A Sequentialization Procedure for Fault-Tolerant Protocols

Cezara Drăgoi, Patricio Inzaghi Pronesti

To cite this version:

Cezara Drăgoi, Patricio Inzaghi Pronesti. A Sequentialization Procedure for Fault-Tolerant Protocols. 2022. hal-03796317

HAL Id: hal-03796317
https://inria.hal.science/hal-03796317

Preprint submitted on 4 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Sequentialization Procedure for Fault-Tolerant Protocols*

Cezara Drăgoi[2,3] and Patricio Inzaghi Pronesti[1,2][0000−0001−9404−7077]

[1] DI ENS, Université PSL - CNRS, Paris 75005, France pinzaghi@ens.fr
[2] INRIA Paris
[3] Informal Systems

Abstract. We introduce a sequentialization procedure for fault-tolerant protocols that takes as input a Distal program and produces a sequentialized counterpart as output. The sequentialization procedure captures a representative subset of the behaviors of the input system and is easier to model check; for a broad class of protocols, it captures a representative for every behavior. Our notion of sequentialization-equivalence extends the well-studied notion of communication closure in distributed protocols, which relates asynchronous and synchronous executions. We implemented our sequentialization and applied it to verify several consensus protocols, including ZooKeeper Atomic Broadcast, and Raft, using the P framework. We considered P models that include critical safety bugs present in implementations and known by the community. The P model checker found these bugs only when using the sequential model but not in the original asynchronous counterparts.

1 Introduction

Correctly designing and implementing fault-tolerant distributed systems is hard. Many bugs appear both at the protocol and at implementation level and the design of effective tools to find bugs early is an important challenge in formal methods. One successful direction of research is the development of high-level Domain Specific Languages designed for facilitating verification or testing of distributed systems, together with efficient verification and testing tools. Notable examples are Ivy [1] Promela/Spin [2], Coyote, [3] and P [4]. The bane of all these tools is state-space explosion: as the complexity of the protocols grow, systematic exploration can only cover a minuscule portion of the state space.

We show how systematic testing of fault-tolerant distributed protocols can be improved by using the *sequentialization* approach, which produces a sequential version that captures an interesting subset of all behaviors. The sequential version has fewer behaviors, allowing systematic testing tools to scale better, but any bug in the sequentialization is also a bug in the original protocol. For shared memory systems, sequentialization techniques have proved effective in increasing

* Supported by: French National Research Agency ANR project SAFTA (12744-ANR-17-CE25-0008-01).

the number of bugs found in concurrent programs [5, 6]. However, existing sequentialization techniques for message passing protocols are either manual [7], or consider only non-faulty protocols [8], or prove equivalence between given asynchronous and sequential protocols, given both protocols as well as complicated inductive invariants [9]. In contrast, we propose a new automated sequentialization technique for fault-tolerant protocols that uses minimal annotations.

Our sequentialization uses the notion of communication-closure [10], which identifies the conditions under which a set of asynchronous executions is equivalent to one *round-based* execution. In round-based executions, processes proceed in lock-step: all processes *send* messages, *receive* (possibly a subset of) the sent messages, and *update* their state based on the received messages. There are no delayed messages: a message that is not received after it was sent (a.k.a. *rendez-vous*) is lost forever. Round-based executions have no interleaving across rounds and faults are localized within the round boundaries. Compared with asynchronous protocols, they have exponentially fewer behaviors.

We define a sequentialization procedure for protocols written in Distal [11], a DSL for fault-tolerant systems aligned with the syntax of text-book protocols but also with the syntax of P [4], a modeling language used for writing and testing state machine models in industry (roughly, P embeds Distal constructs). First, we compute a round-based representation of Distal protocols, building on the procedure in [12] that takes an asynchronous program (from an appropriate class) and computes an equivalent round-based representation. We extend their procedure to handle common features required by asynchronous programs such as high-level primitives for message passing. Second, we propose a sequentialization of the round-based representation that is complete for arbitrary networks, like the ones required by Paxos [13] or Raft [14], but also for stronger network assumptions, as required by Ben-Or or 2PC [15].

To sum up, we take a Distal program as input and produce as output a new Distal program that is the sequentialization of the input. Since the sequentialization has fewer behaviors, testing tools have an easier time finding bugs.

We implement and evaluate our algorithm using the P framework [4]. We applied the sequentialization on P models for Paxos, Raft, Ben-Or, ViewStamped, UniformVoting, and 2PC. Running P's testing tool on their sequential versions uncovered subtle bugs that were not always found in the original asynchronous P model (due to state explosion). Most notable bugs found exclusively in the sequentialization were in Paxos and Raft. We modeled a version of Paxos that captures the bug scenario in ZAB [16, 17]. The bug is a violation of agreement, where replicas disagree on the order of the commands executed by the replicated state machine and is used as a running example. We modeled the protocol that handles the cluster's configuration in Raft [18, 14]. The bug is a safety violation, where processes disagree on the replicas that run the state machine. To catch it, the sequentialization of Raft takes into account process creation.

Related Work Communication closure has been used in verification [12, 19] and testing [17]. In [19, 12] the authors define a transformation of an asynchronous protocol into a synchronous one, that is further verified using Hoare-

style of reasoning ([19] uses communication closure implicitly). Both works consider a highly-constrained input language, chosen to suit the requirements of the transformation procedure. For example, they do not consider high level message passing primitives. In contrast, we consider Distal protocols as input. Distal is an established language in the theoretical community and in industry (in the form of P). Therefore, our method makes transformations based on synchronizations accessible to a wider audience. Moreover, we define a sequentialization procedure that uncovers bugs which are not found by state-of-the art testing tools for asynchronous protocols. There are many verification and testing tools for sequential programs and shared memory systems [20, 21], that could be applied on the sequentialization computed by our method, contrary to the output of [12, 19], where tools for distributed synchronous protocols are not available.

The communication closure hypothesis has been empirically used in testing large scale systems models [17, 22, 23]. In [17] the authors start from an instrumented large scale system and explore a subset of its executions checking for violations. The current submission starts from a model of the program and proposes a more systematic and efficient exploration of the executions.

