods for synchronous protocols.

1 Introduction

Fault-tolerant distributed systems provide a dependable service on top of unreliable computers and networks. They implement fault tolerance protocols that replicate the system is replicated and ensure that from the outside all (unreliable) replicas are perceived as a single reliable one. This has been formalized under terms like strong consistency, consensus, state machine replication, and Paxos. These protocols are crucial parts of many distributed systems and their correctness is very hard to obtain. Protocol designers are faced with the challenges of buffered message queues, message re-ordering at the network, message loss, asynchrony and concurrency, and process faults. Reasoning about all these features is a notoriously hard as discussed in several research papers [6,36,22,33,38], and as a consequence testing tools like Jepsen [24] found conceptual design bugs in deployed implementations.

Problem statement. A programming abstraction of synchronous rounds [16,30,10] would relieve the designer from many of these difficulties. Synchronous round-based algorithms are more structured, are easier to understand, have simpler behaviors. As one only has to reason about specific global states at the round boundaries, they entail simpler correctness arguments. However, it is also well-understood that synchronous distributed systems, are often "impossible or inefficient to implement" [30, p. 5]. Hence, designers turn to the asynchronous model, in which the performance emerges [28] from the current load of a system, which in normal operation has significant better performance. Thus, no synchronous algorithm is used in any real large scale system we are aware of.

In face of the different advantages of synchronous and asynchronous models, the question is how to connect these two worlds. We consider the question, given an asynchronously algorithm, does it have a synchronous canonic counterpart?

Key Challenges. The main difficulty stems from the fundamental discrepancy in the control structure, that is, *(i)* interleavings and *(ii)* message buffers:
(i) Interleavings: In synchronous round-based models, computation is structured in rounds that are executed by all process in lock-step. There is no interleaving between steps, the beginning and the end of each step is synchronized across processes, by definition. In the asynchronous computational model executions are much less structured. Processes are scheduled according to interleaving semantics. This leads to an exponential number of intermediate global states (exponential in the number of steps) vs. a linear one in the synchronous case.
(ii) Message buffers: In the synchronous model, messages are received in the same round they are sent. Thus, the number of messages that are in-transit is bounded at all times, and depends on the number of processes. In the asynchronous model, fast processes may generate messages quicker than slow processes may process them. Thus communication needs to be buffered, and the buffer size is unbounded. The number of messages that are in a buffer depends on the number of processes, but also on the number of send instructions executed by each process. Moreover, the network may reorder messages, that is, a process may receive a new message before all older ones, that are still in-transit.

Due to the discrepancy, there is no obvious reason why an asynchronous algorithm should have a synchronous canonic form. In general there is none. We characterize asynchronous systems that allow to dissolve this discrepancy.

Key Insights. As not all conceivable asynchronous protocols can be rewritten into synchronous ones, we focus on characteristics of practical distributed systems. From a high-level viewpoint, distributed systems are about coordination in the absence of a global clock. Thus, distributed algorithms implement an abstract notion of time to coordinate. This notion of time may be implicit. However, the local state of a process maintains this abstract time notion, and a process timestamps the messages it sends accordingly. Synchronous algorithms do not need to implement an abstract notion of time, as it is present from the beginning: the round number plays this role and it is embedded in the definition of any synchronous computational model. The key insight of our results is the existence of a correspondence between values of the abstract clock in the asynchronous systems and round numbers in the synchronous ones. Using this correspondence, we make explicit the "hidden" round-based synchronous structure of an asynchronous algorithm. More systematically, in an asynchronous system:

- abstract time is encoded in local variables. Modifications of their values mark the progress of abstract time, and — making the correspondence to round-based algorithms — the local beginning/ending of a round;
- a global round consists of all the steps processes execute for the same value of the abstract clock;
- messages are timestamped with the value of the abstract time of the sender, when the message was sent; a receiver can read the abstract time at which the message was sent, and compare it with its own abstract time;
- in order to have a faithful correspondence to round-based semantics, we consider *communication-closed protocols*: the reception of a message is effectful only if its timestamp is equal to or greater than the local time of the receiver. In other words, stale messages are discarded.

Based on these insights (i) we characterize asynchronous protocols whose executions can be reduced to well-formed canonic executions, (ii) we define a computational model CompHO whose executions are all canonic by definition, (iii) we show how to translate an asynchronous protocol into code for the CompHO framework, and (iv) we show the benefits of this computed canonic form for design, validation, and verification of distributed systems. We discuss these four points by using a running example in the following section.

2 Our approach at a glance

The running example in Fig 1 is inspired by typical fault-tolerant distributed protocols that often rely on the notion of leadership. For example, primary backup algorithms use a leader to order the client requests and to ensure this order among all replicas. A leader is a process that is connected via timely links to a majority of replicas. Hence, only in ballots where such a well-connected leader exists, the system should try to make progress. The algorithm in Fig. 1 implements just the leader election algorithm. All processes execute the same code, and n is the number of processes. In each loop iteration each process queries its `coord` oracle in line 22 to check whether it is a leader candidate. Multiple processes may be candidates in the same iteration. Depending on the outcome, the code then branches to a leader branch and a follower branch. A candidate process sends to all in line 27. Then, the leader branch has the same code as the follower branch, that is, waiting for the first message by a candidate in the loop starting at lines 29 and 55. Thus, if there are multiple candidates there is a race between them. If a message from a candidate for a current of future ballot is received, processes update their ballot in line 34 and 59, and then set their leader estimate in the next line. This estimate is then sent to all in line 41 and 63, and then processes wait to receive messages from a majority ($n/2$), for the current ballot. If all $n/2$ received messages carry the same leader identity, then a process knows a leader is elected in the current ballot, and it records the leader's identity in line 50 and 72. From a more structural viewpoint, because this protocol is supposed to be fault-tolerant, the receive statements in, e.g., line 29 and 43 are non-blocking and may receive `NULL` if no message is there.

```
1  typedef struct Msg { int lab; int ballot;
        int sender;} msg;
2  typedef struct List{ msg *message;  struct
        List * next; int size;} list;
3  enum labels {NewBallot, AckBallot} ;
4  int coord(){return pid }
5  bool filter(msg *m,int b,enum labels l){
6    if (m!=0 && l==NewBallot) return
          (m->ballot>=b&& m->label==l);
7    if (m!=0 && l==AckBallot) return
          (m->ballot==b && m->label==l);
8    return false; }
9  bool all_same(list *mbox){
10 list *x=mbox;
11 if (x!=NULL) val = x->message->sender;
12 while(x!=NULL) { if(x->message->sender !=
        val) return false; else x= x->next;}
13 return true;}
14 int main(){
15 int  me=getId(); int n;
16 struct arraylist *log_epoch, *log_leader;
17 int ballot = 0; enum labels label;
18 list *mbox = NULL; list_create(log_epoch);
19 list_create(log_leader);
20 while(true){
21  label = NewBallot;
22  if (coord() == me){//LEADER'S CODE
23    ballot++;
24    //@ assert tag_leq(oldballot, oldlabel,
           ballot, label);
25    msg *m = create_msg(ballot,label,myid);
26    //@ assert tag_eq(m->ballot,m->label,
           ballot,label);
27    send(m,*);  //send new ballot to all
28    reset_timeout();
29    while(true ){  (m,p) = recv();
30     if (filter(m,ballot,label)) add(mbox, m);
31     if((mbox!=0 && mbox->size==1
            )||timeout())break;}
32 //@ assert mbox_geq(ballot,label,mbox);
33    if (mbox!=0 && mbox->size ==1) {
34         ballot = mbox->message->ballot;
35      //@assert max_tag(ballot,label,mbox);
36        leader = mbox->message->sender;
37 dispose(mbox); label = AckBallot;
38 //@ assert tag_leg(oldballot, oldlabel,
        ballot,label);
39 msg *m = create_msg(ballot,label,myid);
40 //@ assert tag_eq(m->ballot, m->label,
        ballot,label);
41 send(m,*);  //send ack ballot to all
42 reset_timeout(); //wait for AckBallot
43 while  (true){ (m,p) = recv();
44     if (filter(m,ballot,label)) add(mbox, m);
45     if((mbox!=0 && mbox->size>n/2 ) break;
46     if(timeout()) break;}
47 //@ assert mbox_tag_eq(ballot,label,mbox);
48 if (mbox!=0 && mbox->size > n/2&&
        all_same(mbox)){
49     list_add(log_ballot,ballot, true);
50     list_add(log_leader,ballot, leader);
51     out(ballot, leader);}
52 }else{ dispose(mbox); }
53 }else{//FOLLOWER'S CODE
54   ballot++; reset_timeout(); //wait Newballot
55   while(true ){  (m,p) = recv();
56      if (filter(m,ballot,label)) add(mbox, m);
57      if((mbox!=0 && mbox->size==1
             )||timeout())break;}
58   if (mbox!=0 && mbox->size ==1) {
59     ballot = mbox->message->ballot;
60     leader = mbox->message->sender; {
61     dispose(mbox); label = Ackballot;
62     msg *m = create_msg(ballot, label,
           leader);
63     send(m,*);  //send ack ballot to all
64     reset_timeout(); //wait for Ackballot
65     while(true){ (m,p) = recv();
66       if (filter(m,ballot,label))add(mbox, m);
67       if(mbox!=0 && mbox->size>n/2)break;
68       if(timeout())break;}
69      if(!timeout() && all_same(mbox)){
70          list_add(log_ballot,ballot, true);
71          list_add(log_leader,ballot, leader);
72          out(ballot, leader); }
73       }else{ dispose(mbox); }
74 }}} //END FOLLOWER END while END main
```

Fig. 1: Paxos-like (Phase 1) asynchronous leader election protocol.

