Reducing asynchrony to synchronized rounds

Cezara Drăgoi[1] and Josef Widder[2]

[1] INRIA, ENS, PSL, France
[2] TU Wien, Austria

Abstract. Synchronous computation models simplify the design and the verification of fault-tolerant distributed systems. For efficiency reasons such systems are designed and implemented using an asynchronous semantics. In this paper, we bridge the gap between these two worlds. We introduce a (synchronous) round-based computational model and we prove a reduction for a class of asynchronous protocols to our new model. The reduction is based on properties of the code that can be checked with sequential methods. We apply the reduction to state machine replication systems, such as, Paxos, Zab, and Viewstamped Replication.

1 Introduction

Fault-tolerant distributed systems provide a dependable service on top of unreliable computers and networks. Fault tolerance protocols ensure that the unreliable replicas are perceived as a single reliable component from the outside. This intuition has been formalized under terms like strong consistency, consensus, and state machine replication, and has been addressed by distributed algorithms for atomic broadcast, atomic commitment, or primary-backup.

Designing such systems is a hard and error-prone task. The designer has to anticipate faults, concurrency, and asynchrony due to different processing speeds and network delays. While a clear programming abstraction like synchronized rounds [4] would relieve the designer from many of these challenges, it is well-understood that synchronous distributed systems, are often "impossible or inefficient to implement" [20, p. 5]. Hence, designers turn to the asynchronous model, in which the performance emerges [17] from the current load of a system, which in normal operation entails significant better performance. To get the best of the synchronous and the asynchronous worlds, partially synchronous models have emerged [9,4,12] that allow us to reason in synchronous semantics, and capture asynchrony in an abstract way. The central design paradigm is to always ensure safety — even if the networks behaves bad temporarily — and to ensure liveness in good periods.

The probably best-known algorithmic ideas that ensure state machine replication on top of a partially synchronous system are *Paxos* [16] and *Viewstamped Replication* [24]. These ideas have been implemented in several systems, e.g., [14,22,25,23,15].

Verification of these asynchronous systems is in general undecidable. Finding classes of distributed that can be verified is major research challenge. Currently most approaches that address this problem are based on interactive theorem provers [31,18,29,13,28,26]. These methods require expertise in the tools, and changing details in the system design or code entail considerable manual effort in adapting existing proofs. On the contrary, automated verification of round-based (synchronous) distributed algorithms has recently made great progress [21,7,30,1]. The enabler for these techniques is the observation that instead of considering all local states that are the result of interleavings in asynchronous systems, for the verification of distributed systems, it is sufficient to reason about the global state at the boundaries between rounds, which simplifies drastically the proof arguments. Therefore we believe that the difficulty in automating the verification of distributed asynchronous systems comes from the lack of suitable abstractions when we reason about their behaviors.

Contributions. This paper introduces a new round-based (synchronous) model that truthfully captures relevant asynchronous distributed protocols for strong consistency. In order to achieve this, we address the following challenges.

1. We identify a class of asynchronous programs that have an "equivalent" (a term that we make precise) synchronous round-based semantics, namely *communication-closed* programs. While this notion has been originally introduced for CSP [10], its message-passing correspondence requires that a message that process i sends to j when i is in round r, is received by j only at a time when j is in its round r. We introduce a characterization in terms of local variables, so that it can be checked by sequential techniques. Examples of communication-closed asynchronous programs and algorithms include Zab[14], Raft [25], viewstamped replication [24] Paxos [16], atomic broadcast [2].
2. We prove a reduction theorem for such asynchronous programs following the idea of [10]. In contrast to Lipton's reduction [19], 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 round r send events of all processes, then the round r receive events of all processes,

and then all round r computation steps, for all rounds r in increasing order. In this way, an asynchronous execution corresponds to a synchronous (round-based) one where all processes simultaneously send round r messages, before they receive round r messages, etc. This reduction maintains local properties [3], that is, a fragment of LTL specifications that are tolerant to local stuttering.

3. To exploit communication closure, we require the system designer to annotate the program with tags that capture the rounds. Given that designers typically have and intuitive understanding of rounds — which is demonstrated many figures similar to Fig. 2 in work on such systems — this task does not require too much effort by the designer.
4. We extend the Heard-Of Model [4] (HO model for short) by introducing a compositional semantics. By this we address the control flow of programs that implement state machine replication. For instance, in Paxos [16] processes execute a sequence of independent agreements each with the goal of agreeing on what to write in the next element of a growing list of ballots. By compositional reasoning, the code that implements the agreement should be encoded and verified independently of the code that captures the move from one ballot to the next.
5. We define a re-writing procedure that, given an annotated program, produces an algorithm in the HO model, which paves the way for round-based verification. By separation of concerns, we relieve verification of local properties from the complications that are due to asynchrony.

Comparing with verification techniques that investigate alternative synchronous semantics [5,6], we fix not only semantically but also syntactically a synchronous computational model, and we define code annotations that capture its structure and are locally expressible and checkable.

