Formal Verification of a Consensus Algorithm in the Heard-Of Model

Bernadette Charron-Bost
CNRS & LIX, Palaiseau, France

Stephan Merz
INRIA, Nancy, France

Abstract

Distributed algorithms are subtle and error-prone. Still, very few of them have been formally verified, most algorithm designers only giving rough and informal sketches of proofs. We believe that this unsatisfactory situation is due to a scalability problem of current formal methods and that a simpler model is needed to reason about distributed algorithms. We consider formal verification of algorithms expressed in the Heard-Of model recently introduced by Charron-Bost and Schiper. As a concrete case study, we report on the formal verification of a non-trivial Consensus algorithm using the proof assistant Isabelle/HOL.

1 Introduction

Distributed algorithms are often quite subtle, both in the way they operate and in the assumptions under which they work correctly. Indeed, several algorithms have been found to be erroneous, and numerous misunderstandings have arisen due to different interpretations of the precise objectives of the algorithms and of the underlying hypotheses. Formal verification is therefore crucial in distributed computing.

To facilitate their design and understanding, distributed algorithms are generally structured in rounds: during every round, each process first sends messages, then receives messages from other processes, and finally makes a local state transition. However, most existing formal models of distributed algorithms (e.g., [2, 8, 5, 13, 1]) do not take advantage of this structure, but are based on a fine-grained description of systems whose individual processes are represented by communicating state machines. Executions of these models are represented as sequences where events performed by the component state machines are interleaved. Charron-Bost and Schiper [4] recently proposed the Heard-Of (HO) model, a round-based model for fault-tolerant distributed computing, in which executions of distributed algorithms are modeled as infinite sequences of global rounds, which are executed atomically. This coarse-grained abstraction is justified by the assumption that rounds are communication-closed layers: processes react solely to messages sent for the round they currently execute. The second new idea in the HO model is the way system properties (such as degree of synchronism or failure model) are captured by a single predicate on the communication exchanges between processes that can be guaranteed round by round. The coarse-grained atomicity of rounds, obviating the need to consider intermediate system states, and the high level abstraction of communication predicates, encapsulating system guarantees as a whole, both promise to simplify verification of non-trivial distributed algorithms.

In this article we test this expectation by verifying the correctness of *Paxos*, a quite sophisticated Consensus algorithm due to Lamport [11], in the interactive proof assistant Isabelle/HOL [15]. We first describe a generic representation of HO algorithms in Isabelle and then study the *LastVoting* algorithm, the HO version of *Paxos*, as an instance of the generic model. We formally prove that *LastVoting* achieves Consensus among processes despite (benign) communication errors.

Our experience indicates that adopting a round-based model is indeed helpful for formal verification, because it induces a significantly higher level of abstraction than traditional fine-grained models. The proof script is significantly (at least 5 times) shorter than that of the verification of another variant of *Paxos* in Isabelle/HOL [9], based on a fine-grained model. We should emphasize that in this article we do not employ a specific formal method of system development, but focus on the mathematical properties of the HO model. Our results can help to reduce the cost of verification, by model checking or theorem proving, of HO algorithms in standard formal methods for distributed systems, such as I/O automata [14], TLA$^+$ [12] or process algebras.

The paper is structured as follows: Section 2 reviews the HO model and its coordinated variant, formally introduces the Consensus problem, and gives a justification for a coarse-grained abstraction of runs that essentially underlies the verification. Section 3 gives a traditional presentation of the *LastVoting* algorithm, whose formal model in Isabelle is described in Section 4. A detailed correctness proof for the algorithm, following the Isabelle proof, appears in Section 5. Finally, Section 6 discusses what has been achieved and what lies ahead.

2 The round-based HO model

Computations in the HO model are composed of rounds, in which each process exchanges messages, and then takes a step. In the parlance of Elrad and Francez [7], each round is a communication-closed layer in the sense that any message sent in a round can be received only in that round. The technical description of computations is thus similar to the ones proposed by Dwork, Lynch, and Stockmeyer [6], and so the model generalizes the classical notion of synchronized rounds developed for synchronous systems [13]. Typically, communication-closedness in non-synchronous settings is ensured by buffering messages which are early, and by discarding messages which are late.