Consider the asynchronous execution on the left of Figure 2. Process P1 always takes the follower branch, P3 is a candidate in ballot 1, but its messages get delayed. So process P2 times out in ballot 1 in line 57 and 68, and becomes a candidate in ballot 2. Its message reaches P1, which jumps to ballot 2 and sends its leader estimate to all in line 63, as does P2. However, only P1 receives all these message so that it gets over the $n/2$ threshold to set its leader in line 72. Messages marked with a cross are dropped by the network. As the messages by P1 arrive late, i.e., the receiver's local time passed the timestamp of the message, the messages sent by P1 become stale and are disregarded by P2 and P3.

As a result, the late messages sent by P1 have the same effect as if they were dropped by the network. In this view, the execution of the right on Figure 2 is obtained from the one on the left by a "rubber band transformation" [32], that is,

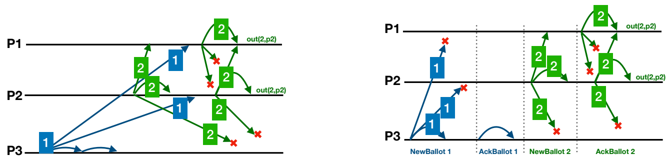

Fig. 2: Asynchronous execution with jumps on the left and corresponding "synchronous" execution on the right

transitions are reordered by maintaining to local control flow, and the causality imposed by sending and receiving a message. We call the execution on the right the canonic form of the execution on the left.

Characterization of existence of a canonic form. Let us understand whether each execution of the asynchronous protocol from the example can be brought into a canonic form. The first observation is that the variables `ballot` and `label` encode abstract time. Let b and ℓ be evaluations of the variables `ballot` and `label`. Then abstract time ranges over $T = \{(b, \ell) \colon b \in \mathbb{N}, \ell \in \{\texttt{NewBallot}, \texttt{AckBallot}\}\}$. We fix `NewBallot` to be less than `AckBallot`, and consider the lexicographical order over T. Then we observe that the sequence of (b, ℓ) induced by an execution at a process is monotonically increasing; thus (b, ℓ) encodes a notion of time. However, a locally monotonic ascending sequence of values is not sufficient to derive a global notion of time, i.e., a globally aligned ascending sequence of values, as in Figure 2 on the right. Technically, aligning means that we need a reduction argument where (i) we tag an event with the local time of the process at which it occurs, and (ii) if in an execution a transition t tagged with (b, ℓ) happens before a transition t' tagged with (b', ℓ') at another process, with $(b', \ell') < (b, \ell)$, then swapping these two transition should again result in an execution. That is, t and t' should commute. This condition is satisfied when stale messages are discarded. In other words, it is ensured if the protocol is communication-closed: first, each process only sends for the current timestamp, e.g., the send statement in line 41 sends a message that carries the current (`ballot`, `label`) pair. Second, each process receives only for the current or a higher timestamp, e.g., received messages are stored, e.g., in line 30, only if they carry the current or a future (`ballot`, `label`) pair; cf. line 5.

We introduce a tag annotation in which the programmer can provide us with the variables and parts of the messages that are supposed to encode abstract time and timestamps. Using these tags, the protocol is annotated with verification conditions stating that (1) abstract time is monotonically increasing (line 24), which use the predicate *tag_leq* in Fig. 10 (page 22), (2) each process sends only for the current timestamp (line 26), and (3) each process receives messages from the current or a higher timestamp (line 32). Given an annotated asynchronous protocol, we check the validity of these assertions using the static verifier Verifast [23]. Using Verifast was extremely useful to prove the conditions on the content of the mailbox, which is an unbounded list (lines 2 and 47). For example, the `assert` at line 47 states that all messages in the mailbox have their

```
typedef struct Msg {int lab; int ballot;
      int sender;} msg;
typedef struct List{ msg *message;
      struct List * next; int size;}
      list;
int coord(){return pid }
bool all_same(list *mbox){
  list *x=mbox;
  if (x!=NULL) val = x->message->sender;
  while(x!=NULL) { if(x->message->sender
        != val) return false; else x=
        x->next;}
 return true;}
int phase(){return round/phase.length}
int coord();
void init(){
    me = getMyId();
    old_mbox1 = false;
    list_create(log_epoch);
    list_create(log_leader);
}
```

```
phase = array[NewBallot; AckBallot]
round NewBallot:
send{
     if(coord()==me) send(pid,*); }
update(list * mbox){
    if (mbox!= 0 && mbox->size ==1) {
        leader = mbox->message->sender;
        old_mbox1 = true;}
round AckEpoch:
send{
        if (old_mbox1) send(leader,*);}
update(list* mbox){
  if(old_mbox1 == true && mbox!=0 &&
        mbox->size >n/2 &&
        all_same(mbox)) {
      out(phase(),leader);
      list_add(log_epoch,phase(),true);
      list_add(log_leader,phase(),leader);
      old_mbox1= false;
      }
```

Fig. 3: Synchronous Paxos-like leader election in CompHO.

ballot and label fields equal with the local variable `ballot` and `label`. The other predicates are given Fig. 10.

If these checks are successful, the existence of a canonical form is guaranteed by a new reduction theorem proven in Section 6. Our reduction uses ideas from [17] where the notion of communication closure was introduced in CSP. Our notion of communication closure is more permissive than the original form, as we allow to react to a message that is timestamped with higher value (e', ℓ') than the current local abstract time (e, ℓ), provided that the code immediately "jumps forward in time" to (e', ℓ'). This corresponds in our example to P1 jumping to ballot 2 upon reception of the message by P2.

In contrast to Lipton's reduction [29], where one proves that actions in an execution can be moved in order to get a similar execution with large atomic blocks of local code, for distributed algorithms one proves that one can group together the (e, ℓ) send transitions of all processes, then the (e, ℓ) receive transitions of all processes, and then all (e, ℓ) computation steps, for all times (e, ℓ) in increasing order. In this way, we formally establish that the asynchronous execution from the left of Figure 2 corresponds to the so-called round-based execution on the right. Executions of this form we call *canonic*.

A computational model for canonic executions. Since we can reduce an asynchronous execution to a canonic one, our goal is to re-write an asynchronous protocol into a program with round-based semantics. For this we first need to establish a programming model for canonic (round-based) protocols. Several *round models* exist in the literature [16,19,39,10]. We adapt ideas from these models for our needs, and introduce our new CompHO model. It allows us to express a more fine grained modeling of faults, network timing, and sub-routines. The closest model from the literature is the Heard-Of Model [10] that CompHO ex-

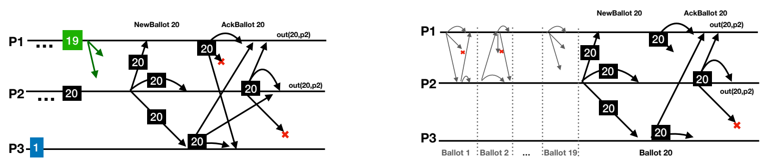

Fig. 4: Execution with jumps

tends to multi-shot algorithms (multiple inputs received during the executions) and has a compositional semantics based on synchronized distributed procedure calls. Figure 3 shows the CompHO program obtained from Figure 1.

The interesting feature that abstracts away faults and timeouts are the so-called *HO* sets. For each round (b, ℓ) and each process p, the set $HO(p,(b,\ell))$ contains the set of processes from which p hears of in that round, i.e., whose messages show up in `mbox` in, e.g., line 22. For instance, in the simplest case, if a message from a process q is lost, it just does not appear in the *HO* set. But also the forward jumping is covered. Consider the execution on the left of Figure 4. While the processes P1 and P2 made progress, P3 was disconnected from them. Then, while locally still being in ballot 1, P3 receives a message for ballot 20. The execution on the right is an execution in CompHO and the jump by Process P3 is captured by $HO(P3,(i,j)) = \emptyset$, for $(1, \texttt{NewBallot}) \leq (i,j) < (20, \texttt{NewBallot})$. For all the skipped rounds, `mbox` evaluates to the empty list.

We have augmented the Heard-Of Model with in() and out() primitives for multi-shot algorithms such as state machine replication where the system gets commands from an external client and should output results.

Computing the canonic form. Having defined the round-based semantics of CompHO, we introduce a rewriting procedure. It takes as input the asynchronous protocol together with the annotations that have been checked to entail a canonic form, and produce as output the protocol rewritten in CompHO.

The main challenges for the rewriting come from the different possible control flows of the programs and their relation to the abstract notion of time. Due to branching (e.g., in line 22), code that appears in different places may be required to be composed into code for the same round; e.g., line 49 on the leader branch belongs to the same round as line 70 on the follower branch and will end up in line 32 of the canonic form. (More precisely, the generated code has branching within a round and the statement of line 32 appears in both branches. We simplified the code given in Figure 3 for better readability.) In addition, there are jumps in the local time; cf. Figure 4. This corresponds in the CompHO to phases and rounds that are skipped over by a process, that is, it neither receives nor sends messages, and maintains (stutters) its local state. The asynchronous statements before and after a jump must be properly mapped to rounds in CompHO. We address these issues in Section 7 and have implemented our solution. We used it to automatically generate CompHO code for several asynchronous protocols.

Benefits of a round-based synchronous normal form. The generated CompHO code represents a valuable design artifact. First, a designer can check whether the implementation meets the original intuition. For instance, the left execution in Figure 4 gives a typical asynchronous execution. In papers on systems, designers explain their systems with well-formed executions like the one on the right. The designer can check with the CompHO code whether the asynchronous protocol implements the intended ballot and round structure, and whether phase jumps can occur only at intended places. Second, it helps in comparing protocols: Different ways to implement the branching due to roles (e.g., leader, follower) leads to different asynchronous protocols. If different asynchronous protocols have the same canonic form, then they encode the same distributed algorithm.