2 Round-based compositional model

We introduce CompHO, a compositional extension of the HO model [4]. The HO model was introduced to model one-shot algorithms, while the systems discussed here require calls to multiple instances. Thus, an algorithm in CompHO consists of an interface and several machines, each defining a distributed procedure, with a main machine as entry point. The interface defines operations used to communicate with an external client.

```
1  Zab-Discovery
2  def init(){
3    leader := leader()}//oracle
4  def round Curr_E:
5  SEND cepoch to leader
6  UPDATE(Mbox)
7  if leader and |MBox(p)| > n/2
8       lab := New_E
9       newepoch := max of vals ∈ Mbox
10 def round New_E:
11 if lab = New_E SEND newepoch to all
12 UPDATE(Mbox)
13 if rcvd newepoch from leader
14      cepoch := newepoch; lab := Ack_E
15 def round Ack_E:
16 if (lab = Ack_E) SEND (log) to leader
17 UPDATE(Mbox)
18 if (leader and |Mbox|> n/2)
19       newlog := max history in Mbox
20       lab := New_L
21 if (!leader && lab=Ack_E) lab := New_L
22 def round New_L:
23 if (lab = New_L and leader) SEND nlog
       to all
24 UPDATE(Mbox)
25 if lab = New_L and newlog ∈ Mbox
26     log := newlog; lab := Ack_L
27 def round Ack_L:
28 if lab = Ack_L SEND Ack_L to leader
29 UPDATE(Mbox)
30 if (leader && MBox(p)>n/2) lab := Cmt
31 if(!leader && lab = Ack_L) lab := Cmt
32 def round Cmt:
33 if leader && lab = Cmt
34      SEND commit to all
35 UPDATE(Mbox)
36 if (lab = Cmt and leader in MBox)
37   for each zvid<i<log.size out(log[i])
38   xviz := log.size
39   log := broadcast (log, leader)
40 leader = leader()
```

Fig. 1: Zab-HO: Zab in CompHO

We start by giving the adaptation of Zookeeper's atomic broadcast protocol [14] in CompHO in Fig. 1 as example. The protocol solves ensures that all replicas store multiple requests issued by a client in the same order in a local `log`. Zab-HO consists of two types of CompHOmachines: `Discovery`, the main machine, and `Broadcast` (that we omit for presentation reasons). `Discovery` iterates over so-called epochs and implements leader election in each epoch. A leader identity is suggested to each process by an oracle `leader()`. As different processes may receive different suggestions, several rounds of message exchange shall ensure that (i) there is at most one leader, and (ii) it has a majority of followers: In the first round `Curr_E`, processes send their epoch number, i.e, `cepoch`, to their leader. If a process believes that it is the leader and has received a majority of messages, it picks the biggest one of them. Next, in `New_E`, a process who believes that it is the leader tries to impose the chosen value to a majority of processes, by sending `newepoch` to them. The Discovery protocol then continues in ping-pong fashion. In the first execution in Fig 2 only the first two processes en-

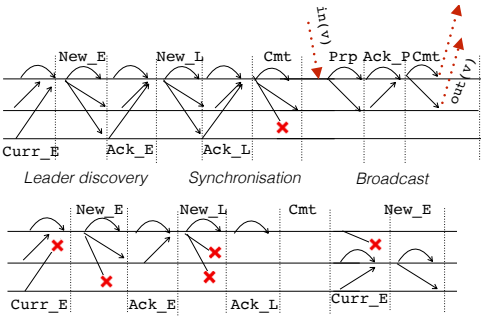

Fig. 2: Executions of Zab-HO.

ter `Broadcast` although initially all processes participate in the leader election.

An established leader and its followers execute a `Broadcast` for this epoch. Processes that do not agree on the leader skip forward to the next epoch. In `Broadcast` the leader accepts requests from the client, and broadcasts them to the replicas. If a process loses connection to its leader, it returns to `Discovery` for the next epoch leader election.

Syntax. Similar to the classic HO model, each CompHO-machine is composed of a set of local variables, an initialization operation `init`, and a non-empty sequence of rounds, called phase. Each process executes in a loop the sequence of rounds defined by the phase. Rounds are communication closed, that is all messages not received within the same round are lost. Each round is composed of a `send` function, that sends messages, followed by an `update` function, which modifies the local state of a process.

A CompHO-machine differs from the classic HO model in the `update` function. A process may call another machine and block until the other machine returns. An `update` may call at most one machine 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. We denote a set or processes that call the same machine by A (active). For the main machine, A is the entire set of processes.

The `init` and `send` operations are assumed to terminate within a number of steps that depends on the number of processes and the input values of the initialization function. The same holds for `update` unless it calls another machine in which case its termination is predicated by a termination of the called machine. Within the called machine, the statement `out_internal` marks the end of computing locally the returned values, and their return to the caller. The computation of the called machine continues until the instruction `exit` is reached.

Lockstep semantics. Each process has a variable interpreted over sets of processes, called the HO-set that captures asynchrony and faults. The messages received by a process p in round r, are the messages that were sent to p by the processes in its HO-set. At the beginning of each round, HO-sets are non-deterministically modified by the environment.