2.1 Processes, states, and state transitions

We suppose that we have a non-empty finite set Π of cardinality $N > 0$ and a set of messages M (possibly with a designated element specifying the null message). We let $\perp \notin M$ be a placeholder indicating the absence of a message, and we denote by $\vec{M}_\perp$ the set of vectors of elements in $M \cup \{\perp\}$ indexed by Π. Associated with each p in Π, we have a *process*, which consists formally of the following components:

- States_p, a set of *states*;
- Init_p, a non-empty subset of States_p known as the initial states;
- for each integer $r \in \mathbb{N}$, a *message-sending function* S_p^r mapping $\mathit{States}_p \times \Pi$ to elements of M;
- for each integer $r \in \mathbb{N}$, a *state-transition relation* $T_p^r \subseteq \mathit{States}_p \times \vec{M}_\perp \times \mathit{States}_p$.

That is, each process has a set of states, among which is distinguished a subset of initial states. The parameter r in the message-sending function and the state-transition relation is called *round number*.[1] The message-sending function at round r specifies, for each state and each process q, the message that p has to send to q in the given state. The state-transition relation at round r specifies, for each state and each vector of messages received from the other processes, the new state to which p moves. The collection of processes is called an *algorithm* on Π.

2.2 Runs and Communication Predicates

In each round r, process p emits the messages to be sent to each process according to S_p^r, waits for the messages sent to it at round r, and then executes a state transition according to T_p^r in its current state and with the vector $\vec{\mu}_p^r$ of messages that it has received. Process p need not receive all of the messages sent to it (some components of $\vec{\mu}_p^r$ may be equal to $\perp$), and the subset of processes which p *hears of* at round r is denoted by $HO(p, r)$.

Computation evolves in an infinite sequence of rounds. Associated with each computation is its *heard-of collection*, which is the collection of subsets of Π indexed by $\Pi \times \mathbb{N}$:

$$(HO(p, r))_{p\in\Pi, r\in\mathbb{N}} ,$$

recording the sets of processes whose messages were received by process $p \in \Pi$ at round $r \in \mathbb{N}$.

System models differ in the sets of heard-of collections that they provide. The features of a specific computational model (synchrony degree, failure model)

[1]Round numbers could also be local variables, and so be part of the local states. As will be seen later, it is preferable to consider round numbers as parameters of the message-sending functions and state-transition relations.

are thus captured as a whole in the predicate over heard-of collections that it guarantees. Formally, such a predicate $\mathcal{P}$, that we call a *communication predicate*, is a boolean function over the collections of subsets of Π indexed by $\Pi \times \mathbb{N}$:

$$\mathcal{P} \ : \ (2^{\Pi})^{\Pi \times \mathbb{N}} \rightarrow \mathbb{B}$$

(where $\mathbb{B} = \{\mathit{ff}, \mathit{tt}\}$ is the set of truth values), different from the constant predicate ff. The weaker the communication predicate is, the more freedom the system has to provide heard-of sets, the harder it will be to achieve coordination among processes in the corresponding model.

As an example, a communication predicate $\mathcal{P}^f_{HO}$ that can be guaranteed in an asynchronous message-passing system with reliable links and at most f processes that fail by crashing, i.e., by halting prematurely, is

$$\mathcal{P}^f_{HO} \ :: \ \forall p \in \Pi, \forall r \in \mathbb{N} : |HO(p,r)| \geq N - f.$$

Using a simple time-out mechanism, we have shown in [4] that the same failure model in the synchronous case is captured by the predicate $\mathcal{P}^f \wedge \mathcal{P}_{reg}$ where

$$\mathcal{P}^f \ :: \ | \bigcap_{p \in \Pi, r \in \mathbb{N}} HO(p,r) \ | \geq N - f$$

expresses that there are at least $N - f$ processes which are always heard by all processes (as the system is synchronous and at most f processes are faulty), and

$$\mathcal{P}_{reg} \ :: \ \forall p, q \in \Pi, \forall r \in \mathbb{N} : HO(p, r+1) \subseteq HO(q, r)$$

guarantees some regularity among heard-of sets, namely any process that is not heard by some process at round r is no more heard by any process at the subsequent rounds (crash failures).