2 Overview

We illustrate our sequentialization procedure using the replicated state machine protocol in Fig. 1, inspired from Paxos [13]. Processes receive different commands, and the goal of the protocol is to make processes agree on a total order over a set of received commands, even when messages are lost or delayed. Each process maintains the log of commands it agreed on, e.g. *abcd*, which is visible to an external observer (line 30). The outputted log of any two processes must respect the prefix order over sequences. A violation of the prefix order, e.g., one process outputs a and another one outputs b, means that the two processes disagree on the first command to be executed by the machine. However, it is correct to have one process output a and another one output ab, it happens when the process outputting a is late and didn't learn yet the second command to be executed.

The protocol in Fig. 1 has a bug in line 9 which generates an execution violating the prefix order property. This bug is fixed by moving this statement to line 23. We choose this example because (1) testing it using P [4] did not find the bug, and (2) it is a simplified version of the bug[4] in the implementation of ZAB [16]. Using P on the sequentialization found the bug.

The protocol is written in Distal [11], an event-driven programming model with upon statements defining how the protocol reacts to receiving a message. The code given in Fig. 1 is executed by all processes[5] using the standard inter-

[4] https://issues.apache.org/jira/browse/ZOOKEEPER-2832

[5] This does not mean all processes go through the same sequence of states, because (1) local state updates based on the received messages and (2) processes might receive a different set of messages.

```
1:  init
2:    ballot = 0; log = ϵ;
3:    if primary(ballot+1) then
4:      ballot = ballot+1;
5:      m = PrepareMsg(ballot);▶ Prepare
6:      send m to ALL;
7:  while true do
8:    upon Prepare with m.ballot > ballot do
9:      last = ballot;
10:     ballot = m.ballot;
11:     promised = false;
12:     primary = m.sender;
13:     m = Ack(ballot, last, log); ▶ Ack
14:     send m to primary;
15:   upon Ack with m.ballot = ballot times n/2
    do
16:     log = longest_log(ballot); ▶ Ack
17:     log.add(newCommand());
18:     m = Propose(ballot, log); ▶ Propose
19:     send m to ALL;
20:   upon Propose with m.ballot ≥ ballot ∧
    ¬promised do
21:     ballot = m.ballot; ▶ Propose
22:     log = m.log;
23:     //Bugfix: last = ballot;
24:     m = Promise(ballot,log); ▶ Promise
25:     send m to ALL;
26:   upon Promise with m.ballot ≥ ballot ∧
    m.log = log times n/2 do
27:     ballot = m.ballot; ▶ Promise
28:     log = m.log;
29:     promised = true;
30:     output(log);
31:     if primary(ballot+1) then
32:       ballot = ballot+1; ▶ Prepare
33:       m = Prepare(ballot);
34:       send m to ALL;
35:   upon timeout() with true do
36:     if primary(ballot+1) then
37:       ballot = ballot+1;
38:       m = Prepare(ballot); ▶ Prepare
39:       send m to ALL;
```

Fig. 1. Simple Paxos protocol in Distal containing a bug (marked in red) where the `last` variable is updated too early. The ▶ marker denotes a new round in the code.

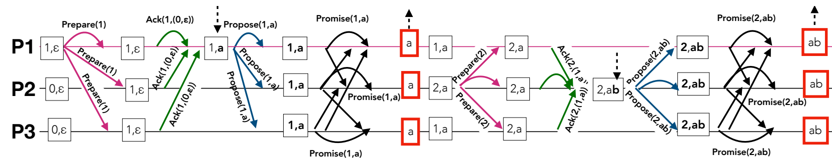

Fig. 2. An execution over two ballots where all messages are delivered.

leaving of steps executed by different processes. To communicate, processes use point-to-point or broadcast. Messages may be dropped of delayed.

Processes go through a sequence of ballots, and in each ballot they try to add a new command to their log. If enough messages are delivered, then the log is extended, otherwise they move on to the next ballot and retry, maybe with a different command. This is a leader-based protocol, where the function $primary(b)$ takes as input a ballot number b and returns the identity of the leader of the ballot, using for example a round-robin scheme. The leader is in charge of (1) starting a new ballot, (2) collecting logs of a quorum of processes, and selecting the longest most recent log out of the received ones, and (3) extending this log with a new command and proposing it to all processes in the network. All processes that receive the new log from the leader broadcast it. Finally, a process outputs a log when it learns that $n/2$ of its peers received the same log from the leader. Fig. 2 shows an execution of the protocol, where all messages are delivered and all processes store a in their logs in the first ballot, and extend the log with b in the second ballot. Figs. 4 and 3 show other executions where the messages send by P3 are delayed or dropped. A naive and inefficient sequentialization scheme

produces a sequential behavior for each interleaving. For example it generates two different sequentializations, one where first P1 sends a `Prepare` message and then P2, and the other way around. Moreover, from one interleaving multiple sequential executions are possible depending on which messages are delayed, lost, or delivered. For example, there will be three sequential executions one when P3 receives the `Prepare` message, one when it is lost, and one when it is delayed.

We propose a more efficient sequentialization procedure, which produces one non-deterministic sequential protocol that is equivalent to an asynchronous one. This equivalence relation is that processes go through the same sequence of states modulo *stuttering* (i.e., consecutive repetition of equivalent states).

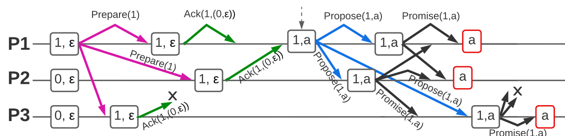

Fig. 3. An execution where all messages sent by P3 are lost.

The sequentialization exploits *the round structure* of the protocol following the approach based on communication-closure [17, 12]. The asynchronous semantics allows an arbitrary interleaving of steps of different processes, executed over a non-deterministic network that can delay, drop, or reorder messages. However, this semantics includes a set of synchronous executions, where all messages are delivered in-time, e.g., Fig. 2. Observing this happy path, we see that the protocol is structured in four rounds, executed in the same sequence in each ballot. Each round only sends/receive one type of message. Processes update their state using only messages of this type.

In the first round the leader sends a `Prepare` message containing the number of the leading ballot. The processes that receive its message update their ballot, if the leader leads a higher or equal ballot. In the next round, processes reply to the leader with an `Ack` message that contains the leader's ballot, the current log stored by the process, and the value of the last ballot the process participated in. If the leader receives more than $n/2$ `Ack` messages it selects the longest log out of the one coming from processes that participated in the most recent ballot.