Assuming a finite, non-empty set of n processes P, the state of a HO machine is represented by the tuple $\langle SU, s, r, msg, A, \mathsf{HO}\rangle$ where: $SU \in \{Snd, Updt\}$ indicates if the next operation is send or update; $s \in [P \to V \to \mathcal{D}]$ stores the process local states; $r \in \mathbb{N}$ is the counter for the executed rounds; $msg \subseteq 2^{P,\mathsf{T},P}$ stores the messages which are in transit, where T is the type of the message payload; $A \subseteq 2^P$ stores the processes

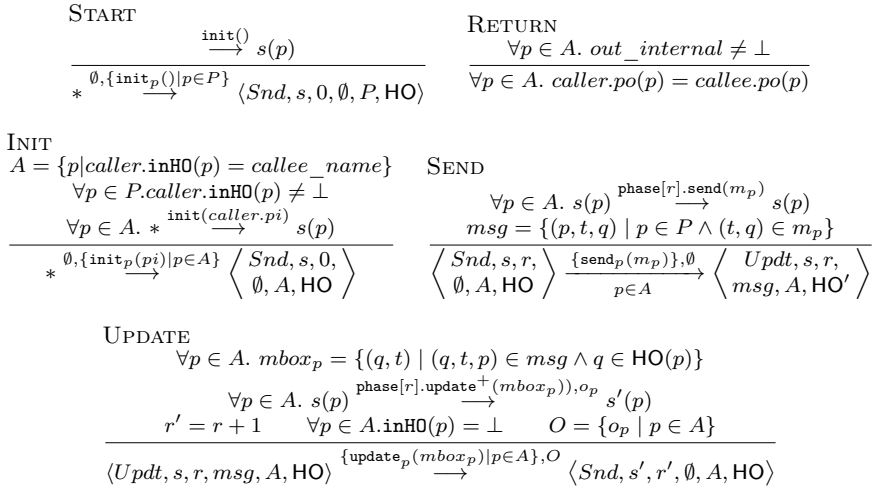

Fig. 3: CompHO semantics.

which are started the current machine; $\mathsf{HO} \in [P \to 2^A]$ evaluates the HO-sets for the current round.

The semantics is shown in Figure 3. Initially the system state is undefined, denoted by $*$. The first transition calls the `init` operation of the main HO machine on all processes (see START in Fig. 3), initializing the state: The round is 0, no messages are in the system. 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 by the environment to be a subset of A. In an UPDATE step, messages are received and the `update` operation is applied in each process. A message is lost if the senders identity does not belong to the HO set of the receiver. The set of received messages is the input of `update`. The `update` operation might produce an observable transition o_p. At the end of the round, msg is purged and r is incremented by 1.

Synchronized HO machine calls. Locally, os some process p, the semantics of HO-machine calls is defined as for procedure calls, where the call of process p is synchronized with the `init` operation of the callee on p (matching the values of the input parameter), and the instruction `out_internal` returns the values computed by the callee. Globally, the INIT rule defined in Fig. 3, forces all processes calling the same machine in the same round, to perform their `init` operation synchronously. Similarly the RETURN rule ensures that all returns from the same HO-machine happen synchronously.

To express global synchronization the rules use two auxiliary variables `inHO` and `outHO`. `inHO` takes values in the set of machine names, `#`, or *nop*. A call to a machine sets the `inHO` variable to the name of that machine. If by the last statement of `update`, no machines where called by the process, the semantics of the last statement sets `inHO` to *nop*. For each called machine, the set of processes A that have called it is set in the `init` operation according to the information in the `inHO` variables of its caller (see Fig.3 INIT).

The variable `outHO` takes values in $\{\perp, \texttt{done}\}$ where $\perp$ which is its initial value. Per process, `outHO` changes value only once during the execution of an HO-machine, when the instruction `out_internal` is reached. For a HO machine call to return to its caller, all processes that made the call must have set `outHO` to `done` (see Fig.3 RETURN).

Definition 1 (HO semantics). *Given an HO protocol $\mathcal{P}$ and a non-empty set of processes P, the semantics of $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]$, is defined by the set of executions of the transition system in Figure 3.*

Environment assumptions. Strong consistency problems, such as, primary backup, are not solvable in asynchronous networks with faults [11]. Therefore, algorithms make assumptions on the network and the faults in order to progress, given typically in english. The HO model enables their formalization by LTL formulas over the HO sets. For example, `Zab-HO`, is safe under any network assumptions but for progress requires that infinitely often there exists a sequence of six rounds, corresponding to a phase of `Discovery` followed by a phase of `Broadcast` where the HO-set leader is greater then $n/2$. The execution prefix in Fig. 2 satisfies this property, while the one in Fig. 5 does not.

3 Building HO-machines from asynchronous protocols

We define a procedure that builds a CompHO-machine from an asynchronous protocol. First we fix a class of asynchronous protocols in Sec. 3.1. In Sec. 3.2 we characterize asynchronous protocols that have an equivalent CompHO semantics. Finally in Sec. 3.3 we define a code-to-code rewriting.

3.1 Asynchronous systems

We consider distributed systems where all processes execute the same code (modulo process identifiers). The code uses local variables, denoted `Vars`, which are separated into *algorithm variables* `AVars`, evaluated over

```
1   p := ⊥ a := ⊥ h := <> zvid := ⊥
2   while (true){//@lab,p
3     leader := leader()
4     lab := Curr_E
5     send (p,lab) to leader
6     if (leader){ mbox := ∅
7       while(|mbox|<n/2 && !retry){
8         mbox := ∪ receive _Curr_E(m,p))
9         retry := {T/F}}
10      if (retry) continue;
11      Q := mbox.senders
12      p = max({m.1| m ∈ mbox}) + 1
13      lab = New_E
14      send (p,lab) to Q
15      }
16    lab := New_E
17    mbox := receive _New_E(leader,p)
18    if (mbox = ∅) continue
19    p := mbox.m.1
20    lab := Ack_E
21    send (lab,p,a,h) to leader
22    if(leader){ mbox := ∅
23      while(|mbox|<n/2 && !retry){
24        mbox := ∪ receive _Ack_E(Q);
25        retry := {T/F}}
26      if(retry) continue
27      h := max(mbox.payload)
28      Q := mbox.senders
29      lab := New_L
30      send (lab,p,h) to Q
31    }
```