Finally, the canonic form paves the way for automated verification. The specification of the running example (in the asynchronous and synchronous version) is that in any ballot b, if two processes p_1 and p_2 find that leader election was successful (i.e., their `log_ballot` entry is true), then they agree on the leader:

$$\forall p_1, p_2,\ \forall b\ (\texttt{log\_ballot}[p_1][b] = \mathit{true} \wedge \texttt{log\_ballot}[p_2][b] = \mathit{true}) \Rightarrow \texttt{log\_leader}[p_1][b] = \texttt{log\_leader}[p_2][b] \quad (1)$$

To prove this property in the asynchronous model, already for the first ballot, i.e., $b = 1$, one needs to introduce auxiliary variables to be able to state an inductive invariant. A process elects a leader if it receives more than $n/2$ messages having the same *leader id* as payload, so one needs to reason about the set of messages they received. As discussed in [40], this can only be achieved by introducing auxiliary variables that record the complete history of the message pool. Then one can state an invariant over the asynchronous execution that relates the local state of processes (decided or not) with the message pool history.

The proof of the same property for the synchronous protocol requires no such invariant. Due to communication closure, no messages need to be maintained after a round terminated, that is, there is no message pool. One just needs to consider the transition relation of a phase, or ballot (conjunction of two rounds). The global state after the transition, that is, at the end of the phase, captures exactly which processes elected a leader in the considered phase.

In general, to prove the specification, we need invariants that quantify over the ballot number b. As processes decide asynchronously, the proof of ballot 1 for some process p must refer to the first entry of `log_ballot` of processes that might already be in ballot 400. Thus, the invariants need to capture the complete message history and the complete local state of processes. The proof in the synchronous case is modular: For any two phases, messages do not interfere and processes write to different ballot/phase entries. Therefore the agreement proof for one ballot generalizes for all ballots.

Many verification techniques benefit from a reduction from asynchronous to synchronous. In particular, the model checking techniques in [31,44] are desingned specifically for the Heard-Of model [10,4], and can be applied on our output. Theorem provers like Isabelle/HOL where successfully used to prove total correctness of algorithms in the Heard-Of model [9]. We used deductive verification

e :=	expression	S :=	statement
$\mid$ c	constant	$\mid$ S ; S	sequence
$\mid$ x	variable	$\mid$ x := e	assignment
$\mid$ f($\overrightarrow{e}$)	operation	$\mid$ reset_timeout(e)	reset a timeout
		$\mid$ send(m,p) $\mid$ send(m,*)	send message
types := Pid	process Id	$\mid$ (m,p) := recv()	receive message
$\mathbb{T}$	user defined	$\mid$ if e then S else S	
	or primitive type	$\mid$ while true S	
$\mathbb{MT}$	payload type	$\mid$ break	
p : Pid,	m : $\mathbb{MT}$	$\mid$ continue	
Mbox:	set of ($\mathbb{MT}$, Pid)	$\mid$ x = in()	client entry
P := $\Pi_{p\in\mathcal{P}id}[S]_p$	protocol	$\mid$ out(e)	client output

Fig. 5: Syntax of asynchronous protocols.

methods for the Heard-Of model [13] and proved the (partial) correctness of the synchronous version of the running example (and other protocols).

3 Asynchronous protocols

Protocols are written in the core language in Fig 5. All processes execute the same sequential code, which is enriched with send, receive, and timeout statements.

The communication between processes is done via typed messages. Message payloads, denoted $\mathbb{MT}$, are wrappers of primitive or composite type. Wrappers are used to distinguish payload types from the types of the other program variables. Send instructions take as input an object of some payload type and the receivers identity or $*$ corresponding to a send to all (broadcast). Receive statements return an object of payload type and the identity of the sender, that is, one message is received at a time. Receives are not blocking. If no message is available, receive returns $\bot$. We assume that each loop contains at least one send or receive statement. The iterative sequential computations are done in local functions, i.e., `f(`$\overrightarrow{e}$`)`. The instructions `in()` and `out()` are used to communicate with an external environment (processes not running the protocol).

The semantics of a program $\mathcal{P}$ is the asynchronous parallel composition of the actions performed by all processes. Formally, the state of a protocol $\mathcal{P}$ is a tuple $\langle s, msg\rangle$ where: $s \in [P \rightarrow \texttt{Vars} \cup \texttt{Loc} \rightarrow \mathcal{D}]$ is a valuation of the variables in $\mathcal{P}$ where the program location is added to the local state and $msg \in [\mathbb{MT} \rightarrow (P, \texttt{T}, P, \mathbb{N}) \rightarrow \mathbb{N}]$ is the multiset of messages in transit (the network may lose and duplicate messages). Given a process p $\in$ P , s(p) is the local state of p, which is a valuation of p's local variables, i.e., s(p) $\in$ [Vp $\rightarrow$ D]. We use a special value $\bot$ to represent the state of crashed processes. When comparing local states, $\bot$ is treated as a wildcard state that matches any state.

The messages sent by a process are added to the global pool of messages msg, and a receive statement removes a messages from the pool. The interface operations `in` and `out` do not modify the local state of a process. These are the only statements that generate observable events.

An execution is an infinite sequence $s0\ A0\ s1\ A1 \ldots$ such that $\forall i \geq 0$, si is a protocol state, $Ai \in A$ is a local statement and $(si \xrightarrow{Ai} si + 1)$ is a transition of the form $\langle s, msg\rangle \xrightarrow{I,O} \langle s', msg'\rangle$ corresponding to the execution of Ai, where $\{I, O\}$ are the observable events generated by the Ai (if any). We denote by $[\![\mathcal{P}]\!]$ the set of executions of the protocol $\mathcal{P}$.

4 Round-based model

We introduce CompHO by first presenting the syntax and semantics of the intra-procedural version of CompHO, extending it then to inter-procedural case.

Intra-procedural CompHO Model. CompHO captures round-based distributed algorithms: all processes execute the same code and the computation is structured in rounds, where the round number is an abstract notion of time: processes are in the same round, and progress to the next round simultaneously. We denote by P the set of processes and $n = |P|$ is a parameter. Faults and timeouts are modeled by messages not being received. In this way the central concept is the Heard-Of set, HO-set for short, where $HO(p, r)$ contains the processes from which process p has *heard of* — has received messages from — in round r.

Syntax. A CompHO protocol is composed of local variables, an initialization operation `init`, and a non-empty sequence of rounds, called phase. The syntax is given in Fig. 6. A round is an object with a send and update method, and the phase is a fixed-size array of rounds. Each round is parameterized by a type T (denoted by $round_{\mathsf{T}}$) which represents the payload of the messages. The `send` function has no side effects and returns the messages to be sent, a partial map from receivers to payloads, based on the local state of each sender. The `update` function, takes as input the received messages, i.e., a partial map from senders to payloads, and updates the local state of a process. It may communicate with an external client via `in`, which returns an input value, and `out` which outputs a a value to the client. For data computations, `update` uses iterative control structure only indirectly via auxiliary functions, like `all_same` in the running example, whose definition we omit.

$$
\begin{array}{rl}
\textit{protocol} ::= & \textit{interface}\ \textit{variable}^{*}\ \textit{init}\ \textit{phase} \\
\textit{interface} ::= & \texttt{in}\colon () \to \textit{type} \mid \texttt{out}\colon \textit{type} \to () \\
\textit{variable} ::= & \textit{name}\colon \textit{type} \\
\textit{init} ::= & \texttt{init}\colon () \to [P \to V \to \mathcal{D}] \\
\textit{phase} ::= & \textit{round}^{+} \\
\textit{round}_{\mathsf{T}} ::= & \texttt{send}\colon [P \to V] \to [P \rightharpoonup \mathsf{T}] \\
 & \texttt{update}\colon [P \rightharpoonup \mathsf{T}] \times [P \to V] \\
 & \to [P \to V]
\end{array}
$$

Fig. 6: CompHO syntax.

Semantics. The set of executions of a CompHO protocol is defined by the execution of the `send` and `update` functions of the rounds in the phase array in a loop, starting from the initial configuration defined by `init`.

A protocol state is a tuple $\langle SU, s, r, msg, P, HO\rangle$ where:

– P is the set of processes executing the protocol;
– $SU \in \{Snd, Updt\}$ indicates if the next operation is send or update;

START

$$\frac{\xrightarrow{\texttt{init()}} s(p)}{* \xrightarrow{\emptyset, \{\texttt{init}_p() | p \in P\}} \langle Snd, s, 0, \emptyset, P, HO \rangle}$$

SEND

$$\frac{\forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r]\texttt{.send}(m_p)} s(p) \qquad msg = \begin{array}{c}\{(p, t, q) \mid \\ p \in P \wedge (t, q) \in m_p\}\end{array}}{\langle Snd, s, r, \emptyset, P, HO \rangle \xrightarrow[p \in P]{\{\texttt{send}_p(m_p)\}, \emptyset} \langle Updt, s, r, msg, P, HO' \rangle}$$

UPDATE

$$\frac{\begin{array}{c}\forall p \in P.\ mbox_p = \{(q, t) \mid (q, t, p) \in msg \wedge q \in HO(p)\} \\ \forall p \in P.\ s(p) \xrightarrow{\texttt{phase}[r]\texttt{.update}(mbox_p)), o_p} s'(p) \qquad r' = r + 1 \qquad O = \{o_p \mid p \in P\}\end{array}}{\langle Updt, s, r, msg, P, HO \rangle \xrightarrow{\{\texttt{update}_p(mbox_p) | p \in P\}, O} \langle Snd, s', r', \emptyset, P, HO \rangle}$$

Fig. 7: CompHO semantics.

– $s \in [P \to V \to \mathcal{D}]$ stores the process local states;
– $r \in \mathbb{N}$ is the round number, i.e., the counter for the executed rounds;
– $msg \subseteq 2^{P, \mathsf{T}, P}$ stores the in-transit messages, where T is the type of the message payload;
– $HO \in [P \to 2^P]$ evaluates the HO-sets for the current round.