In the next round the leader extends this log and broadcasts a `Propose` message with the current ballot and the new log. In the final round all processes that receive the new proposed log, broadcast this log and the current ballot number in a `Promise` message. A process that receives more than $n/2$ `Promise` messages with the same log and the current ballot outputs that log.

Faulty executions respect the round structure as well: locally, processes respect the ballot order and the round order within a ballot. Fig. 3 shows an execution of the first ballot where sent messages by P3 are lost. To transform it into a synchronous execution we use the fact that any send, receive, or update of some round r, it's a *left mover* [24] w.r.t. actions of other processes from rounds higher than r and a right mover w.r.t. actions from earlier rounds.

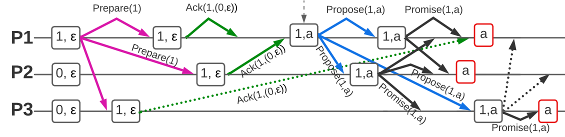

Fig. 4. Messages sent by P3 are delayed. Dotted lines represent stale messages that are not used by the receiver.

The execution in Fig. 4 respects the round structure, even if the messages sent by P3 are delivered after P1, and P2 moved past the round the messages coming from P3 were sent for. In [10, 12] it's proved that a message received with a delay causes a violation of the round structure only if (1) the process is in a higher round and (2) the process use the message's payload to update its local state. In the considered execution, P1 and P2 are in the second ballot where the messages from P3 arrive, and they ignore all messages coming from the first ballot, like the ones sent by P3, therefore their reception does not cause a change of state in P1 and P2.

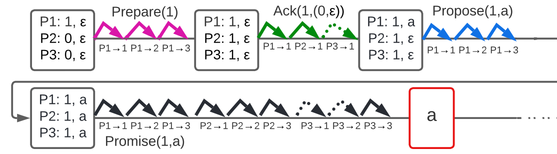

Fig. 5. Sequential execution equivalent to Fig. 3. Boxes represent the global state, arrows are messages (dashed are lost messages) and the color its round.

All executions of the protocol in Fig. 1 are equivalent to potentially faulty round-based executions, like those in Fig. 2, where messages can be lost but not delayed. Round-based executions impose a total order over actions performed by processes across rounds. The sequentialization maintains this order, and adds a total order over actions performed by processes within one round. Note that within one round there are only message chains of length at most one, and each process sends at most one message. Therefore the order in which processes send messages does not matter, all are equivalent and the sequentialization picks one. For each receive, it adds a non-deterministic choice modeling a message dropped by the network. Let us consider round-based executions where no messages are lost in Fig. 2. In this case, an equivalent sequential execution replaces any send and its matching receive by one assignment, and order them according to a chosen order over processes. In the presence of faults, the equivalent sequential execution consists only of those assignments corresponding to not dropped messages (Fig. 5).

In [12] and [19] the authors exploit the round structure for verification. They compute the synchronous version of the protocol (over a more restricted input). The resulting synchronous protocol is equivalent with the original asynchronous one in the absence of network assumptions, i.e., any message can be lost or de-

layed. When the protocol is correct under a network that meets a certain amount of reliability, e.g., Ben-Or, the synchronous protocol produced by these previous methods is an over-approximation of the asynchronous one. Since for testing over-approximations are not useful, in the presence of network assumptions, the sequentialization we propose introduces more restrictions over the number of messages that can be lost, by restricting the number of non-deterministic choices in the resulting sequential program.

In summary, we propose a method to obtain a non-deterministic sequential protocol, that is equivalent with an asynchronous one, where the equivalence relation is that processes go through the same sequence of states modulo stuttering. The sequentialization is precise for fault models commonly used in distributed protocols. As an intermediate step of the sequentialization we compute the round-based version of a Distal asynchronous protocol, where all executions are structured in rounds, and messages sent in a round are either received in the same round or lost, a.k.a., communication-closed protocols. For this step we extend the work in [12] to a more general input language and the procedure we propose uses lighter annotations where the user needs to specify the rounds only in the message types. The sequential protocol is non-deterministic because for each round it will consider all the possible sets of messages that can be lost in that round. The reduction from asynchronous to round-based to sequential preserves the sequence of states processes go through locally. This implies that at the global system level it preserves the so-called local properties which includes consensus. We tested safety properties, e.g., all processes agree on the order of commands.

3 Asynchronous Protocols

In this section we present Distal [11], a DSL for fault-tolerant systems, and P [4] a modeling language for event-driven systems equipped with a bug-finding tool.

3.1 Distal: Syntax and Semantics

We consider asynchronous protocols written in Distal [11]. The system is composed of N processes, where N is a parameter. Each process is associated with a unique identifier, which serves as an address for sending and receiving messages. All processes execute the same protocol $\mathcal{P}$ written using the syntax in Fig. 6. Protocols are composed by an *init* statement and a main loop, composed by a sequence of *upon* statements. An upon statement is followed by a predicate *guard* and a body with instructions to be executed. Processes can access a read-only mailbox variable `mbox`, which contains the received messages. Distal follows the event-driven paradigm where the state of a process tries to be updated upon the reception of a message. Processes exchange messages using instructions `send` and `send to all` that take m a message of type T as input and a PID. All variables are local to a process, there are no global or shared variables. The guard of each upon is a formula over the local state and `mbox`. Guards apply to different

message types and check the values of the received message, e.g., `upon Prepare with m.ballot > ballot` in Fig. 1 line 8, or cardinally conditions `upon Ack with m.ballot=ballot times n/2` which says more than $n/2$ `Ack` messages have been received with the same ballot value as the process' `ballot` (Fig. 1 line 15).

type $\mathbb{M}$ ::=	struct { field **Identifier**; }	
e ::=	const | x | $f(\vec{x})$	**Expressions**
Action ::=	x = e	**Statements**
	| `if` e `then` Action `else` Action	
	| `send(p,m)` | `send(m) to ALL` | `send to` p	
	| Action ; Action	
U ::=	`upon` $\mathbb{M}$ `with` Guard `do` Action | U ; U	**Upon block**
$\mathcal{P}$ ::=	`init` : Action; `loop` : U	**Program**

Fig. 6. Syntax of Distal protocols, p is a PID, $x \in$ *Identifier*, m is a message of some message type in $\mathbb{M}$.