Fig. 4: Asynchronous Zab

values of basic data types, e.g., enumerate, int, int$*$, and *reception variables* `RVars`, evaluated over sets/arrays of values of record types, where the record types corresponds to message payload types. The latter record types are denoted by $\mathbb{T}$. Given a variable `m` of type $\mathsf{T} \in \mathbb{T}$, `m.j` denotes the j^{th} component of the record type T. The protocol defines an interface for the communication with the client, denoted in the following `in`, `out`.

.

Fig. 4 shows an extract from `Zab-HO Discovery` in Fig 1. Like in the HO version, processes send their current epoch number `p`, to the leader defined by an oracle. A leader waits for $n/2$ messages and sets its epoch number `p`, to the maximum received value (Lines 6-12). Then, a leader sends the new epoch number to all the processes it received messages from (the Quorum `Q`)(Line 13). The code is structured in two loops, the outer one implementing the leader election algorithm, and an inner where the previously established leader starts accepting new requests. The remainder of the code is similar and therefore not given in Fig. 4.

The time-out for waiting for a message is captured by the non-deterministic assignment to the variable `retry`. When a follower does not receive a message from the leader, it skips to the next iteration of the loop (trying to elect another leader). Similarly, the leader jumps to the next iteration if it does not receive enough messages. Fig 5 shows such a behavior.

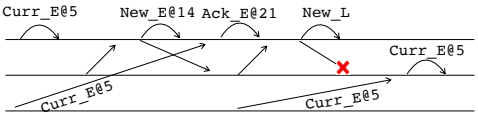

Fig. 5: Executions in Zab.

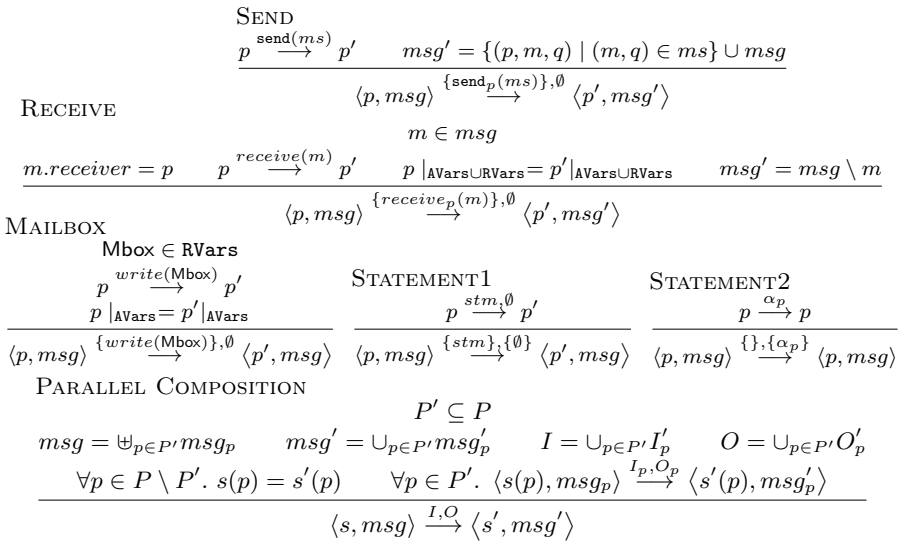

Fig. 6: Semantics of asynchronous protocols. PARALLEL COMPOSITION is defined over global states. All the other rules are defined over local states.

The semantics of `receive`(see Fig. 6) allows any message to be received, modulo a check on the receiver's identity. However, all programs filter the received messages, w.r.t. their payloads. For example, the `receive` at line 8 in Fig. 4 waits for messages of type (`int`, `enum`) where the first component equals `Curr_E`. We assume that these filters only read the algorithms variables. Also, we assume that when a process waits for multiple messages, it does it in a reception loop, where only the reception variables are written (lines 7-9). The only message payloads that can be read by the algorithms variables are those stored in the reception variables. More precisely, we assume that locally a processes goes through the following sequence of transitions $\mathsf{Statement}^*(\mathsf{Send}^*\ (\mathsf{Receive}\ \mathsf{Mailbox})^*\ \mathsf{Statement}^*)^*$, whose semantics is given in Fig. 6.

Formally, the state of a protocol $\mathcal{P}$ is a tuple $\langle s \uplus s_b, msg\rangle$ where: $s \in [P \to \mathtt{Vars} \cup \mathtt{Loc} \to \mathcal{D}]$ is a valuation of the variables in $\mathcal{P}$ where the program location is added to the local state; s_b is a local buffer used to store delivered but unused messages; $msg \in [\mathbb{T} \to (P, \mathsf{T}, P, \mathbb{N}) \to \mathbb{N}]$ is the multiset of messages in transit. Fig. 6 defines the local transition system of a process, and the system semantics which is the asynchronous parallel composition of the local transition systems.