The semantics is shown in Figure 7. Initially the system state is undefined, denoted by $*$. The first transition calls the `init` operation on all processes (see START in Fig. 7), initializing the state: The round is 0, no messages are in the system. START brings the system into a *Snd* state that requires the next transition to be a SEND. After that, an execution alternates SEND and UPDATE transitions. In the SEND step, all processes send messages, which are added to a pool of messages *msg*, without modifying the local states. The values of the *HO* sets are updated non-deterministically to be a subset of P. The messages in *msg* are triples of the form (sender, payload, recipient), where the sender and receiver are processes and the payload has type `T`. The triples are obtained from the map returned by `send` to which we add the identity of the process that executed `send`. In an UPDATE step, messages are received and the `update` operation is applied in each process. A message is lost if the sender's identity does not belong to the *HO* set of the receiver. The set of received messages is the input of `update`. If the processes communicate with an external process, then `update` might produce observable events o_p. These events correspond to calls to `in`, which returns an input value, and `out` that sends the value given as parameter to the client. The communication with external processes is non-blocking; we assume that the function `in` always returns a value when called. At the end of the round, *msg* is purged and r is incremented by 1.

Example 1. The right diagram of Fig. 2 corresponds to an executions of the CompHO protocol in Fig. 3. The SEND step of round `AckEpoch` consists of process P3 sending in line 21, and the environment dropping its messages to P1 and P2. As they do not receive messages, UPDATE does not result in a state change due to line 23. Hence `old_mbox1` does no change so that the guard in line 23 evaluates to false at P2 and P3, so that they do not send in the `AckEpoch` round.

Inter-procedural CompHO Model. We introduce *distributed procedure calls* to capture realistic examples. In Multi-Paxos [27] processes agree on a order over client commands. This order is stored in a local log, that contains the commands received/committed so far. Consider Figure 8. Here a new leader gets elected with a `NewBallot` and `AckBallot` message exchange, almost as in our example in Section 2. The difference is the `AckBallot` round where followers (1) send only to the leader instead of an all to all communication, (2) the message payload contains the current log of the follower. Then the leader computes the longest log and sends it to its followers, in the third round called `NewLog`. Those that receive the new log start a subprotocol. The subprotocol iterates through an unbounded number of phases each consisting of a sequence of rounds, `Prepare`, `PrepareOK` and `Commit`, in which the replicas put commands in their logs. Iteratively, the leader takes a new input command from the client and forwards it to the replicas using a `Prepare` message. Followers reply with `PrepareOK` acknowledging the reception of the new command. If the leader receives $n/2$ acknowledgements it sends a `Commit` message, otherwise it considers its quorum lost and returns to leader election. A follower that does not receive a message from the leader, considers the leader crashed, and control returns from the subprotocol to the leader election protocol.

We only sketch the model. The inter-procedural CompHO protocol differs from its intra-procedural version only in the `update` function. A process may call another protocol and block until the call to this other protocol returns. An `update` may call at most one protocol on each path in its control flow (a sequence of calls can be implemented using multiple rounds). Due to branching, only a subset of the processes may make a call in a round. Thus, an inter-procedural CompHO protocol is a collection of (inter/intra-procedural) non-recursive CompHO protocols, that call each other, with a main protocol as entry point.

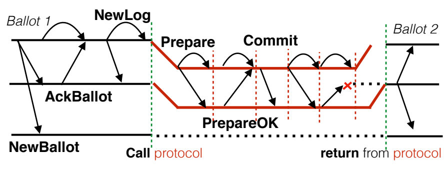

Fig. 8: Inter-procedural execution

5 Formalizing Communication Closure using Tags

We now introduce tags that use so-called synchronization variables to annotate protocols. A tagging function induces a local decomposition of any execution, where a new block starts whenever the evaluation of the synchronization variables changes. (Recall the set T for our example in Section 2.) This tagging thus represents a novel formalization of *communication-closed* protocols using syntactic definitions of local decompositions.

Definition 1 (Tag annotation). *For a protocol $\mathcal{P}$, a* tag annotation *is a tuple* $(\mathtt{SyncV}, \mathtt{tags}, \mathtt{tagm})$*:*

– $\texttt{SyncV} = (v_1, v_2, \dots, v_{|\texttt{SyncV}|})$ *is a tuple of fresh variables,*
– $\texttt{tags} : \texttt{Loc} \rightarrow [\texttt{SyncV} \overset{injective}{\rightharpoonup} \texttt{Vars}]$*, is a function that annotates each control location with a partially defined injective function, that maps* SyncV *over protocol variables, and*
– $\texttt{tagm} : \mathbb{MT} \rightarrow [\texttt{SyncV} \overset{injective}{\rightharpoonup} \mathsf{T}]$ *is an injective partially defined function, that maps variables in* SyncV *to components of the message type* T *(of the same type).*

The evaluation of a tag over $\mathcal{P}$*'s semantics is denoted* $(\llbracket\texttt{tags}\rrbracket, \llbracket\texttt{tagm}\rrbracket)$*, where*

– $\llbracket\texttt{tags}\rrbracket : \Sigma \rightarrow [\texttt{SyncV} \rightarrow \mathcal{D} \cup \bot]$*, is a function over the set of local process states,* $\Sigma = \bigcup_{s \in \llbracket\mathcal{P}\rrbracket} \bigcup_{p \in P} s(p)$*, defined by* $\llbracket\texttt{tags}\rrbracket_s = (d_1, \dots, d_{|\texttt{SyncV}|})$*, with*
  - $d_i = \llbracket \mathsf{x_i} \rrbracket_s$ *if* $\mathsf{x_i} = \texttt{tags}(\llbracket\mathsf{pc}\rrbracket_s)(v_i) \in \texttt{Vars}$*, where* v_i *is the* i^{th} *variable in* SyncV *and* pc *is the program counter,*
  - *otherwise* $d_i = \bot$.
– $\llbracket\texttt{tagm}\rrbracket : \mathbb{MT} \rightarrow \mathcal{T} \rightarrow [\texttt{SyncV} \rightarrow \mathcal{D} \cup \bot]$ *is a function that for any value* $m = (m_1, \dots, m_t)$ *of message type* T *associates a tuple* $\llbracket\texttt{tagm}\rrbracket_{m:\mathsf{T}} = (d_1, \dots, d_{|\texttt{SyncV}|})$ *with*
  - $d_i = m_j$ *if* $m_j = \texttt{tagm}(\mathsf{T})(v_i)$*, where* v_i *is the* i^{th} *variable in* SyncV *and* $\texttt{tagm}(\mathsf{T})$*, the mapping of* SyncV *over the message type* T*, is defined in* v_i*;*
  - $d_i = \bot$*, otherwise.*

Example 2. For the protocol in Fig. 1, we consider the tag annotation over two variables (v_1, v_2) that at all control locations associates v_1 with the ballot number, and v_2 with `label`. The first two components of messages of type (`int`, `enum`, `int`) are mapped to (v_1, v_2). A message $m = (3, \texttt{NewBallot}, 5)$ (sent in line 41) is evaluated by `tagm` into $(3, \texttt{NewBallot})$. The state tag evaluates into $(3, \texttt{NewBallot})$ if the value of the variable `ballot` is 3.

We characterize tag annotations that imply communication closure:

Definition 2 (Synchronization tag). *Given a program* $\mathcal{P}$*, an annotation tag* (SyncV, tags, tagm) *is called* synchronization tag *iff:*

(I.) for any local execution $\pi = s_0 A_0 s_1 A_1 \ldots \in \llbracket\mathcal{P}\rrbracket_p$ *of a process* p *in the semantics of* $\mathcal{P}$*,* $\llbracket\texttt{tags}\rrbracket_{s_0}\llbracket\texttt{tags}\rrbracket_{s_1}\llbracket\texttt{tags}\rrbracket_{s_2}\ldots$ *is a monotonically increasing sequence of tuples of values w.r.t. the lexicographic order.*

(II.) for any local execution $\pi \in \llbracket\mathcal{P}\rrbracket_p$*, if* $s \xrightarrow{send(m,p)} s'$ *is a transition of* π*, with* m *a value of some message type, then* $\llbracket\texttt{tags}\rrbracket_s = \llbracket\texttt{tagm}\rrbracket_m$ *and* $\llbracket\texttt{tags}\rrbracket_s = \llbracket\texttt{tags}\rrbracket_{s'}$.

(III.) for any local execution $\pi \in \llbracket\mathcal{P}\rrbracket_p$*, if* $s \xrightarrow{(m,p)=recv(cond)} sr$ *is a transition of* π*, with* m *a value of some message type, then*
  – *if* $(m,p) \neq \texttt{null}$ *then*
    - $\llbracket\texttt{tags}\rrbracket_s \leq \llbracket\texttt{tagm}\rrbracket_m$ *if* $\texttt{tagm}(T)$ *is surjective.* *Moreover,* $\{\{\llbracket\texttt{tagm}\rrbracket_m \mid m \in \llbracket\mathsf{Mbox}\rrbracket_s\} \subseteq \{\llbracket\texttt{tags}\rrbracket_s, \llbracket\texttt{tagm}\rrbracket_m\}$.
    - $\llbracket\texttt{tags}\rrbracket_s = \llbracket\texttt{tagm}\rrbracket_m$*, otherwise. Also* $\llbracket\texttt{tags}\rrbracket_s = \llbracket\texttt{tags}\rrbracket_{sr}$.

   – *if* $(m,p) = \texttt{null}$ *then* $s = sr$.