The semantics of a protocol $\mathcal{P}$ is the asynchronous parallel composition of the actions performed by all processes. Formally, the state S of a protocol is a tuple $\langle s, msg\rangle$ where $s \in [P \to Vars \cup Loc \to \mathcal{D}]$ is a valuation of the local variables of each process, including the program location in the local state and $msg : P \to Msg$ is the global set of messages in transit. Given a process $p \in P$, s_p is the local state of p, which is a valuation of p's local variables, and msg_p is the set of messages in transit towards p. When a replica starts, it executes the *init* code block and then runs the main loop forever. Executing an action makes a process change its state. Every process has a message pool that other processes write messages to. The semantics of $send(p, m)$ adds the message m to p's message pool.

```
1  state Propose {
2   entry {
3    if(primary(phase, ps) == this){
4      BroadCast(Propose, (phase, log));
5    }
6   }
7
8   on Propose do (m: Propose) {
9     if(m.phase == phase){
10     log = m.payload; goto Promise;
11    }
12   }
13 } // END state Propose
```

Fig. 7. A snippet of Paxos in P.

In every iteration of the loop a process checks for new messages, moving a subset of its message pool to its local `mbox`. Messages dropped by the network never appear in `mbox`. Several upons could be enabled in the same iteration, but to keep local determinism only the first one will be executed, i.e., the listing order breaks the ties [6]. The network assumptions are defined at execution time

[6] Distal does not emphasize the loop and allows multiple upon statements to be executed in a sequence. The latter is captured by multiple loop iterations where no new messages are delivered in between.

in Distal. We consider both protocols: the ones that make no assumptions for safety, where messages can be reordered, delayed or dropped; or those whose network assumptions for safety are given as first-order formulas over the messages received by processes (examples are given in Sec. 4.1).

P and Distal. P programs are composed of a state machine with several states, where each state has an *entry* function and handlers for different event types which are essentially messages. Fig. 7 shows a snippet of the running example in P. There is a one-to-one correspondence between the upon statements and P message handlers. The latter does not include a `guard`, it triggers on reception. We incorporate the guard as an `if` statement (line 9).

Distal has the high level concept of `times` that is not present in P, we emulate it using a counter variable. In general, P models consist of a single state that handle all system messages, making the translation even more direct. Distal does not provide any implementation nor tools for doing random testing. On the other hand, P provides a well maintained state-of-the-art random testing framework that is used extensively.

4 Round-Based Protocols

In this section we introduce round-based protocols, we define a set of sufficient conditions for an asynchronous protocol to have an equivalent round-based version, and we sketch a rewriting that computes this round-based version.

4.1 Round-Based Syntax and Semantics

The syntax of round-based protocols consists of an initialization function `init` and a phase consisting of a non-empty finite sequence of rounds $r_1, ..., r_k$.

All processes execute the initialization function followed by the given sequence of rounds in lock-step, in a loop. The round number is an abstract notion of time: all processes are in the same round. In each round processes send messages in one synchronized step, using `SEND`. Each process receives in one atomic step a non-deterministically chosen subset of the messages that were sent to it. We denote by $mailbox : P \to 2^{Msg}$ the set of received messages in the current round per process. Messages sent in a round, are either received in the same round or lost. All processes update the local state synchronously, using `UPDATE`.

There are protocols, like Paxos or ViewStamped, that do not make any assumptions on the set of delivered messages to guarantee safety, e.g. agreement, all processes agree on an order of commands [7]. Other protocols are designed for stronger networks. Two representative network assumptions come with Ben-Or [25] and UniformVoting [26]. Ben-Or requires that in each round each process receives at least $n - f$ messages, where f is the number for faulty processes, i.e., $\forall r \in rounds : \forall p \in P : |mbox(p, r)| > n - f$. UniformVoting requires that in

[7] Consensus solutions always work under network assumption, at least for ensuring liveness, but checking liveness is beyond the scope of the paper.

every round, there is one process called *kernel*, such that the message exchanges between any process p and the kernel are received. The kernel may change between rounds: $\forall r \in \mathit{rounds} : \exists k \in P : \forall p \in P : k \in \mathit{mbox}(p, r) \land p \in \mathit{mbox}(k, r)$, where $k \in \mathit{mbox}(p, r)$ is interpreted as follows: if there is a message sent by process k to p then it is received.

4.2 Round-Based Asynchronous Protocols

In this section we define a set of conditions which ensure that an asynchronous protocol is *round-based*, i.e., it has an equivalent round-based semantics. Two executions are equivalent if each process goes through the same sequence of local states, modulo stuttering, in both executions. We introduce synchronization tags, a lightweight annotation for checking the existence of a round structure.

Definition 1. *A synchronization tag in $\mathcal{P}$ is a tuple $\langle(\mathit{phase}, \mathit{round}), \mathit{tagm}\rangle$ where phase and round come from ordered domains and round takes a bounded number of values tagm : $\mathcal{M} \to [\{(\mathit{phase}, \mathit{round})\} \to \mathcal{M} \cup \mathit{Fields}(\mathcal{M})]$ for each message type $\mathbb{M} \in \mathcal{M}$ maps phase and round over the fields of $\mathbb{M}$, or the type itself. For each message $m : \mathbb{M}$ we denote tagm by $m.\mathit{phase}$ and $m.\mathit{round}$.*

A protocol is round-based if there is a synchronization tag and two variables *phase* and *round*, such that, (1) the values of $(\mathit{phase}, \mathit{round})$ monotonically increase (w.r.t. the lexicographic order) in any execution of the protocol, (2) for every message sent m, either using $\mathit{send}(p, m)$ or $\mathit{broadcast}(m)$, m is timestamped with $m.\mathit{phase} = \mathit{phase}$ and $m.\mathit{round} = \mathit{round}$, (3) each guard uses messages timestamped with values greater or equal than the current value *phase* and *round* (4) actions only use (i.e., read) the messages from the mbox that are timestamped with current value of *phase* and *round* (5) between a send/broadcast and a receive either there are only receive statements or the values of *phase* and *round* have been updated. If there is any update between two receive steps then it must update also *phase* and *round*.