The SEND rule states that the messages sent by a process are added to the global pool of messages msg. The RECEIVE rule defines message reception. It affects only the message buffers and auxiliary variable (which are scoped in the reception loop). The MAILBOX rule defines a write to a reception variable, denoted `Mbox`. All the other statements have the usual semantics. The rule STATEMENT1 says that statements corresponding to interface operations, denoted α_P, do not modify the local state, while STATEMENT2 says that all the other statements have the usual semantics. While we omit the rules that describe the fault model, we consider that processes, do not recover, and crashed process do not modify the global state. Messages can be duplicated and dropped by the network.

The set of executions of a protocol $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]$, is the set of executions of the transition system in Fig. 6.

3.2 Reduction

We introduce conditions over the local executions of a protocol, and prove them to be sufficient to have an "equivalent" (indistinguishable) CompHO–semantics. We start by defining relations over executions; first the causality relation: Given a protocol $\mathcal{P}$ and an execution $\pi \in [\![\mathcal{P}]\!]$, two transitions t_1 and t_2 are causally ordered in π, denoted $t_1 <_\pi t_2$, if they are ordered by the local program order, or if t_1 is a send and the message sent in t_1 is written in the mailbox of the process that executes the receieve t_2.

Definition 2 (Indistinguishability). *Given two executions π and π' of a protocol $\mathcal{P}$, a process p cannot distinguish locally between π and π', w.r.t. a set of variables W, denoted $\pi \simeq_p^W \pi'$, iff 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 (1) no process can distinguish between them, i.e., $\forall p.\ \pi \simeq_p^W \pi'$, and (2) the causality order between send and receives is the same in both executions.*

We consider protocols where the local execution of each process i goes through a sequence of layers $L_i^1, L_i^2, \ldots$, so that inter-process communication occurs only within the layer, i.e., for processes i and j, if i in layer L_i^k sends a message to j, then j "receives" the message in layer L_j^k, a.k.a., communication closed layers [10]. We consider this "receive" operation as a local action that takes a message that is delivered to the process as input, and determines based on the state of the process whether the message should be dropped, or its content should be written to some variable

of the process. Dismissing old messages is implicitly done in all the state machine replication protocols we are aware of. However, many algorithms react on messages from "future" rounds, that is, if a process is in layer k and receives a message from a process in layer $k' > k$, then it concludes that its local computation is behind the local computation of other processes, so that it skips forward to k'. In our theory, this behavior does not break communication closure as long as the receptions are followed by a local transition that skips forward to that layer. We shall prove that a protocol of that structure have an indistinguishable CompHO-semantics, where an HO round corresponds to a layer.

Tag annotations. In the systems we consider, we observed that layers are uniquely identified by values of a subset of the protocol's variables, and the layers order coincides with the lexicographic order on the product of the domains of these variables. Thus, we introduce *tag annotations*, that

1. associate each control location with a subset of (algorithm) variables, postulating that the values of those variables define the layer where the execution of the corresponding instruction belongs to, and
2. map each message type to some of these variables, postulating that each message sent or received is tagged with the layer it belongs to.

Definition 3 (Tag annotation). *Given a protocol $\mathcal{P}$, a* tag annotation *is a tuple* $(\mathtt{SyncV}, \prec, \mathtt{tags}, \mathtt{tagm})$ *where:*

- $\mathtt{SyncV}$ *is a subset of the protocols variables and* $\prec$ *is a total ordered over* $\mathtt{SyncV}$*,*
- $\mathtt{tags} : \mathtt{Loc} \to 2^{\mathtt{SyncV}}$*, is a function that annotates each control location with variables in* $\mathtt{SyncV}$*, and*
- $\mathtt{tagm} : \mathbb{T} \to [\mathsf{T} \overset{injective}{\rightharpoonup} \mathtt{SyncV}]$ *is an injective partially defined function, that maps a component of a record type* T *to a variable in* $\mathtt{SyncV}$ *of the same type.*

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

- $\llbracket\mathtt{tags}\rrbracket : \Sigma \to [\mathtt{SyncV} \to \mathcal{D}]$*, 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* $\llbracket\mathtt{tags}\rrbracket_s = (v_1, \ldots, v_{|\mathtt{SyncV}|})$*, with* $v_i = \llbracket\mathsf{x_i}\rrbracket_s$ *if* $\mathsf{x_i} \in \mathtt{tags}(\llbracket\mathsf{pc}\rrbracket_s)$ *otherwise* $v_i = \bot$*, where* $\mathsf{x_i}$ *is the* i^{th} *variable in* $\mathtt{SyncV}$ *w.r.t.* $\prec$*, and* pc *is the program counter.*
- $\llbracket\mathtt{tagm}\rrbracket : \mathbb{T} \to \mathcal{T} \to [\mathtt{SyncV} \to \mathcal{D} \cup \bot]$ *is a function that for any value* $m = (m_1, \ldots, m_t)$ *of record type* T *associates a tuple* $\llbracket\mathtt{tagm}\rrbracket_{m:\mathsf{T}} = (v_1, \ldots, v_{|\mathtt{SyncV}|})$ *with*

- $v_i = m_j$ *if* $\mathsf{x_i} = \mathtt{tagm}(\mathsf{T})(j)$*, where* $\mathsf{x_i}$ *is the* i^{th} *variable in* `SyncV` *and* $\mathtt{tagm}(\mathsf{T})(j)$ *is the mapping of the* j^{th} *component of the record type* T*,*
- $v_i = \bot$*, otherwise.*