(IV.) for any local execution $\pi \in [\![\mathcal{P}]\!]_p$, *if* $s \overset{\mathsf{stm}}{\rightarrow} s'$ *is a transition of* π *such that*
   – $s \neq s'$, $s \mid_{\mathbb{MT},\texttt{SyncV}} = s' \mid_{\mathbb{MT},\texttt{SyncV}}$, *that is, s and s' differ on the variables that are neither of some message type nor synchronization variables,*
   – *or* $\texttt{stm}$ *is a* send, break, continue, or out(),

then $[\![\texttt{tags}]\!]_s = max\{[\![\texttt{tagm}]\!]_m \mid m \in [\![\mathsf{Mbox}]\!]_s\}$, *for all* Mbox:$\mathsf{Set(T)}$, $\mathsf{T} \in \mathbb{MT}$, *with* $[\![\mathsf{Mbox}]\!]_s \neq \emptyset$. *That is, observable state changes and sends happen only if the state tag matches the maximal received message tag.*

If an annotation tag is a synchronization tag, the variables that annotated the protocol are called synchronization variables.

Condition (I.) states that the variables incarnating the abstract time are not decreased by any local statement. Condition (II.) states that any message sent is tagged with a timestamp that equals the local time of the current state. Condition (III.) states that any message received and stored is tagged with a timestamp greater or equal than the current time of the process. All messages timestamped with greater values than the local time, must have equal timestamps. with the tag of the state where it is received. Finally, (IV.) states if messages from future rounds are stored in the reception variables, any statement that is executed in the following must not change the observable state, but rather increase the tag until the process has arrived at the maximal time it received a message from.

Tags and CompHO *protocols.* An intra CompHO protocol defined in Section 4 executes a (infinite) sequence of phases, each consisting of a fixed number of rounds. It is thus natural to annotate the code of an asynchronous protocol with a tag ($\texttt{phase}, \texttt{round}$). In an inter CompHO protocol, within a round, processes may call an inner CompHO protocol. Here, an instance of an inner round can be identified by phase and round of the outer (calling) protocol, and phase and round of the inner protocol. We are thus led in the following to consider tags that capture this structure:

We start with preliminary definitions. Given two values $a \in (\mathcal{D}_A, \prec_A)$ and $b \in (\mathcal{D}_B, \prec_B)$, $succ(a,b) = (sa, sb)$ where (1) if b is the maximum value in $(\mathcal{D}_B, \prec_B)$, then $sb = 0$ where 0 is the minimum value in $(\mathcal{D}_A, \prec_A)$ and sa is the successor of a w.r.t. the order $\prec_A$, denoted $succ(a)$; (2) else $a = sa$ and $sb = succ(b)$ is the successor of b in $(\mathcal{D}_B, \prec_B)$.

Definition 3 (CompHO synchronization tag). *Given a protocol* $\mathcal{P}$ *annotated with a synchronization tag* ($\texttt{SyncV}, \texttt{tags}, \texttt{tagm}$), *the tag is called* CompHO synchronization tag *if* $\texttt{SyncV}$ *has an even number of variables, i.e.,* $\texttt{SyncV} = (v_1, v_2, \dots, v_{2m-1}, v_{2m})$, *such that each pair* (v_{2i-1}, v_{2i}) *has a different type (at least on one of the components) and*

– v_{2i} *takes a constant number of values, forall i in* $[1,m]$,
– *the monotonic increasing order is refined; for any local execution* $\pi = s_0 A_0 s_1 A_1 \ldots \in [\![\mathcal{P}]\!]_p$ *of a process p in the semantics of* $\mathcal{P}$, $[\![\texttt{tags}]\!]_{s_0} \leq_{succ} [\![\texttt{tags}]\!]_{s_1}$ *where* $(a_1, a_2, \dots, a_{2m-1}, a_{2m}) \leq succ(a'_1, a'_2, \dots, a'_{2m-1}, a'_{2m})$ *iff*

- *if* $(a'_{2i-1}, a'_{2i}) \geq succ((a_{2i-1}, a_{2i}))$ *then* $(a'_{2j-1}, a'_{2j}) = (a_{2j-1}, a_{2j})$ *for all* $j < i$ *and* $(a'_{2j-1}, a'_{2j}) = (\bot, \bot)$ *forall* $j > i$.

Further, if $(a'_{2i-1}, a'_{2i}) = (succ(a_{2i-1}), 0)$ *or* $(a'_{2i-1}, a'_{2i}) = succ((a_{2i-1}, a_{2i}))$*, the tag is called* incremental.

For every $1 \leq i \leq m$*,* v_{2i-1} *is called a* phase tag *and* v_{2i} *is called* round tag.

5.1 Verification of synchronization tags

Given a protocol $\mathcal{P}$ annotated with a (CompHO-) tag (`SyncV`, `tags`, `tagm`), checking that the tag is a (CompHO-) synchronization tag reduces to checking a reachability problem on the local code, that is, in a sequential system.

The non-sequential instructions are the sends and receives, appearing in Conditions (II.) and (III.) of Definition 2. Checking that sent messages are tagged with the tag of the state they are sent in, that is Condition (II.), reduces to checking equality between local variables: the components (tagged by `tagm`) of a message type variable m and the local variables associated by `tags` at the control location that sends m. Recall that *send* does not modify the local state, so, it can be replace with an assert corresponding to the aforementioned equality.

Checking that messages with lower tags are dropped, that is, Condition (III.), is done by checking that the messages that are added to `mbox` have values (on the tagged components) greater than or equal to the local variables associated by `tags` at the control location where the addition occurs. We assume that *recv* may return any message and we check that the filters that guard the message's addition to the mailbox respect the order relation w.r.t. the state tags. This is again expressed by a state property that relates message fields with tag variables.

Conditions (I.) and (IV.) in Def. 2 (and Def. 3) translate into transition invariants over the synchronization variables. They state that the lexicographic order (monotonic or increasing) is preserved by any two consecutive assignments to the synchronization variables.

We automated these checks with the static verifier Verifast, and report in Section 8 on our experiments.

6 Reducing an asynchr. execution to its canonic form

After having introduced synchronization tags, we now show that any execution of an asynchronous protocol that has a synchronization tag can be reduced to a canonic execution. The proof proceeds in several steps, where in each step we will obtain a more restricted execution. The steps are as follows:

Asynchronous executions. We start with an asynchronous execution `ae` $\in [\![\mathcal{P}]\!]$ as defined in Section 3. Due to asynchronous interleavings, an action at process p that belongs to round k may come before an action at some other process q in round k', for $k' < k$.

Big receive. In order to capture jumping forward in rounds, we will regroup statements at different process to arrive at an asynchronous execution, where for each process a sequence of receive statements (followed by local computations stm for a jump) appears in a block. Thus, we can replace these blocks by a single atomic *Receive*. The resulting executions we denote by $[\![\mathcal{P}_{\mathsf{Rcv}}]\!]$.

Monotonic executions. We reduce asynchronous executions with Big receive semantics to execution where all tags are (non-strictly) monotonically increasing. As a result, all actions for round k' appear before all actions for all rounds k, for $k' < k$.

Round-based executions. We reduce monotonic executions to CompHO executions as defined in Section 4.

In each step, we maintain the following important property between the original execution and the execution we reduce to:

Definition 4 (Indistinguishability). *Given two executions π and π' of a protocol $\mathcal{P}$, we say a process p cannot distinguish locally between π and π' w.r.t. a set of variables W, denoted $\pi \simeq_p^W \pi'$, if the projection of both executions on the sequence of states of p, restricted to the variables in W, agree up to finite stuttering, denoted, $\pi\lfloor_{p,W} \equiv \pi'\lfloor_{p,W}$.*

Two executions π and π' are indistinguishable *w.r.t. a set of variables W, denoted $\pi \simeq^W \pi'$, iff no process can distinguish between them, i.e., $\forall p.\ \pi \simeq_p^W \pi'$.*

We focus on indistinguishability because it preserves so-called local properties [8], or equivalently properties that are closed under local stuttering. Important fault-tolerant distributed safety and liveness specifications fall into this class: consensus, state machine replication, primary back-up, k-set consensus, etc.

Definition 5 (Local properties). *A property ϕ is* local *if for any two executions a and b that are indistuingishable $a \models \phi$ iff $b \models \phi$.*

In the following we will denote by S_i a global state and by $s_i(p)$ the local state of process p in the global state S_i.

Reducing Asynchrony to Big receive. This reduction considers the receive statements. If the local execution is of the form $\pi = \dots s_i^p, receive_i, s_{i+1}^p$, $receive_{i+1}$, $s_{i+1}^p, \dots$, in the asynchronous execution, the two receive actions can be interleaved by actions of other processes. Following the theory by Lipton [29], all receive statements are right movers with respect to all other operations of other processes, as the corresponding send must always be to the left of the receive. In this way, we reduce an asynchronous execution to one where local sequences of receives appear as block. By the same argumentation, this block can be moved right until the first stm action of this process. Again the resulting block can be moved to the right w.r.t. actions at other processes. By repeating this argument, we get an asynchronous executions with blocks that consist of several receives (possibly just one receive) and stm statements such that at the end the local state tag matches the maximal received message tag, i.e., the process has jumped forward to a round from which it received a message. We will subsume such a block by an (atomic) action *Receive*, and denote by $[\![\mathcal{P}_{\mathsf{Rcv}}]\!]$ the asynchronous semantics with the atomic Receive.

Reducing Big receive to monotonic.

Theorem 1. *Given a program $\mathcal{P}$ if there is a synchronization tag* $(\texttt{tags}, \texttt{tagm})$ *for $\mathcal{P}$, then $\forall \mathsf{ae} \in [\![\mathcal{P}_{\mathsf{Rcv}}]\!]$, if $\mathsf{ae} = \ldots S_{i-1}, A^p_i, S_i, A^q_{i+i}, S_{i+i} \ldots$, and $[\![\texttt{tags}]\!]_{s_i(p)} > [\![\texttt{tags}]\!]_{s_{i+1}(q)}$, then*

$$\mathsf{ae}' = \ldots S_{i-1}, A^q_{i+i}, S'_i, A^p_i, S_{i+i} \ldots \in [\![\mathcal{P}]\!].$$

Further $\mathsf{ae}', \mathsf{ae}$ are indistinguishable w.r.t. all protocol variables, i.e., $\mathsf{ae}' \simeq \mathsf{ae}'$.