We require the user to annotate only the message type with a synchronization tag, and we add two fresh auxiliary variables *phase* and *round* to each protocol. Initially *phase* and *round* have minimal default values. We add assignments to these variables (1) before each send s.t. the second condition is satisfied, i.e., *phase* and *round* are equal to the tag of the sent message, and before each action such that the fourth condition is satisfied, i.e., *phase* and *round* are assigned to the maximal tag of the messages in the guard preceding the action.

The synchronization tag of Paxos in Fig. 1 is conformed by the variable `ballot` for the phase, where phase is an integer. The protocol has no variable that tracks the round, it's highlighted using the symbol ▶. The round domain takes `Prepare` $\preceq$ `Ack` $\preceq$ `Propose` $\preceq$ `Promise` as values. For all messages *round* is mapped onto the message type, and *phase* is mapped on the `ballot` field.

The synchronization tag of the P model in Fig. 7 consists of the field `phase` of each event, for the phase, and the event type for the round. Because the P version of the protocol has a state machine structure that groups handlers/upon

statements into states, the round is the state the process is in, so both the phase and the round are present in the P model. The transformation to Distal replaces the states with a local variable that will track the round/state the process is in. The sequentialization method includes an additional testing tool that checks if the synchronization tag satisfy the five properties.

4.3 Computing a Protocol's Phase Structure

Given a Distal program, we want to compute its round-based counterpart. For this, we need to understand in which order the upons can be executed, under which conditions, and be able to delimit the boundaries between phases in the code. The statements between any two phase variable assignments is what we call the protocol's phase structure. We find it by unfolding the iterations of a Distal program, preserving the order in which the upons happen and their context. Fig. 8 shows the syntax of an unfolded program $\mathcal{P}_{phase}$ and Fig. 9 describes the UNFOLD procedure. The output program satisfies Proposition 1.

type $\mathbb{M}$	::= struct { field **Identifier**; }	
e	::= const | x | $f(\vec{x})$	**Expressions**
S	::= x = e | `if` e `then` S `else` S | S ; S	**Statements**
SEND	::= `send(p, m)` | `send(m) to ALL` | `noop`	**Send actions**
C	::= `if` e `then` ; SEND ; S ; U | C ; C	**Conditionals**
U	::= `mbox` = `havoc()` ; C | `continue`	**Statements**
$\mathcal{P}_{phase}$	::= `init` : $\mathcal{P}$`.init()` ; S ; `loop` : U	**Program**

Fig. 8. Syntax for the phase structure, p is a PID, $x \in$ *Identifier*, and m is a message type in $\mathbb{M}$.

UNFOLD starts by creating a program with an initializing function and a `while(true)` statement with an empty body. It follows by *unfolding* the main loop, this is: 1) inserting a `mbox = havoc();` statement; 2) for each `upon guard do action` in $\mathcal{P}$ it creates an `if(guard) {action}` statement inside the while body (line 8). In the following iterations we repeat the unfolding for every `if` statement created in the previous one, given by the function *leafs*. This procedure is repeated K times, where K is the number of rounds in a phase.

Proposition 1. *For each execution $\tilde{\pi} \in \mathcal{P}_{phase}$ there is a $\pi \in \mathcal{P}$ s.t. π and $\tilde{\pi}$ are equivalent ($\pi \approx \tilde{\pi}$), i.e., their sequence of states is the same modulo stuttering.*

Proof. $\mathcal{P}_{phase}$ doesn't introduce or restrict behaviors of $\mathcal{P}$. Let $\overline{\pi} = [\langle \overline{s_0}, \emptyset \rangle]$ be an execution that starts with $\overline{s_0} = \mathcal{P}_{phase}.init()$ and an empty mailbox. UNFOLD defines $\mathcal{P}_{phase}.init() = \mathcal{P}.init()$ (line 2), so in $\mathcal{P}$ exists $\pi = [\langle s_0, \emptyset \rangle]$ such that $s_0 = \overline{s_0}$. $\langle \overline{s_1}, msg_1 \rangle$ is the result of executing $\mathcal{P}_{phase}$'s first iteration ($height = 1$) from state $\overline{s_0}$ where $havoc()$ returns msg_1. The unfolded conditionals respect the original order in $\mathcal{P}$. Given the same state and mailbox, the selected upon is uniquely determined. $\mathcal{P}_{phase}$ and $\mathcal{P}$ are in the same state with the same mailbox so they execute the same upon, i.e., $\overline{\pi} = \pi = [\langle \overline{s_0}, \emptyset \rangle, \langle \overline{s_1}, msg_1 \rangle]$. The same

argument can be followed at most K times, when the unfolding stops with a phase variable increment. For the following $K + 1...$ transitions, we show that the code to execute is congruent to the first K iterations of UNFOLD. The phase variable is interpreted as a symbolic variable. When a new phase starts, the set of enabled upons is the same as the one considered from the initial state, but with a greater phase value.

4.4 Delimiting Rounds' Boundaries

Round boundaries are defined by round variable assignments. 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. Fig. 10 shows the code of the `Ack` round extracted from our example's unfolded program $\mathcal{P}_{phase}$.

```
1:  procedure UNFOLD(P)
2:    P̄ ← init : P.init(); loop : noop;
3:    for height ∈ 1 to K do
4:      for body in leafs(P_phase) do
5:        body.append(mbox_height = havoc())
6:        for upon in upons(P) do
7:          ifStm ← if(upon.guard){upon.action}
8:          body.append(ifStm)
9:    P̄ ← deadCodeElimination(P̄)
10:   return P̄
```

Fig. 9. Procedure that translates an asynchronous program $\mathcal{P}$ into an unfolded program $\overline{\mathcal{P}}$

We start by iterating line by line starting from the *init* function of $\mathcal{P}_{phase}$ and traverse the main loop until we reach the first assignment of the round variable to `Ack` (line 13 in Fig. 1). Then, we start collecting a sequence of instructions until the next assignment of the round variable (line 18).

All the code before the first assignment is ignored. We introduce ghost flag variables, e.g, `f`, to preserve the conjunction of all the guards leading to the collected code, conserving the execution context. In this case, we cannot send an `Ack` message without having received a valid `Prepare` message.