For our example Zab, the natural tag annotation associates all control locations of the outer loop with `p`, the epoch number, and `lab`, a label, with $\mathtt{p} \prec \mathtt{lab}$. Filtering out the message payload, messages of type $(\mathtt{int}, \mathtt{enum})$ and $(\mathtt{int}, \mathtt{enum}, \mathtt{int}*)$ are mapped to $(\mathtt{p}, \mathtt{lab}, \bot, \bot)$ (its a quadruple because the inner loop that executes broadcast is also annotated by two different variables). A message $m = (3, \mathtt{Curr\_E})$ is evaluated by `tagm` into $(3, \mathtt{Curr\_E}, \bot, \bot)$. $\mathtt{tags}(@8)$ in a state s evaluates into $(3, \mathtt{Curr\_E}, \bot, \bot)$ is the value of the variable p is 3 in s. Note that `Curr_E` is the only value `lab` can take at location @6. In the CompHO a layer corresponds to the phase and the round number. Therefore each program location should be annotated with an even number of variables. Intuitively, instructions tagged with the same variables belong to the same machine. In case of nested machine calls, the code belonging to called machine is annotated also with the phase and number of its caller.

We now characterize tag annotation tag that we will prove to identify communication closed protocols.

Definition 4 (Synchronization tag). *Given a program* $\mathcal{P}$*, an annotation tag* $(\mathtt{SyncV}, \mathtt{tags}, \mathtt{tagm})$ *is called* synchronization tag *iff:*

(I.) 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* $\mathcal{P}$*,* $[\![\mathtt{tags}]\!]_{s_0}[\![\mathtt{tags}]\!]_{s_1}[\![\mathtt{tags}]\!]_{s_2} \ldots$ *is a monotonically increasing sequence of tuples of values w.r.t. the lexicographic order.*

(II.) for any local execution $\pi \in [\![\mathcal{P}]\!]_p$*, if* $s \xrightarrow{send(p,m)} s'$ *is a transition of* π*, with* m *a value of some record type, then* $[\![\mathtt{tags}]\!]_s = [\![\mathtt{tagm}]\!]_m$ *and* $[\![\mathtt{tags}]\!]_s = [\![\mathtt{tags}]\!]_{s'}$*.*

(III.) for any local execution $\pi \in [\![\mathcal{P}]\!]_p$*, if* $s \xrightarrow{receive(m,p);write(\mathtt{Mbox})} s'$*, is a transition of* π*, with* m *a value of some record type, then*

- *if* $m \in [\![\mathsf{Mbox}]\!]_{s'}$ *then*
  - $[\![\mathtt{tags}]\!]_s \leq [\![\mathtt{tagm}]\!]_m$ *if* $\mathtt{tagm}(T, [\![\mathtt{pc}]\!]_s)$ *totally defined,*
  - $[\![\mathtt{tags}]\!]_s = [\![\mathtt{tagm}]\!]_m$*, otherwise.*
- $m \notin [\![\mathsf{Mbox}]\!]_{s'}$*,* $s = s'$

(IV.) for any local execution $\pi \in [\![\mathcal{P}]\!]_p$*, if* $s \xrightarrow{\mathsf{stm}} s'$ *is a transition of* π *with* $[\![\mathtt{tags}]\!]_s < [\![\mathtt{tagm}]\!]([\![\mathsf{Mbox}]\!]_s)$ *then* $[\![\mathtt{tags}]\!]_{s'} = [\![\mathtt{tagm}]\!]([\![\mathsf{Mbox}]\!]_s)$*, where* $[\![\mathtt{tagm}]\!]([\![\mathsf{Mbox}]\!]_s) = max\{[\![\mathtt{tagm}]\!]_m \mid m \in [\![\mathsf{Mbox}]\!]_s\}$*.*

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 layer numbers are not decreased by any local statement. Condition (II.) states that any message sent is tagged with a layer number that coincides with the layer of the current state. Condition (III.) states that any message received and stored is tagged with a layer number greater or equal than the current layer of the process. If a message contains more than a layer number, its tag must coincide with the tag of the state where it is received. Finally, (IV.) states if messages from future layers are stored by a process in its reception variable, any statement that is executed once the reception is over must make the process jump to the maximal layer it received a message from.

Reductions. 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. In several steps, we are going to reduce an asynchronous execution ae to an HO execution se in which all processes go through the same sequence of states (ignoring the variables where messages are stored). The first reduction considers the receive statements. If the local execution is of the form $\pi = \ldots s_i^p, receive_i, s_{i+1}^p, receive_{i+1}, s_{i+1}^p, \ldots$, in the asynchronous execution, the two receive actions can be interleaved by actions of other processes. Following the theory by Lipton [19], 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. We thus get an asynchronous executions with blocks that consist of several receives (possibly just one receive) followed by stm. We will subsume such a block by a single (atomic) action *Receive*, and by abuse of notation use the same symbol $[\![\mathcal{P}]\!]$ for asynchronous semantics with the atomic Receive.