A *run of A under the communication predicate* $\mathcal{P}$ is then defined to be a set of initial states $(s_p^0)_{p \in \Pi}$, a heard-of collection $(HO(p,r))_{p \in \Pi, r \in \mathbb{N}}$ that satisfies $\mathcal{P}$, and a collection of state transitions $((s_p^r, \vec{\mu}_p^r, s_p^{r+1}))_{p \in \Pi, r \in \mathbb{N}}$ starting from $(s_p^0)_{p \in \Pi}$, compatible both with the sending functions S_p^r and the heard-of collection $(HO(p,r))_{p \in \Pi, r \in \mathbb{N}}$, and such that $(s_p^r, \vec{\mu}_p^r, s_p^{r+1}) \in T_p^r$.

Most fault-tolerant distributed algorithms given in the literature are structured in rounds. This is why we consider that the restriction to round-based models is a reasonable assumption for the formal verification of such algorithms. However, concerning sheer expressiveness of the computational models, it is not clear whether every problem that can be solved in a finer-grained model (such as the asynchronous model in [8]) in which "late" messages are not discarded, also has a solution in round-based models.

2.3 Events and fine-grained executions

From the definition of a process in the HO model, it follows that each process p can execute three types of atomic actions that may change the state of p

itself and the state of the channels incident on p: the sending of a message, the reception of a message, or an internal action. Moreover, the sending of a message does not modify the sender's state, and process states at the end of round r do not depend on the order in which messages are received at round r. The state of a channel c from p to q is entirely determined by the actions that are executed by p and q, and can be modelled by the set of messages that have been sent along c and not yet received. A (global) *configuration* is a tuple of component process and channel states, one per component. An initial configuration is one in which the state of each process p is in $Init_p$, and the state of each channel is the empty set.

An *event* e takes one configuration to another one, and involves a single action by one process. An event by process p is formally defined as a triple (s_p, a, s'_p), where a is an action (a sending, a receipt, or an internal action) executed by p in the state s_p, and s'_p is the new state of p. For each event e, let $round(e)$ denote the round at which e occurs. From the definition of processes, we easily deduce conditions under which an event is *enabled* in a configuration σ.[2] In this classical fine-grained modelling, an *execution* of an algorithm is then defined to be an infinite sequence

$$\lambda = \sigma_0, e_0, \sigma_1, e_1, \cdots$$

of alternating configurations and events where σ_0 is an initial configuration, and σ_{i+1} is the configuration reached from σ_i by event e_i (which must be enabled in σ_i).

Reasoning about a distributed algorithm in a round-based model such as the HO model would be highly simplified if we could ignore intermediate system states of (event-by-event) executions, and could instead pretend that processes execute rounds as single atomic actions and synchronously. In other words, we would like to substitute the notion of a *run* for the one of an *execution* when verifying algorithms, since the structure of the former is much simpler. A reduction result is shown in Section 2.5 that proves the validity of such reasoning in the HO model, for interesting properties.

2.4 The Consensus problem

A *problem* Σ *for* Π is a set of executions, or equivalently, a predicate over executions:

$$\Sigma : \lambda \mapsto \Sigma(\lambda) \in \mathbb{B}.$$

In this paper, we concentrate on the well-known agreement problem, called *Consensus*, regarded as the fundamental problem that must be solved to implement a fault-tolerant system. In this problem, each process p has an initial value v_p from a fixed set V, and must reach an irrevocable decision on one of the initial

[2]We do not make the conditions explicit as it would require to introducing some heavy additional notation, and as we do not use the conditions in the following. See [3] for an explicit formalization of these concepts.

values. Thus each value v in V corresponds to an initial state s_p^v of process p where p holds v as its initial value:

$$\sigma_0(p) = s_p^v$$

Process p has also disjoint sets of *decision states* Δ_p^v, one per value v in V, meaning that p has decided on value v. Let $\sigma(p)$ denote process p's state in the configuration σ. is true of an execution $\lambda = \sigma_0, e_0, \sigma_1, e_1, \cdots$ if λ satisfies the four following properties.