Finally, the code of every round is split into a `SEND` block, consisting of the (unique) send statement guarded by the conditionals preceding them and an `UPDATE` block that contains the rest of the code except the mailbox's havoc. This completes the code of $\mathcal{P}_{round}$.

```
1  if(mbox(Prepare,m.ballot > ballot)){
2    f = true;
3    m = new Ack(ballot, last, log);
4    send(m, primary);
5    if(f && mbox(Ack,m.ballot == ballot,n/2)){
6      log = longest_log(ballot);
7      log.add(newCommand());
8    }
9  }
```

Fig. 10. Unfolded round `Ack` from motivating example.

This procedure is based on [12], but the input received in that work is significantly different. In [12] the reception loops are found explicitly in the code, these are replaced with calls to a havoc function that non-deterministicaly fills the mailbox. Their work also assumes that every iteration

of the main loop moves to a (greater) new phase and it does not check that this holds. Alg. 9 guarantees this property and the proposition 2 too.

Proposition 2. *Let $[\![\mathcal{P}]\!]$ be the set of executions of $\mathcal{P}$. Given a protocol that makes no network assumptions, $[\![\mathcal{P}]\!] \approx [\![\mathcal{P}_{round}]\!]$, otherwise $[\![\mathcal{P}]\!] \subseteq [\![\mathcal{P}_{round}]\!]$.*

5 Sequentialization of Round-Based Protocols

In this section we define a transformation of a round-based protocol into a sequential one, that preserves safety properties.

5.1 Equivalence with No Network Assumptions

Reductions that over approximate the set of executions are not suitable for testing. If an equivalence exists, given a round-based protocol $\mathcal{P}_{round}$ we build a sequential protocol $\mathcal{P}_{seq}$ using Algorithm 1, such that, given an initial (global) state c_0, all the (global) states reachable from c_0 in $\mathcal{P}_{round}$ are also reachable executing $\mathcal{P}_{seq}$ from c_0. Equivalently, we say that proposition 3 holds.

Proposition 3. *Given a round-based protocol that makes no network assumptions, $[\![\mathcal{P}_{round}]\!] \approx [\![\mathcal{P}_{seq}]\!]$.*

Proof. Let $\rho = \|_{i=1}^{n}\ send_*(i,1) \parallel ... \parallel send_*(i,n); \|_{i=1}^{n}\ update(i);$ be the execution of a $\mathcal{P}_{round}$ round where $\|$ denotes the non-determinism of actions.

The round-based semantics ensure that between any two processes p and q there is at most one message sent from p to q and vice versa. Consequently, the order in which send and receive actions are executed does not matter. We obtain $\rho' = send_*(1,1); ...; send_*(n,n); \|_{i=1}^{n}\ update(i);$ such that $\rho' \approx \rho$.

Two update functions of the same round, on different processes are independent, we can remove other source of non-determinism fixing an arbitrary order $\rho'' = send_*(1,1); send_*(1,n); ...; send_*(n,n); update(1); ...; update(n);$ and this results in $\rho'' \approx \rho$. This reasoning is valid for any arbitrary round.

Algorithm 1 does as follows. The state of $\mathcal{P}_{seq}$ is defined from the global state of $\mathcal{P}_{round}$. The sequential program manipulates the following variables: an integer variable n, corresponding to the number of processes executing the round-based protocol, for each variable v of type T in $\mathcal{P}_{round}$, it has s_v an array of type $ID \to T$, where each index i gives the value of the variable for process p_i. For example, in $\mathcal{P}_{round}$, `mbox` is a local variable that stores the messages received in a round. It changes its type in each round because each of them sends different types of messages. The sequentialization $\mathcal{P}_{seq}$ manipulates several arrays, each storing elements of some message type, and $\texttt{mbox}_r[p_i]$ is the value of `mbox` in round r on process p_i. The transition relation of $\mathcal{P}_{seq}$ defines a total order over all actions performed by all processes, i.e., an order across all `send` and `update`.

Algorithm 1 Sequentialization

```
1: for s = 1 to n do ▶ RoundSeq (send)
2:   for r = 1 to n do
3:     mailbox_R(p_r) += (*)p_s.send(p_r)
4: for i = 0 to n do ▶ RoundSeq (updt)
5:   p_i.update(mailbox_R(p_i))
6:   mailbox_R(p_i) = ∅
7: while true do ▶ ProtocolSeq
8:   for R = 1 to K do
9:     RoundSeq(R)
```

Round-based protocols impose a total order over actions performed by processes across rounds. The sequentialization maintains this order, and it is mainly concerned with the code of one round. The sources of non-determinism at the round level are: (1) the order in which processes send messages (2) the order in which processes execute update (3) the order in which messages are received and (4) which messages are received.

The round-based semantics ensure that between any two processes p and q there is at most one message sent from p to q and vice versa. Consequently, the order in which send and receive actions are sequentialized does not matter.

The update function takes the set of received messages as input, and performs a local computation. Two update functions of the same round, on different processes are independent.

Therefore, we fix one order across processes, denoted $p_1, p_2, \ldots p_n$ where the index gives the order relation. The calls to send and update are sequentialized according to this order, where all sends go before all updates, lines 1 and lines 4 in Algorithm 1.

For each message sent the sequential program makes a non-deterministic choice whether to deliver it or not. Each `send-receive` pair is replaced with an assignment, that non-deterministically adds or not the sent message to the receiver's mailbox.

Algorithm 1 uses "$*$" to represent a non-deterministic choice in line 3, i.e., if the message sent by process p_s to process p_r is received by p_r.

A protocol consisting of K rounds is sequentialized in a while loop that executes the sequentialization of one round after another, in the order in which they are defined in the round-based protocol.

```
1: procedure DELIVERFN(round)
2:   for r = 1 to n do
3:     senders = pick(n − f, P)
4:     for s = 1 to n do
5:       if p_s ∈ senders then
6:         mbox_round(p_r) += p_s.send[p_r]
7:       else
8:         mbox_round(p_r) += (*)p_s.send[p_r]
```

```
1: procedure KERNEL(round)
2:   kernel = pick(1, P)
3:   for s = 1 to n do
4:     for r = 1 to n do
5:       if p_s ∈ kernel then
6:         mbox_round(p_r) += p_s.send[p_r]
7:       else
8:         mbox_round(p_r) += (*)p_s.send[p_r]
```

Fig. 11. Sequentialization for stronger network assumptions. *RoundSeq (send)* is replaced accordingly with DELIVERFN or KERNEL procedures.