Lemma 1. *Given a program $\mathcal{P}$ if there is a synchronization tag* $(\mathtt{tags}, \mathtt{tagm})$ *for $\mathcal{P}$, then $\forall \mathsf{ae} \in [\![\mathcal{P}]\!]$, if $\mathsf{ae} = \ldots S_{i-1}, A_i^p, S_i, A_{i+i}^q, S_{i+i} \ldots$, and $[\![\mathtt{tags}]\!]_{s_i(p)} > [\![\mathtt{tags}]\!]_{s_{i+1}(q)}$, then $\mathsf{ae}' = \ldots S_{i-1}, A_{i+i}^q, S_i', A_i^p, S_{i+i} \ldots \in [\![\mathcal{P}]\!]$. Further $\mathsf{ae}', \mathsf{ae}$ are indistinguishable w.r.t. all protocol variables, i.e., $\mathsf{ae}' \simeq \mathsf{ae}'$.*

Proof. From (I.) follows that $p \neq q$, so that swapping cannot violate the local control flow. As $p \neq q$, if A_{i+i}^q is a send or a stm, the action at p has no influence on the applicability of A_{i+i}^q to S_i. The only remaining case is that A_{i+i}^q is a *Receive*. Only if A_i^p 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 lemma, we obtain an asynchronous execution $\mathsf{ae}' = \ldots S_{i-1}, A^p_i, S_i, A^q_{i+i}, S_{i+i} \ldots$, where for each i and any two processes p and q, $[\![\texttt{tags}]\!]_{s_i(p)} \leq [\![\texttt{tags}]\!]_{s_{i+1}(q)}$. The asynchronous execution ae' is thus a decomposition of ae into layers. We now do a reduction within the layers. The local code within each layer is structured in that first are send, then Receive, and then other statements. By the arguments from [3], the send actions are left movers with respect to all other operations, Receive actions are left movers with all statements except sends. We thus obtain ae'' where within a layer all send actions come before all Receive actions, which come before all other actions. As in [3], we can now subsume all send actions within a layer to a global send step in the layer. The same we do for receive actions. As it is sufficient to check states only when the tags change, we also subsume the other actions. We thus obtain a *synchronous execution* se, where the messages received in a Receive statement correspond to the HO sets; if a process did not receive a messages this corresponds to an empty HO set which results in skipping this round. As observed in [3], such a reduction maintains *local properties* that are sufficient to express specifications of, e.g., replicated state machines. Fig. 5 shows an asynchronous execution that is indistinguishable from the second HO execution in Fig. 2.

Theorem 1. *Given a program $\mathcal{P}$ if there exists a synchronization tag $(\texttt{tags}, \texttt{tagm})$ for $\mathcal{P}$, then $\forall \mathsf{ae} \in [\![\mathcal{P}]\!]$ there exists an indistinguishable execution that belongs to the HO semantics.*

Local properties are those that are tolerant to local stuttering. Indistinguishability is an equivalence relation over traces, and local properties are closed under indistinguishability. They have been formalized in [3,8].

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

Given an annotation tag, our conditions from Definition 4 can be checked on the local code. They translate into assert statements and transition invariants over the reception and synchronization variables.

3.3 Code to code translation

Theorem 1 ensures that communication closure preserves local behaviors under reduction, but does not formally imply a code to code translation. We introduce a rewriting algorithm `make-HO`, that takes as input a protocol $\mathcal{P}$ annotated with a tag $(\texttt{SyncV}, \prec, \texttt{tags}, \texttt{tagm})$ and produces a CompHO-machine $HO(\mathcal{P})$ that preserves all local behaviors, or aborts.

The rewriting `make-HO` tries to generate an HO machine for each loop, where in the case of nested loops, the machine associated with the outer loop calls the machine associated with the inner loop. For this, the tagging function must define two unique synchronization variables for each loop, corresponding to the phase and the round number of the machine associated with the loop. An inner loop will be annotated also with the synchronization variables of its outer loops. If this is not the case the algorithm aborts. To match loop iterations with phases of an HO machine, the variable representing the phase number must be increased in each iteration, otherwise the rewriting procedure aborts (if phases expend over loop iterations more involved analysis is required to identify them. However we did not encountered an example to motivate us to explore this). For the programs we have rewritten, this check was trivial as the required variables are explicit in the code.

We now split a loop body to rounds. Let us fix a loop in the following. If the round variable does not take values in a bounded domain the rewriting aborts. Otherwise, let $l_1, \ldots, l_k$ be the values `round` can take, with $l_i < l_j$ iff $i < j$. The rewriting needs a marking of the control locations where the round variable change values (from the definition of a tag the value can only increase). This marking corresponds to the rounds switch points. Again this was immediate to identify on our examples, because messages are tagged with labels from a finite domain (e.g., `Curr_E`, and `cepoch`), defining the exact values of the rounds. If the round variable changes values more than k times in a loop body the rewriting aborts (one loop iterations goes over multiple phases). With this marking, `make-HO` now takes care of the branching within the loop body:

Decomposing branches into code for rounds. For each path π in the loop body, `make-HO` computes a partition $\mathcal{R}_\pi$ of its program locations. Each $R \in \mathcal{R}_\pi$ contains code for one round, i.e., a sequence of locations starting at the next location after a marked control location, and ending either at the next marked location, or at the end of the loop body. For simplicity, we assume each element of these partitions contains at most one send instruction (possibly a "send to all" instruction) and at most one reception

loop. These partitions are totally ordered, by the order over the program locations in the control flow graph.