Irrevocability. Once a process decides a value, it remains decided on that value.

$$\forall p \in \Pi, \forall v \in V, \forall i \in \mathbb{N} : \sigma_i(p) \in \Delta_p^v \Rightarrow \forall j \geq i : \sigma_j(p) \in \Delta_p^v.$$

Agreement. No two processes decide differently.

$$\forall p, q \in \Pi, \forall v, w \in V, \forall i, j \in \mathbb{N} : \sigma_i(p) \in \Delta_p^v \wedge \sigma_j(q) \in \Delta_q^w \Rightarrow v = w.$$

Integrity. Any decision value is the initial value of some process.

$$\forall v \in V, \forall p \in \Pi, \forall i \in \mathbb{N} : \sigma_i(p) \in \Delta_p^v \Rightarrow \exists q \in \Pi : \sigma_0(q) = s_q^v.$$

Termination. All processes eventually decide.

$$\forall p \in \Pi, \exists i \in \mathbb{N}, \exists v \in V : \sigma_i(p) \in \Delta_p^v.$$

It is easy to devise Consensus algorithms for synchronous systems, i.e., Consensus algorithms that work under the communication predicate $\mathcal{P}^f \wedge \mathcal{P}_{reg}$. Besides, it has been proven that there is no Consensus algorithm in asynchronous systems that are subject to even a single crash failure [8], which corresponds to the impossibility result in a round-based model to solve Consensus under the communication predicate $\mathcal{P}_{HO}^1$ [16]. In fact, partial synchrony assumptions are sufficient to make Consensus solvable: Dwork, Lynch, and Stockmeyer [6], and then Lamport [11] presented round-based Consensus algorithms that maintain Agreement in the event of any number of benign errors, and take a decision if the system is stable during a "sufficiently long" period, which is captured by a communication predicate that holds when the heard-of sets at several (namely 4) consecutive rounds are "large enough". Our purpose here is precisely to give a formal proof of Lamport's algorithm, known as the *Paxos* algorithm.

2.5 Causality relation. Equivalent executions

More important than the total ordering of events in an execution, which may be purely accidental, is the *causality relation* between events originally defined by Lamport [10]. Formally, this relation is defined as follows. Given two events e_i and e_j in some execution λ, e_j directly depends on e_i, denoted $e_i \prec_1 e_j$, if one of the following conditions holds:

1. e_i and e_j are events by the same process, and e_i occurs before e_j in λ.

2. e_i is the sending of some message, and e_j is its reception.

The causal ordering of the set of events occurring in λ, written $e_i \prec e_j$, is the transitive closure of the relation $\prec_1$. Obviously, the relation $\prec$ is an irreflexive partial ordering, and the total ordering $e_0, e_1, e_2, \cdots$ preserves the causality relation, i.e., is a linear extension of $\prec$. Conversely, any linear extension of the causality relation provides a possible execution with the same initial configuration. We say that two executions λ and λ' are *equivalent*, written $\lambda \simeq \lambda'$, if they share the same initial configuration, the same set of events and the same causality relation. The definitions of $\simeq$ and of events ensure that whenever $\lambda \simeq \lambda'$ for two executions $\lambda = \sigma_0, e_0, \cdots$ and $\lambda' = \sigma'_0, e'_0, \cdots$, then the sequences of local states $(\sigma_i(p))_{i\in\mathbb{N}}$ and $(\sigma'_i(p))_{i\in\mathbb{N}}$ of a process p obtained from these two executions agree up to finite stuttering.

As Chandy and Lamport explain in [2], problems that correspond to predicates the truth-value of which depends on the total ordering of events in an execution (and not only on the causality relation) cannot be solved in a distributed system. Thereby, we only consider predicates Σ that are invariant under $\simeq$, i.e,

$$\lambda \simeq \lambda' \Rightarrow \Sigma(\lambda) = \Sigma(\lambda').$$

The Consensus problem is easily seen to be invariant under $\simeq$.

Due to the particular structure of algorithms in the HO model, executions inherit some nice properties. First, we show that the round order is consistent with the causality relation.

Proposition 2.1 *If e and e' are two events in an execution such that $e \prec e'$, then $round(e) \leq round(e')$.*

Proof: By definition of the relation $\prec$, it is sufficient to argue when $e \prec_1 e'$. There are two cases to consider.