5.2 Protocols with Network Assumptions

If the protocol makes assumptions about the set of messages delivered then, by proposition 4, we know that the sequentialization given in Alg. 1 produces an over-approximation of the round-based executions. We strengthen Alg. 1 for the most common fault models to preserve the equivalence between the synchronous protocol and the sequential one. For protocols that do not tolerate faults, e.g., 2PC, each sent message is received. The sequentialization is deterministic.

Ben-Or is not correct unless each process receives at least $n - f$ messages in each round, where f is the number of tolerated faults. In this case the equivalent sequentialization, (DELIVERFN in Fig. 11), picks randomly which $n - f$ messages to deliver to each process. When the network requires the existence of a non-empty kernel, a set of processes that everyone can communicate reliably with, e.g., UniformVoting, the sequentialization (KERNEL in Fig. 11) guesses the processes in the kernel in beginning of each round and always delivers messages between them.

Proposition 4. *Given a round-based protocol that assumes a Deliver n-f or a Kernel network,* $[\![\mathcal{P}_{round}]\!] \approx [\![\mathcal{P}_{seq}]\!]$.

6 Experimental Evaluation

We evaluated the proposed sequentialization on several consensus and replicated state machine protocols and looked for safety violations. For the evaluation we use P [4]. We consider implementation-inspired asynchronous models, and their sequential versions obtained with the algorithms in Sec. 4.3, 5.

First we check that the asynchronous models are round-based. Even though the evaluated protocols are known to be round-based, we test the conditions in Sec. 4.2 for a given synchronization tag using P's monitoring framework. Every send, receive or mailbox read makes a call to an `announce` primitive, where the monitor observes the state of the calling machine and asserts these conditions.

All modeled implementations[8] contain a safety bug. We compared every asynchronous model with its sequential counterpart using P model checker, measuring the time needed for finding these bugs. We found that the most subtle bugs are not found in the asynchronous models, but they are in the sequential version. The experimentation setup consists of manually constructed models in P of the protocol in both asynchronous and sequential versions, a test driver that instantiates the experiment defining the size of the network and other environment variables, and a specification machine that monitors safety violations during the execution. The checking tool systematically explores behaviors of the system model, trying different interleavings of the processes' actions. Each experiment shows the average time (in seconds) to find the bug in 100 executions of 10,000 different schedulers with a timeout of 1 hour.

Bugs are caused by messages being dropped/delayed and processes waiting for messages up to a timeout. To model faults, we implemented a `Timer` machine

[8] https://github.com/vstte22seqprocedure/artifacts

that each process instantiates. The timer machine non-deterministically informs the process that the time waiting for a message expired, making the process move to the next round of the protocol. We use a wrapper around `send`, every time a message is sent, a non-deterministic boolean function chooses to actually send it or to drop it. Next, we describe the bug in each benchmark.

Paxos. This is the example from Sec. 2. Both the asynchronous version and the sequential one contain a bug found in ZAB [9]. The bug occurs when a process sets the variable `last = ballot` at the very beginning of a new phase, when a `Prepare` message is received. This leads to a non-confirmed log being considered as the latest log in the cluster, and leads to a violation of agreement: one replica knows a to be the first command while another one thinks that b is the first. The assignment of `last` should be moved to the `Propose` state upon receiving a message from the primary, confirming that a quorum of processes already have the latest log. The bug requires ten rounds and four phases.

Raft (membership changes). Raft is another consensus algorithm for managing a replicated log. This protocol allows changes into the cluster's configuration, adding or removing nodes to the system. The version presented in [18] contains a bug that produces a safety violation (split brain).[10] This happens when there is a membership change during two consecutive terms and the two leaders have different knowledge of the system's configuration. This causes log entries to be considered as committed using disjoint sets of processes and corrupting the global state. Contrary to *Paxos*, the size of the network is not fixed. At each phase the set of processes might change. To capture this in the sequential model, we introduced a *global* configuration variable that includes all the processes of the system, including the new ones trying to join the cluster. Every process has a "local" knowledge about the current state of the cluster stored in a mapping from processes to set of processes. As we mentioned before, this incomplete knowledge about the system size leads to the mentioned bug.

Ben-Or/Uniform Voting. Ben-Or [25] and Uniform Voting [26] are not leader-based decentralized consensus algorithms. Ben-Or solves binary input consensus, while Uniform Voting considers arbitrary input values, and is a deterministic version of Ben-Or. Once a process decides a value, it keeps deciding the same value forever, the original estimate of each process must be overwritten by the decided value. The bug we introduced omits this, producing executions where all processes decide one value but, later on due to some messages being lost, a process decides a different value. The result for Ben-Or* in Table 1 read as follows: the time comes from using Algorithm 11 as described, but when an under approximation is used, using only two quorums for all the execution the number goes down to 9,12. Ben-Or is designed to work under a particular network assumption, where $n-f$ messages are delivered in each round, otherwise safety is not guaranteed. In the second Ben-Or experiment, we have weakened the network assumptions, and allowed the processes to move on to the next round/phase even

[9] https://issues.apache.org/jira/browse/ZOOKEEPER-2832

[10] https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J?pli=1

if fewer than $n - f$ messages are received. As expected, this leads to a violation of agreement. However this violation is found only using the sequential model.

Table 1. Seconds to find a bug in Asynchronous and Sequential protocols under different network environments. † denotes a timeout (1 hour). R means messages can be reordered, D means messages can be arbitrarily delayed, T means processes can timeout and move to the next round/phase, MD means messages drops.

Network assumption	Protocol	Network	Async	Sequential
Required	Paxos	R D T	†	0,53
	Paxos	R,D,T,MD	†	0,53
	Ben-Or*	R,D	15,04	30,97 / 9,12
	Raft	R,D,T,MD	†	158,44
Weaker	ViewChange	R,D,T,MD	22,02	0,21
	Ben-Or	R,D,T	†	0,19
	UniformVoting	R,D,T,MD	18,22	33,74

Similarly, Uniform Voting requires a non-empty set of processes, called the kernel, to communicate reliably with the entire network, otherwise safety is violated. The kernel is needed because Uniform Voting does not rely on a quorum, the vote and decision is based on a minimum argument. We weaken this network assumption and found a violation of agreement. Typically there is no proof showing that these assumptions cannot be weakened, and there is no understanding what happens if they are weakened. Protocol designers would like to play with the network assumptions and see how the protocol behaves.