Constructing a round from paths in $\mathcal{R}_\pi$ Let us first assume that along each path there is code for each round, i.e., the partition associated with each path is of size k. Then, any element R of $\mathcal{R}_\pi$, for any path π, maps uniquely to values in $\{l_1, \ldots, l_k\}$. `make-HO` groups all $R_i^\pi \in \mathcal{R}^\pi$, for all π, into a global round l_i as follows:

Send. First, the global `send` operation of round l_i is a sequence of if statements, one for each path π, each ending with the send operation in R_i^π, if present. (The conditions $cond_\pi$ in the if of a `send` are built as in the case of `update` except (3), see below).

Receive. In each R_i^π, `make-HO` replaces the reception loop with one atomic check over the received set of messages: `while(cond && !retry) { m=receive (); if(m ..) mbox := ..}` is replaced with the HO instruction `if (!cond)`.

Update The `update` operation of round l_i, has one if statement per path π, whose condition $cond_\pi$ is (1) the `round` variable equals l_i, (2) the conjunction of all conditions on the path π from the loop entry to the first instruction of R_i^π, and (3) the condition on the received messages from the reception loop. The code on the if branch of this conditional is the sequence of statements after the reception loop in R_i^π.

If there is no code for each round in each path, then there are two cases. If this is due to a `continue` statement — some rounds are skipped over — this encodes implicit branching. In this case the missing code can be filled with no-ops. If this is not the case `make-HO` aborts.

Theorem 3. *Given an asynchronous program $\mathcal{P}$ annotated with a tagging function* $(\texttt{SyncV}, \prec, \texttt{tags}, \texttt{tagm})$ *if the algorithm `make-HO` applied $\mathcal{P}$ returns an HO machine $HO(\mathcal{P})$, then the executions of $HO(\mathcal{P})$ are indistinguishable from the executions of $\mathcal{P}$, w.r.t. the algorithm variables, and $HO(\mathcal{P})$ preserves all the local properties of $\mathcal{P}$.*

4 Conclusion

Current verification techniques for state machine replication (local properties) are monolithic from the verification perspective, i.e., they apply general purpose verification techniques to selected protocols. For instance, IronFleet [13] uses mechanized proofs and IVY uses encodings in extensions of EPR [27]. In search for modular verification, we distinguish two

approaches: the ones based on the atomic objects abstraction [32], that use verification techniques developed for shared memory for distributed systems, and the ones based on synchronous abstractions [5,6,8]. We chose to explore synchronous abstractions because they match the intuition of the system designers, and we are interested in exploring domain-specific verification techniques for distributed systems. Thus, we have introduced conditions on the local code that ensure the existence of an equivalent synchronous semantics. Future works consists in automating the reduction scheme we introduced using automated verifiers like IVY [27] or CL [7].

One aspect of such distributed systems is recovery: a process that is late can asks the leader at any time for the latest state. This communication is not closed as considered in this paper. However, recovery has no clear specification and has no impact on the qualitative (safety and liveness) specification of the system; it is a matter of performance. We thus suggest that recovery should be verified independently from the verification of the running system, and leave this challenge for future work.

References

1. Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. Parameterized model checking of synchronous distributed algorithms by abstraction. In *VMCAI*, pages 1–24, 2018.
2. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.
3. Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A reduction theorem for the verification of round-based distributed algorithms. In *RP*, volume 5797 of *LNCS*, pages 93–106, 2009.
4. Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
5. Ankush Desai, Pranav Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In Andrew P. Black and Todd D. Millstein, editors, *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*, pages 709–725, 2014.
6. Ankush Desai, Sanjit A. Seshia, Shaz Qadeer, David Broman, and John C. Eidson. Approximate synchrony: An abstraction for distributed almost-synchronous systems. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 429–448, 2015.
7. Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In Kenneth L. McMillan and Xavier Rival, editors, *VMCAI*, pages 161–181. Springer, 2014.
8. Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

9. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, April 1988.
10. Tzilla Elrad and Nissim Francez. Decomposition of distributed programs into communication-closed layers. *Sci. Comput. Program.*, 2(3):155–173, 1982.
11. Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
12. Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 143–152, 1998.
13. Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, 2017.
14. Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256, 2011.
15. Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.
16. Leslie Lamport. Generalized consensus and paxos. Technical report, March 2005.
17. Gérard Le Lann. Asynchrony and real-time dependable computing. In *WORDS*, pages 18–25, 2003.
18. Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *POPL*, pages 357–370, 2016.
19. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
20. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.
21. Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff Bounds for Consensus Algorithms. In *CAV*, pages 217–237, 2017.
22. Andre Medeiros. ZooKeeper's atomic broadcast protocol: Theory and practice. Technical report, 2012.
23. Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.
24. Brian M. Oki and Barbara Liskov. Viewstamped replication: A general primary copy. In *PODC*, pages 8–17, 1988.
25. Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 305–319, 2014.
26. Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1(OOPSLA):108:1–108:31, 2017.
27. Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1(OOPSLA):108:1–108:31, 2017.
28. Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630, 2016.
29. Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015.

30. Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In *PLDI*, pages 599–613, 2016.
31. Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for Change in a Formal Verification of the RAFT Consensus Protocol. In *CPP*, pages 154–165, 2016.
32. Álvaro García-Pérez, Alexey Gotsman, and Yuri Meshman. Paxos consensus, deconstructed and abstracted. In *ESOP*, 2018. to appear.