1. e and e' are events by the same process p. From the definition of p's computation round by round, it follows that $round(e) \leq round(e')$.

2. e is the sending of some message, and e is its reception. Since every round is a communication-closed layer, e and e' occur at the same round, i.e., $round(e) = round(e')$. □

It follows that events in any given execution λ can be reordered without affecting the causality relation in a way that preserves round numbers, as expressed in the following proposition.

Proposition 2.2 *For any given execution λ, there exists some equivalent execution λ^r that preserves round numbers, i.e., e occurs before e' in λ^r only if $round(e) \leq round(e')$.*

Proof: Consider an arbitrary total ordering $p_1, \cdots, p_N$ on the set Π. We construct λ^r inductively, round by round. For each round r, the events in λ that occur in round r are ordered in λ^r as follows: we consider the sequence of the message emissions by p_1 at round r in λ, followed by the sequence of the emissions by p_2 at round r in λ, and so on up to p_N. Then we carry on with the sequence of the receptions by p_1 at round r in λ, and so on up to the sequence of the receptions by p_N at round r in λ. Finally, we complete this fragment by the sequence of the internal actions by $p_1, \cdots, p_N$ that occur in λ at round r. The resulting infinite sequence of events is obviously a total ordering of all the events in λ that extends the causality relation in λ. From this linear extension of the events in λ and the initial configuration of λ, we get an infinite sequence λ^r of alternating configurations and events which is an execution equivalent to λ. Moreover by construction, for any pair of events e and e' such that $round(e) < round(e')$, e occurs before e' in λ^r. □

The execution λ^r reorders the events in λ round-by-round. Note that there are actually many round-by-round executions that are equivalent to λ: they differ only in the way the events of each round are ordered.

Each round-by-round execution λ^r of algorithm A can be naturally mapped into a run of A that we denote $\rho(\lambda^r)$. Moreover, we easily check that two equivalent round-by-round executions correspond to the same run in this mapping. Thanks to the latter remark and Proposition 2.2, for any execution λ, we can define $\rho(\lambda)$ to be equal to some $\rho(\lambda^r)$ where λ^r is a round-by-round execution equivalent to λ. Then we introduce a weakening of the relation $\simeq$ as follows: λ and λ' are *weakly equivalent*, denoted $\lambda \sim \lambda'$, if λ and λ' are mapped into the same run, i.e., $\rho(\lambda) = \rho(\lambda')$. Clearly, $\sim$ is an equivalence relation that contains the relation $\simeq$. The latter inclusion is strict since in each round of a run, neither the ordering of send events by process p nor the ordering of its receptions are specified. A problem Σ that is invariant under the weak equivalence $\sim$, i.e., a predicate over executions such that

$$\lambda \sim \lambda' \Rightarrow \Sigma(\lambda) = \Sigma(\lambda'),$$

actually corresponds to a predicate no more over executions but over runs. For such a problem Σ, the coarse-grained abstraction of runs in the HO model is justified, and we say that the HO algorithm *A solves Σ under the communication predicate $\mathcal{P}$* if any run of A under $\mathcal{P}$ satisfies Σ.

Due to the very definition of a process in the HO model, the configuration at the end of each round does not depend on the ordering of events in each round, as it is formally expressed in the following proposition.

Proposition 2.3 *If λ and λ' are two weakly equivalent round-by-round executions, then for any round $r \in \mathbb{N}$, the configurations in λ and λ' just at the end of round r are equal.*

In other words, the sequence of the configurations at the end of each round in a round-by-round execution λ^r is entirely determined by the corresponding run $\rho(\lambda^r)$. Combined with the invariance of the Consensus problem under $\simeq$, Proposition 2.3 ensures that Consensus is actually invariant under $\sim$. This formally justifies that the coarse-grained abstraction of runs in the HO model is sufficient with regard to the Consensus problem and the correctness of Consensus algorithms. Another proof of this reduction result is given in [3], and is applied for model checking Consensus algorithms.