Viewstamped Replication (view change). In this experiment we consider the leader election protocol used in Viewstamped Replication [27]. We introduced an artificial bug to the protocol where the function that returns the PID of the current leader to be elected is buggy, instead of returning the same PID for a given phase to all processes, it chooses one non-deterministically. Also, the original protocol gathers quorums of messages to guarantee safety, here we introduce another simple bug where the number of collected messages is less than $n/2$.

Table 1 shows our results. The upper half lists the experiments when the network assumptions of each protocol are respected, the lower one depicts the scenario when these networks are weakened.

7 Conclusions

We propose a technique that reduces testing event-driven asynchronous protocols to testing sequential ones. The sequentialization uses the round structure of protocols, which reduces the number of interleavings the sequentialized version needs to explore. The modularity of the method allows to add more sequentializations for network assumptions not considered in this work and therefore run the tool for new protocols. If no sequentialization produces an equivalent set of executions, the method remains interesting for testing because it can be used with a stronger network assumption that under approximates it.

References

1. O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 614–630. [Online]. Available: https://doi.org/10.1145/2908080.2908118
2. G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997, conference Name: IEEE Transactions on Software Engineering.
3. P. Deligiannis, M. McCutchen, P. Thomson, S. Chen, A. F. Donaldson, J. Erickson, C. Huang, A. Lal, R. Mudduluru, S. Qadeer, and W. Schulte, “Uncovering bugs in distributed storage systems during testing (not in production!),” in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST’16. USA: USENIX Association, Feb. 2016, pp. 249–262.
4. A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: safe asynchronous event-driven programming,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 321–332. [Online]. Available: https://doi.org/10.1145/2491956.2462184
5. A. Bouajjani, M. Emmi, and G. Parlato, “On Sequentializing Concurrent Programs,” in *Static Analysis*, ser. Lecture Notes in Computer Science, E. Yahav, Ed. Berlin, Heidelberg: Springer, 2011, pp. 129–145.
6. S. Qadeer and D. Wu, “KISS: Keep it simple and sequential,” *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, vol. 39, no. 6, pp. 14–24, Jun. 2004.
7. M. Bertran, F.-X. Babot, and A. Climent, “Formal Sequentialization of Distributed Systems via Program Rewriting,” *Electr. Notes Theor. Comput. Sci.*, vol. 188, pp. 53–75, Jul. 2007.
8. A. Bakst, K. v. Gleissenthall, R. G. Kici, and R. Jhala, “Verifying distributed programs via canonical sequentialization,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 110:1–110:27, Oct. 2017. [Online]. Available: https://doi.org/10.1145/3133934
9. B. Kragl, C. Enea, T. A. Henzinger, S. O. Mutluergil, and S. Qadeer, “Inductive sequentialization of asynchronous programs,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. London UK: ACM, Jun. 2020, pp. 227–242. [Online]. Available: https://dl.acm.org/doi/10.1145/3385412.3385980
10. T. Elrad and N. Francez, “Decomposition of distributed programs into communication-closed layers,” *Science of Computer Programming*, vol. 2, no. 3, pp. 155–173, Dec. 1982.
11. M. Biely, P. Delgado, Z. Milosevic, and A. Schiper, “Distal: A framework for implementing fault-tolerant distributed algorithms,” in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2013, pp. 1–8, iSSN: 2158-3927.
12. A. Damian, C. Drăgoi, A. Militaru, and J. Widder, “Communication-Closed Asynchronous Protocols,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 344–363.

13. L. Lamport, “Paxos Made Simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, Dec. 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/
14. D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, ser. USENIX ATC’14. USA: USENIX Association, Jun. 2014, pp. 305–320.
15. C. Mohan and B. Lindsay, “Efficient commit protocols for the tree of processes model of distributed transactions,” *ACM SIGOPS Operating Systems Review*, vol. 19, no. 2, pp. 40–52, Apr. 1985. [Online]. Available: https://doi.org/10.1145/850770.850772
16. F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. Hong Kong: IEEE, Jun. 2011, pp. 245–256. [Online]. Available: http://ieeexplore.ieee.org/document/5958223/
17. C. Drăgoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, “Testing consensus implementations using communication closure,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–29, Nov. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3428278
18. D. Ongaro, “Consensus: Bridging Theory and Practice,” phd, Stanford University, Stanford, CA, USA, 2014, aAI28121474 ISBN-13: 9798662514218.
19. K. V. Gleissenthall, R. G. Kici, A. Bakst, D. Stefan, and R. Jhala, “Pretend synchrony: synchronous verification of asynchronous distributed programs,” *Proc. ACM Program. Lang.*, 2019.
20. B. Demsky and P. Lam, “SATCheck: SAT-directed stateless model checking for SC and TSO,” *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 20–36, Oct. 2015. [Online]. Available: https://doi.org/10.1145/2858965.2814297
21. M. Kokologiannakis, I. Marmanis, V. Gladstein, and V. Vafeiadis, “Truly stateless, optimal dynamic partial order reduction,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–28, Jan. 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3498711
22. M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, and K. Y. Rozier, “Model Checking at Scale: Automated Air Traffic Control Design Space Exploration,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 3–22.
23. J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, “Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 836–850. [Online]. Available: https://doi.org/10.1145/3477132.3483540
24. R. J. Lipton, “Reduction: a method of proving properties of parallel programs,” *Communications of the ACM*, vol. 18, no. 12, pp. 717–721, Dec. 1975. [Online]. Available: http://portal.acm.org/citation.cfm?doid=361227.361234
25. M. Ben-Or, “Another advantage of free choice (Extended Abstract): Completely asynchronous agreement protocols,” in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, ser. PODC ’83. New York, NY, USA: Association for Computing Machinery, Aug. 1983, pp. 27–30. [Online]. Available: https://doi.org/10.1145/800221.806707

26. B. Charron-Bost and A. Schiper, "The Heard-Of model: computing in distributed systems with benign faults," *Distributed Computing*, vol. 22, no. 1, pp. 49–71, Apr. 2009. [Online]. Available: https://doi.org/10.1007/s00446-009-0084-6
27. B. Liskov and J. Cowling, "Viewstamped Replication Revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012.