Proof. From $[\![\texttt{tags}]\!]_{s_i(p)} > [\![\texttt{tags}]\!]_{s_{i+1}(q)}$ and (I.) follows that $p \neq q$, so that swapping cannot violate the local control flow. As $p \neq q$, if A^q_{i+i} is a send or a stm, the action at p has no influence on the applicability of A^q_{i+i} to S_i. The only remaining case is that A^q_{i+i} is a *Receive*. Only if A^p_i sends a message m that is received in A^q_{i+i}, A^q_{i+i} cannot be moved to the left. We prove by contradiction that this is not the case: By (II.), $[\![\texttt{tags}]\!]_{s_i(p)} = [\![\texttt{tagm}]\!]_m$. By (III.) and (IV.), and the atomicity of Receive, $[\![\texttt{tags}]\!]_{s_{i+1}(q)} = [\![\texttt{tagm}]\!]_m$. Thus, $[\![\texttt{tags}]\!]_{s_i(p)} = [\![\texttt{tags}]\!]_{s_{i+1}(q)}$ which provides the required contradiction to the assumption of the lemma $[\![\texttt{tags}]\!]_{s_i(p)} > [\![\texttt{tags}]\!]_{s_{i+1}(q)}$.

The statement on indistinguishability follows from the reduction. □

By inductive application of the theorem, we obtain:

Corollary 1. *Given a program $\mathcal{P}$ if there is a synchronization tag* $(\texttt{tags}, \texttt{tagm})$ *for $\mathcal{P}$, then $\forall \mathsf{ae} \in [\![\mathcal{P}_{\mathsf{Rcv}}]\!]$, there is a monotonic asynchronous execution* $\text{mono}(\mathsf{ae}) = \ldots S_{i-1},\ A^p_i,\ S_i, A^q_{i+i}, S_{i+i} \ldots$, *where for each i and any two processes p and q, $[\![\texttt{tags}]\!]_{s_i(p)} \leq [\![\texttt{tags}]\!]_{s_{i+1}(q)}$.*

The monotonic execution $\text{mono}(\mathsf{ae})$ is thus a sequential composition of actions of rounds in increasing order, that is, all actions of round k occur before all actions in round $k+1$, for all k. Thus, the global state between the last round k action and the first round $k+1$ action constitutes the boundary between these rounds. In the following section we will show that we can simplify the reasoning within a round.

Reducing a Round to a Synchronous round. In order to reduce monotonic executions into *HO* semantics we re-use arguments by [8], which we have to extend for asynchronous programs. We consider distributed programs of a specific form: The local code within each round is structured in that first there are send, then Receive, and then other statements. Similarly, we only consider protocols where it is sufficient to check states only when the tags change. If the local code within a round is "subsumed" to a single local transition, we do not lose any observable events. Rather, the subsumption is locally stutter equivalent to the original asynchronous semantics.

As we start from monotonic executions here, we can restrict ourselves to swapping actions within a round and only have to care about moving send and receive actions. For this, we can use the arguments from [8]: the send actions are left movers with respect to all other operations, Receive actions are left movers with all statements except sends. By repeated application of their arguments, we arrive at executions where within a round all send actions come before all Receive actions, which come before all other actions. We call these executions send-receive-compute executions:

Proposition 1. *For each monotonic asynchronous execution, there exists an indistinguishable asynchronous send-receive-compute execution.*

All sends are non-interfering and can thus be "accelerated" or "subsumed" in one global send action. As in CompHO all messages sent in a round must be of identical payload type, the type to be sent in the subsumed action is the union of the payload types. Similar for receive. Here, the *HO* sets are defined by the processes of which a process received messages in its receive operations. If in the original execution π process p jumped over round r, there are no send, receive, and local computation actions for p in r. As we require in the *HO* semantics that every process performs these steps in each round, we have to complete the execution with `nop` steps for the missing rounds. As they do not change which messages are received in the asynchronous execution, and which local states the processes go through, we again remain stutter equivalent, and obtain.

Proposition 2. *For each asynchronous send-receive-compute execution, there exists an indistinguishable CompHO execution se, where the messages received in a Receive statement correspond to the HO sets.*

Following Definition 5, local properties are those closed under indistinguishability, so that we obtain the following theorem.

Theorem 2. *If there exists a synchronization tag* (`SyncV`, `tags`, `tagm`) *for* $\mathcal{P}$, *then* $\forall ae \in [\![\mathcal{P}]\!]$ *there exists an CompHO-execution se that satisfies the same local properties.*

7 Code to code rewriting of Asynchronous to CompHO

We introduce a rewriting algorithm `make-CompHO` that takes as input an asynchronous protocol $\mathcal{P}$ annotated with a synchronization tag and either produces a (inter-procedural) CompHO protocol, denoted CompHO($\mathcal{P}$), whose executions are indistinguishable from the executions of $\mathcal{P}$, or aborts.

Replacing reception loops with atomic mailbox definition. We consider asynchronous protocols where message reception is implemented in a distinguished loop, that we refer to as "reception loop". A reception loop is a simple `while(true)` loop, that (1) contains `recv` statements, (2) writes only to variables of message type, or containers of message type objects, (3) the loop is exited either because

```
if (coord() == me){
   ballot++;
   msg *m = create_msg(ballot,label,myid);
   if (mbox!=0 && mbox->size ==1) {
         ballot = mbox->message->ballot;
         leader = mbox->message->sender;}}
```

```
if !(coord() == me){
 ballot++;
 if (mbox!=0 && mbox->size ==1) {
     ballot = mbox->message->ballot;
     leader = mbox->message->sender;}}
```

Fig. 9: Two blocks defining round `NewBallot` in the protocol from Fig. 1.

of a timeout or because some condition over the message type variables wrote in the loop holds. The algorithm in Fig. 1 has four reception loops, at lines 29, 43, 55, and 65. The exit of a reception loop is typically cardinality constraints or timeout, e.g., `mbox`$\rightarrow$`size > n/2` or `mbox`$\rightarrow$`size == 1`.

A reception loop is replaced by havoc assignments of the message type/container of message type variables written by the loop. The code following the loop is left unchanged except in the following cases: (1) the boolean conditions that refer to a loop timeout are replaced by the negation of all the other conditions to exit the loop; (2) if the loop does not have a timeout exit, that is, processes wait until all required messages are received, the code following the loop is wrapped into an if statement, allowing its execution only if the loops exit condition holds. In the rest of the section we consider only protocols whose reception loops have been replaced by havoc statements.

Rewriting protocols with incremental synchronization tags. Let $\mathcal{P}$ be a protocol consisting only of one loop annotated with an incremental CompHO synchronization tag (ph, rd). The rewriting in this section builds a (intra-procedural) CompHO protocol in two steps: (1) each iteration of $\mathcal{P}$'s loop defines a phase and (2) the code of each phase (the loops body) is decomposed into rounds.

Phases are matched with loop iterations if ph (representing the phase number) is increased exactly once in each iteration to its successive value (like the loop counter). To this we assume the protocol verified for strengthened annotations regarding tags monotonicity: the relation $ph = old\_ph + 1$ is an invariant of the loop, where old_ph is previous value of ph. If ph has initial value 1, then the phase number matches the iteration number. Otherwise ph is shifted by a bounded value. The communication closure induced by the tags (see Theorem 2) ensures that two processes communicate only when they are in the same iteration. Hence, it is sound to construct a phase i by composing the ith loop iteration of all processes. Within a phase it remains to locate the round boundaries.

A CompHO synchronization tag, ensures that the round variable rd takes a bounded number of values: in the running examples these values are `NewBallot` and `AckBallot`. Round bounderies are defined by the beginning/end of a loop iteration and the assignments to the round variable rd.

Processes can have different behaviors in the same round, depending on their local state and the messages received, although they execute the same code and go through the same sequence of rounds. For example, in the round `NewBallot` only the processes designated coordinators by the oracle send a message. Similarly, in the `AckBallot` round only the processes that received a message in this round are going to update their logs. As usual these different behaviors are captured by branching instructions in the loops's body, and each path in the loop's body identifies a possible process behavior in sequence of rounds.

For each value ℓ of rd, to compute the code of round ℓ, we consider each path π in the control flow graph of the loop's body and we identify (1) a block of instructions (possibly empty) B^{π}_{ℓ}: a sequence of instruction in π that starts with $rd = \ell$ and ends with the instructions preceding the next assignment to rd; (2) the context under which each block B^{π}_{ℓ} is executed, that is a condition $cond^{\pi}_{\ell}$ that is the conjunction of all the branches leading to $rd = \ell$ on the path π. The B_{ℓ} is the sequential composition of all `if (`$cond^{\pi}_{\ell}$`)` B^{π}_{ℓ} with π path in the control flow. Fig. 9 shows the two blocks defining round NewBallot, corresponding to the leader follower paths in the control flow graph.

To maintain the context in which a sequence of instructions is executed, i.e., $cond^{\pi}_{\ell}$, we introduce auxiliary variables. For each variable `x` in a conditional we introduce an auxiliary variable old_x (of the same type with x), that is assigned only once to `x`, i.e, $old\_x :=$ `x`, before the condition is evaluated. The conditionals $cond^{\pi}_{\ell}$ are defined over these auxiliary variables. If the variable is not read without being first assigned to a default value, we can abstract old_x to boolean. This is the case of all our benchmarks, where auxiliary variables remember values of the mailbox in previous rounds.

Moreover, if the values of the round variables do not take all values in their domain, each condition $cond^{\pi}_{\ell}$ is conjuncted with the check whether the round number of the CompHO equals the round tag variable. Intuitively, if the check fails, the asynchronous code has set the round tag to a future round (of the same phase), which results in skipping the CompHO round.