2.6 Coordinated HO model

Numerous algorithms for Consensus are coordinator-based – e.g., the Consensus algorithms proposed by Dwork, Lynch, and Stockmeyer [6], Chandra and Toueg's algorithm [1], as well as Lamport's *Paxos* [11]. The correctness of these algorithms is guaranteed by certain properties concerning the choice of coordinators: for example, Termination in *Paxos* requires that during some phase, a majority of processes hears of the coordinator of the phase. For such algorithms, we introduce a slight variation of the HO model,[3] namely the *Coordinated HO (CHO) model*, for which algorithms refer to the notion of coordinators, and predicates are stated not just about heard-of sets, but also about coordinators.

A CHO algorithm is much like an ordinary HO algorithm. Reflecting the fact that the messages sent by a process p in a round of a CHO algorithm do not uniquely depend on the current state, but also on the identity of a *coordinator*, the message-sending function S_p^r is no longer a function from $\mathit{States}_p \times \Pi$ to M but instead a function

$$S_p^r \ : \ \Pi \times \mathit{States}_p \times \Pi \ \rightarrow \ M.$$

Similarly, the state of process p at the end of a round does not only depend on its current state and the collection of the messages it has just received, but also on the identity of its coordinator. So, the state-transition relation T_p^r is such that

$$T_p^r \subseteq \mathit{States}_p \times \vec{M}_\perp \times \Pi \times \mathit{States}_p$$

where $\vec{M}_\perp$ still denotes the set of vectors, indexed by Π, of elements in $M \cup \{\perp\}$. The functions $(S_p^r)_{r\in\mathbb{N}}$ and the relations $(T_p^r)_{r\in\mathbb{N}}$ define the *coordinated process* p, and the collection of coordinated processes is called a *coordinated algorithm.*

In every round r, each process p (1) applies the message-sending function S_p^r to the current coordinator and the current state to generate the messages to be sent, and (2) changes its state according to T_p^r, the current state, the incoming messages, and its current coordinator. The combination of the two steps is called a *coordinated round*, and p's coordinator at r is denoted $\mathit{Coord}(p, r)$.

We say that r is a *uniformly coordinated round* if

$$\forall p, q \in \Pi : \mathit{Coord}(p, r) = \mathit{Coord}(q, r)$$

[3]The reader is referred to [4] for a discussion of the differences between the HO and CHO models.

and r is *well coordinated* if

$$\forall p \in \Pi : \mathit{Coord}(p,r) \in \mathit{HO}(p,r).$$

Uniformly and well coordinated rounds play a key role for reaching Agreement in coordinated Consensus algorithms such as the *LastVoting* algorithm presented in Section 3.

With each computation we associate not only the heard-of set collection, but also the *coordinator collection* $(\mathit{Coord}(p,r))_{p\in\Pi, r\in\mathbb{N}}$, that the system provides. A predicate over both heard-of sets and coordinator collections is called a *communication-coordinator predicate*. For example, we shall consider the predicate

$$\exists r_0 \in \mathbb{N}, \forall p,q \in \Pi : \mathit{Coord}(p,r_0) = \mathit{Coord}(q,r_0) \wedge \mathit{Coord}(p,r_0) \in \mathit{HO}(p,r_0)$$

which guarantees that there is eventually a uniformly and well coordinated round.

Similar to a run of a standard HO algorithm, a *run of* a CHO algorithm *A under the communication-coordinator predicate* $\mathcal{P}^C$ consists of an initial configuration $(s_p^0)_{p\in\Pi}$, a heard-of collection $(\mathit{HO}(p,r))_{p\in\Pi,r\in\mathbb{N}}$ and a coordinator collection $(\mathit{Coord}(p,r))_{p\in\Pi,r\in\mathbb{N}}$ that satisfy $\mathcal{P}^C$, and a collection of state transitions $((s_p^r, \vec{\mu}_p^r, q, s_p^{r+1}))_{p\in\Pi,r\in\mathbb{N}}$ starting from $(s_p^0)_{p\in\Pi}$, compatible with the sending functions S_p^r, the heard-of and the coordinator collections, and such that$(s_p^r, \vec{\mu}_p^r, q, s_p^{r+1}) \in T_p^r$. Finally, the notion of what it means for a CHO algorithm to solve a problem that is invariant under the relation $\sim$ is similar to the one for an HO algorithm.