Finally, the code of every round, that is, B^{π}_{ℓ} is split into a $Send^{\pi}_{\ell}$ block, consisting of all send statements B^{π}_{ℓ} guarded by the conditionals preceding them and an $Update^{\pi}_{\ell}$ block that contains the rest of the code in B^{π}_{ℓ} except the mbox's havoc. We assume $Update^{\pi}_{\ell}$ contains no send, no recv, and no assignments to message type variables. Otherwise the rewriting aborts. (One could try compiler optimization techniques to reorder instructions towards the imposed order.)

The rewriting eliminates the phase and round tag variables (if no rounds are skipped) from the local process variables all program locations are tagged with the same variables. Reads of these variables are replaced with reads of the round, respectively phase number, of the CompHO protocol.

Example 3. For example the asynchronous protocol in Fig. 1 in Sec. 2 is rewritten into an intra-procedural CompHO one, given in Fig. 3 in Sec. 2. However Fig. 3 contains a simplification w.r.t. what is automatically generated. The code between the lines 30 to 34 appears twice, ones if the process is leader and ones if it is not. Similar for the first round.

An asynchronous protocol is structured, if receptions loops are emphasized as defined in Sec. 7 and the blocks associated with a round are a sequence of send followed by update statements.

Theorem 3. *Given a structured asynchronous protocol $\mathcal{P}^{async}$ consisting of only one loop, that is annotated with a strictly incremental* CompHO *synchronization tag* (`SyncV`, `tags`, `tagm`) *of size two,* `SyncV` $= \{$`phase_tag`, `phase_tag`$\}$, `make-CompHO` *builds an intra-procedural* CompHO *protocol $\mathcal{P}^{\mathsf{CompHO}}$ whose executions are indistinguishable from the executions of $\mathcal{P}^{async}$. The resulting protocol has only one phase that consists of as many rounds as the domain of evaluation of the round_ tag. $\mathcal{P}^{\mathsf{CompHO}}$ sends exactly the same messages as $\mathcal{P}^{async}$.*

Jumping over phases. The catch up mechanism allows processes to receive messages from future rounds, which leads to a jump to the received phase number. Moreover, in general non-incremental tags allow processes to skip to future tags (which may happen, e.g., if the leader of the current phase is suspected to have crashed). In this section, we reduce the problem of rewriting a protocol with non-incremental tags to the rewriting a protocol with incremental tags.

In Sec. 7 the loop counter coincides (modulo an initial shift) with the phase tag. Jumping over phases potentially increases the phase tag by more than one, "desynchronizing" it from the loop counter. To apply the rewriting from Sec. 7 we introduce empty loop iterations, when the loop counter is smaller than the phase tag, and we reinterpret the initial increasing tags over the new loop counter, resulting into an incremental tag annotation.

First we identify the jumping control locations. These are locations where the phase tag (1) is assigned a value that depends on the mailbox and (2) the communication closure checks show that a message tag in the mailbox may be strictly greater than local tag; cf. Section 5.1. In this case the tool partitions the path with the jumping instruction into the three sequences of instructions `Before_Jump`, `Jump`, and `After_Jump`. Since jumps are conditional, we have to capture the cases without jump, where `Before_Jump` and `After_Jump` are both part of a round and the case with jump where `Before_Jump` and `After_Jump` are parts of code for different rounds. The rewriting encodes this cases with an auxiliary boolean variable that non-deterministically flags a jump, and a `continue` statement before the jumping instruction.

In all examples we explored, `Before_Jump` is either empty or consists only of one `send` instruction. Both cases are simpler and correspond to no code being execution in CompHO semantics, and it is naturally captured by empty HO sets there (in case there are send instruction the messages can always be lost).

Protocols with nested loops. Let us consider a protocol $\mathcal{P}$ without reception loops. The rewriting algorithm proceeds bottom-up: it starts rewriting the most inner loop using the procedure above. For each outer loop it first replaces the nested loop with a call to the computed CompHO protocol, and then applies the same rewriting procedure. Since we considered passing by value procedure calls in the CompHO semantics, all local variables are input parameters.

Inner loops appearing on different branches may belong to the same subprotocol; in other words these different loops exchange messages. If `tags` associates different synchronization variables to different loops then the rewriting builds one (sub-)protocol for each loop. Otherwise, the rewriting merges the loops tagged with the same synchronization variables into one CompHO protocol.

To soundly merge several loops into the same CompHO protocol, the rewrite algorithm identifies the context in which the inner loop is executed.

Theorem 4. *Given a structured asynchronous program $\mathcal{P}^{async}$ with a CompHO synchronization tagging function* $(\mathtt{SyncV}, \mathtt{tags}, \mathtt{tagm})$*, then* `make-CompHO` *applied $\mathcal{P}^{async}$ returns an inter-procedural CompHO protocol $\mathcal{P}^{\mathsf{CompHO}}$ whose executions are indistinguishable from the executions of $\mathcal{P}^{async}$.*

8 Experimental results

We implemented the rewriting procedure in a prototype tool and applied the tool to several fault-tolerant distributed protocols. The tool is available online.[4] Fig. 11 summarizes our experimental results.

Verification of synchronization tags. The tool takes protocols in a C embedding of the language from Sec. 3 as input. We use a C embedding to be able to use Verifast [23] for checking the conditions in Sec. 5.1, i.e., the communication closure of an asynchronous protocol. Verifast is a deductive verification tool based on separation logic for sequential programs. The C embedding uses the prototype of the functions `send` and `receive` (we assume their semantics is the one in Sec. 3).

```
predicate tag_leq(int old_ballot, int
      old_label, int ballot, int label) =
      (old_ballot < ballot) ||
      (old_ballot ==ballot &&
      old_label<=label) ;
 predicate mbox_tag_eq(int ballot, int
       round, struct List* n;) =
 n == 0 ? true :
 n->message |-> ?msg &*& msg->ballot |->
       ?v &*& msg->label |-> ?r &*&
       msg->sender |-> _ &*&
 malloc_block_Msg(msg) &*&
       malloc_block_List(n) &*& n->next
       |-> ?next &*& n->size |-> ?s &*&
 n!=next &*& ballot== v &*& round== r
       &*& mbox_tag_eq(ballot,round,
       next) ;
```

Fig. 10: Predicates in separation logic expressing the order relation over tags and the condition that all messages in the mailbox are timestamped with the receiver's local time.

The user specifies in a configuration file the synchronization tag by (i) defining the number of (nested) protocols in the input file, (ii) for each protocol, the phase and round variables, and (iii) for each messages type the fields that encode the timestamp, i.e., the phase and round number. Fig. 11 gives the names of phase and round variables of published protocols we use as benchmarks.

The tool expects the input file to be annotated with assert statements for checking the conditions in Definition 2 w.r.t. the tags given in the configuration file and the auxiliary annotations Verifast needs to prove these asserts (inductive invariants). The annotations are defined over program variables and auxiliary history variables. Auxiliary variables are necessary to encode the monotonicity of the tags. The tool calls Verifast and checks that the input contains `assert`

[4] https://github.com/alexandrumc/async-to-sync-translation

Protocol	Tags	Annot.	Async	Sync
Consensus [7, Fig.6]	$ph = r_p$ rd= {Phase1, Phase2, Phase3, Phase4}	661	332	251
Two phase commit	ph = i, rd= {Query, Vote, Commit, Ack}	588	252	242
Figure 1^{*,V}	ph = ballot, rd = {NewBallot, AckBallot}	650	255	110
ViewChange* [35]	$ph1$ = `view`, $rd1$ = {StartViewChange, DoViewChange, StartView}	720	352	172
Normal-Op^V [35]	ph = op_number rd = {Prepare, PrepareOK, Commit}	628	266	182
Multi-Paxos^{*,V} [27]	$ph1$ = ballot, $rd1$ = {NewBallot, AckBallot, NewLog} $ph2$ = op_number, $rd2$ = {Prepare, PrepareOK, Commit}	1646	621	405

Fig. 11: Communication-closed asynchronous protocols. The superscript * identifies protocols that jump over phases. The superscript V marks the protocols whose synchronous counter-part we verified.

`tag_leq(old_ballot, old_label, ballot, label)` after each (pair of) assignment(s) of the phase and round variables. Observe that conditions (I.)–(III.) in Definition 2 are numeric constraints over the phase and round variables, and several other tools might verify them. However, condition (IV.) requires reasoning about the content of the mailbox, a potentially unbounded data structure. Here is where we used the strength of Verifast, to reason about dynamically allocated data structures. The size of the program annotated with the proofs for the asserts is given in LoC in the column "Annot." in Fig. 11. If all the checks are passed, then the rewriting proceeds, otherwise the tool outputs a warning.

Rewriting. While checking the verification tags can be done for any annotated asynchronous protocol, the rewriting tool checks whether the asynchronous protocol is in a specific form and only then translates it into CompHO. While in theory this is a restriction, the benchmarks in Fig. 11 show that well-known algorithms are rewritten by our tool. For instance, the algorithm [7, Fig. 6] solves consensus using an eventually strong failure detector. The algorithm jumps over rounds in a specific way. If a special decision message is received, a process jumps forward to a decision round and outputs the decision value. The resulting algorithm is much like Last Voting in [10, Fig. 5]. ViewChange is a leader election algorithm similar to the one in ViewStamped: unlike in the running example (unlike Paxos), in ViewChange processes first agree to change the current leaders, and than on a leader. The phase number is the view number (like in Paxos), that is, two processes either agree on the identity of a leader in a view or they know of no leader. Normal-Op is the sub-protocol used in ViewStamped to implement the broadcasting of new commands by a stable leader. Multi-Paxos is described in Sec 4. It is Paxos from [27] over sequences, without fast paths, where the classic path is repeated as long as the leader is stable. In Paxos parlance, the tags for leader election (outer protocol) are *Phase1a*, *Phase1b*, *Phase1aStart* (in this order). The rounds of the sub-protocol are called *Phase2aClassic*, *Phase2bClassic*, *learn*. We considers that acceptors and leaders play also the role of learners.

Our tool has rewritten the protocols from Fig. 11. The implementation uses pycparser [3], a parser for the C language written in pure Python, to obtain the abstract syntax tree of the input protocol. The last two columns of Fig. 11 give the size in LoC of the asynchronous protocol without annotations and the size of its synchronous counterpart computed by the rewriting procedure from Sec. 7.

Verification. We have verified the safety specification (agreement) of the CompHO counter-parts of the running example (Figure 1), Normal-Op, and Multi-Paxos, by deductive verification using the Consensus Logic (CL for short) defined in [13]. To this, we encoded the specification and the transition relation in CL, and used CL's semi-decision procedure for satisfiability [14] to discard the verification conditions. For Multi-Paxos we did a modular proof. First we prove the correctness of the sub-protocols (executed in case of a stable leader). Its specification is that the logs of all processes that execute the sub-protocol are equal at the beginning and at the end of each phase (after an iteration of `Prepare`, `PrepareOk`, `Commit`), knowing that processes start the sub-protocol with equal logs. Moreover, the sub-protocol preserves the invariant property that a majority of processes have the same prefix, consisting of all the committed commands. Then we prove the leader election outer loop correct. Its specification states that there is at most one leader in a ballot (like in (1)) and that a majority of processes have the same prefix, consisting of all the committed commands. The leader picks the longest log of its followers. The fact that all committed values are logged by a majority of processes ensures that the new log proposed by the leader will not have lost any committed commands. However, there are no guarantees for the uncommitted commands.

9 Related work

Our goal is to link synchronous or round-based models to asynchronous models via the notion of communication closure [17]. Exploiting this for better design and simpler paper-and-pencil proofs was considered, e.g., in [34,11,18]. Several round-based computational models are based on this idea [16,30,10,19,39]. Commonly the underlying idea is to design an algorithm for the round-based setting and deduce results for the asynchronous. A method that takes round-based code as input and generates asynchronous code was given in [15]. However, for efficiency reasons, designers often prefer to work with asynchronous code. Therefore, in this paper we start from asynchronous protocols, and compute the round-based canonic form as design artifact and for verification purposes. From the canonic form one can choose one of the existing automated verification methods for round-based distributed algorithms [42,43,9,12,13,20,31,1].

There are several other frameworks for the verification of asynchronous distributed algorithms, e.g., Verdi [45], IronFleet [22], ByMC [25], Ivy [37], and Disel [40]. Very interesting distributed algorithms have been verified in these frameworks. Still, they require considerable expertise either in manually fitting asynchronous code to the fragment that can be dealt with by the method, or in guiding interactive theorem provers. Typically, these works also consider verification of specific algorithms which makes it hard to generalize ideas.

Our research belongs to an effort to develop techniques for automated reduction to synchronized executions. Three concurrent approaches in this quest are the exciting results in [5], [26] and [2,21]. Compared to their work, our approach is less guided by specific communication patterns of existing systems. Rather we put communication closure in the center of our considerations. Hence, we are more permissive to different communication structures. For instance, the recent paper [21] does not allow skipping rounds, with the side effect that they cannot model that a process remains leader for several consecutive iterations, which is an important efficiency mechanism in systems that implement ideas from Paxos [27] and Viewstamped Replication [35]. The notion of k-synchronizability in [5] is restricted to FIFO communication channels. In contrast our method does not make any assumptions about the communication model between processes (works for UDP, TCP/IP). Moreover, unbounded jumps over phases cannot be captured by k-synchronizability. The method in [26] adopts verification methods for remote procedure calls to leader/follower communication. As a result they do not support rounds with all-to-all communication or that a leader plays also the role of a follower; both is the case in our running example.

10 Conclusion and future work

We formalized the notion of communication closure of asynchronous protocols and showed that several challenging benchmarks satisfy this property. We showed that communication closure captures formally the intuition of protocol designers and is an enabler for a synchronous canonic form of asynchronous protocols. This canonic form enables the use of different verification techniques, and we verified the several benchmarks using the Consensus Logic framework [13].

We consider the verification of synchronous round-based protocols an orthogonal problem, however progress in this research area has direct impact on the verification of asynchronous protocols that are communication-closed. Roughly the main difficulty regarding automating reasoning of synchronous systems comes from the data they manipulate and not from their control structure.

Our methods preserves relevant safety and liveness properties. Reasoning about liveness in CompHO requires assumptions about the HO sets, which can be done in Consensus Logic. However, besides initial theoretical results [41], the connection between HO sets and the asynchronous world is formally not well understood, yet. Thus, there is no automated method that translates asynchronous receptions loops with time-outs, etc. into HO sets. This would be required for total correctness regarding liveness and is subject to future work.

References

1. Aminof, B., Rubin, S., Stoilkovska, I., Widder, J., Zuleger, F.: Parameterized model checking of synchronous distributed algorithms by abstraction. In: VMCAI. pp. 1–24 (2018)
2. Bakst, A., von Gleissenthall, K., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL 1(OOPSLA), 110:1–110:27 (2017)
3. Bendersky, E.: pycparser. https://github.com/eliben/pycparser, (retrieved Nov 7, 2018)
4. Biely, M., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper, A., Widder, J.: Tolerating corrupted communication. In: PODC. pp. 244–253 (2007)
5. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: CAV. pp. 372–391 (2018)
6. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: An engineering perspective. In: PODC. pp. 398–407 (2007)
7. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
8. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: RP. LNCS, vol. 5797, pp. 93–106 (2009)
9. Charron-Bost, B., Debrat, H., Merz, S.: Formal verification of consensus algorithms tolerating malicious faults. In: SSS, pp. 120–134. Springer (2011)
10. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22(1), 49–71 (2009)
11. Chou, C., Gafni, E.: Understanding and verifying distributed algorithms using stratified decomposition. In: PODC. pp. 44–65 (1988)
12. Debrat, H., Merz, S.: Verifying fault-tolerant distributed algorithms in the heard-of model. Archive of Formal Proofs 2012 (2012)
13. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: VMCAI. pp. 161–181 (2014)
14. Drăgoi, C., Henzinger, T.A., Zufferey, D.: psync. https://github.com/dzufferey/psync, (retrieved Nov 12, 2018)
15. Drăgoi, C., Henzinger, T.A., Zufferey, D.: Psync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL. pp. 400–415 (2016)
16. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. JACM 35(2), 288–323 (Apr 1988)
17. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2(3), 155–173 (1982)
18. Engelhardt, K., Moses, Y.: Safe composition of distributed programs communicating over order-preserving imperfect channels. In: IWDC. pp. 32–44 (2005)
19. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In: PODC. pp. 143–152 (1998)
20. v. Gleissenthall, K., Bjørner, N., Rybalchenko, A.: Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In: PLDI. pp. 599–613 (2016)
21. v. Gleissenthall, K., Gökhan Kici, R., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony. In: POPL (2019), (to appear)
22. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving safety and liveness of practical distributed systems. Commun. ACM 60(7), 83–92 (2017)

23. Jacobs, B., Smans, J., Piessens, F.: A quick tour of the VeriFast program verifier. In: APLAS. LNCS, vol. 6461, pp. 304–311 (2010)
24. Jepsen: Distributed systems safety research. Web page jepsen.io. Retrieved Nov 7, 2018 (2018)
25. Konnov, I.V., Lazic, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL. pp. 719–734 (2017)
26. Kragl, B., Qadeer, S., Henzinger, T.A.: Synchronizing the asynchronous. In: CONCUR. pp. 21:1–21:17 (2018)
27. Lamport, L.: Generalized consensus and paxos. Tech. rep. (March 2005), https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/
28. Le Lann, G.: Asynchrony and real-time dependable computing. In: WORDS. pp. 18–25 (2003)
29. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
30. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
31. Marić, O., Sprenger, C., Basin, D.A.: Cutoff Bounds for Consensus Algorithms. In: CAV. pp. 217–237 (2017)
32. Mattern, F.: On the relativistic structure of logical time in distributed systems. Parallel and Distributed Algorithms pp. 215–226 (1989)
33. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in egalitarian parliaments. In: SOSP. pp. 358–372 (2013)
34. Moses, Y., Rajsbaum, S.: A layered analysis of consensus. SIAM J. Comput. 31(4), 989–1021 (2002)
35. Oki, B.M., Liskov, B.: Viewstamped replication: A general primary copy. In: PODC. pp. 8–17 (1988)
36. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC '14. pp. 305–319 (2014)
37. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: safety verification by interactive generalization. In: PLDI. pp. 614–630 (2016)
38. van Renesse, R., Schiper, N., Schneider, F.B.: Vive la différence: Paxos vs. viewstamped replication vs. zab. IEEE Trans. Dependable Sec. Comput. 12(4), 472–484 (2015)
39. Santoro, N., Widmayer, P.: Time is not a healer. In: STACS. LNCS, vol. 349, pp. 304–313 (1989)
40. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL 2(POPL), 28:1–28:30 (2018)
41. Shimi, A., Hurault, A., Queinnec, P.: Characterizing asynchronous message-passing models through rounds. In: OPODIS (2018), (to appear)
42. Tsuchiya, T., Schiper, A.: Model checking of consensus algorithms. In: SRDS. pp. 137–148 (2007)
43. Tsuchiya, T., Schiper, A.: Using bounded model checking to verify consensus algorithms. In: DISC. pp. 466–480 (2008)
44. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Computing 23(5-6), 341–358 (2011)
45. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI. pp. 357–368 (2015)