As for heard-of sets, processes are provided with coordinators by the system, and the properties of the collections $(\mathit{Coord}(p,r))_{p\in\Pi,r\in\mathbb{N}}$ are part of the system assumptions. The way coordinators are provided is not specified in the model: processes may use some external devices (physical devices or oracles) that are capable of reporting the name of coordinators to every process, or it may be the result of some computation (in other words, the CHO algorithm is emulated by an ordinary HO algorithm). In the second case, a very common "off-line" strategy, usually called the *rotating coordinator* strategy, consists in selecting for every process p in Π:

$$\mathit{Coord}(p,r) = p_{1+(r \bmod n)}$$

when $\Pi = \{p_1, \ldots, p_n\}$. With the rotating coordinator strategy, Agreement on the name of a coordinator is for free, that is, every round is uniformly coordinated. On the other hand, the on-line strategy that consists in selecting p's coordinator in its heard-of set provides well coordinated rounds for free (in the case heard-of sets do not vary too much in time). A critical point is to achieve rounds which are both uniformly and well coordinated.

3 The *Paxos* algorithm and its CHO version

Most Consensus algorithms work correctly only if some invariant properties on the system behavior are satisfied. That corresponds in the HO model to communication predicates which are conditions that hold at all rounds (e.g., $\mathcal{P}^f_{HO}$ or $\mathcal{P}_{reg}$). Taking a closer look at these algorithms, it turns out that the safety conditions of Consensus, namely Integrity and Agreement, may be violated if there are some "bad" periods during which these predicates do not hold.

In a seminal paper [6], Dwork, Lynch, and Stockmeyer showed how to cope with such bad periods, and designed an algorithm, which solves Consensus if a "sufficiently long" good period occurs. The very novel and basic idea of this algorithm is to satisfy safety conditions no matter how badly processes communicate, that is even if many errors occur in the system.

The same idea is followed in the *Paxos* algorithm designed by Lamport [11]. Like many Consensus algorithms, *Paxos* is a coordinated algorithm. However, it was the first coordinated algorithm that does not rely on any particular coordinator scheme, such as the rotating coordinator scheme mentioned in Section 2.6.

In [4], we designed a CHO algorithm corresponding to *Paxos* that we call *LastVoting*[4]. Rounds in *LastVoting* are grouped into *phases*[5]: phase $\phi \in \mathbb{N}$ consists of the consecutive four rounds 4ϕ, $4\phi+1$, $4\phi+2$, and $4\phi+3$. Processes keep the same coordinator during each phase ϕ, denoted by $Coord(p, \phi)$.

Every process p maintains a variable x_p initialized with its initial value v_p, and another variable ts_p that p sets to $\phi + 1$ when p updates x_p at phase ϕ. At round 4ϕ, p sends the current value of x_p timestamped by ts_p to its coordinator. Then, in round $4\phi+1$, a coordinator of ϕ casts a vote if it has received conclusive informations from enough (namely a strict majority of the) processes; otherwise it casts no vote and misses its turn. A coordinator determines its vote value according to a timing criterion: a coordinator votes for some value it has heard of with the *most recent timestamp*. Upon receiving a vote v, process p sets x_p to v, and sends an acknowledgement to its coordinator at round $4\phi + 2$. If a coordinator receives enough acknowledgements, then it is ready to decide, and informs all processes by broadcasting its vote. If process p knows that its coordinator is ready to decide and has voted for v in phase ϕ, then p decides v at round $4\phi + 3$ by assigning v to its variable $decide_p$. The code of the *LastVoting* algorithm is given as Algorithm 1. Here, only the emissions of non-null messages and the state transitions with a non-null effect are specified. Moreover, as processes are all deterministic in the *LastVoting* algorithm, the T^r_p's are specified as functions (of p's current state, p's coordinator, and the messages that p receives at round r).

The following theorem asserts that *LastVoting* is always safe, even in the presence of multiple coordinators at some phases. Moreover, Termination at phase ϕ_0 is achieved if all processes share the same coordinator c_0 at ϕ_0, c_0

[4]The reader is referred to [4] for a discussion about the precise differences between *Paxos* and *LastVoting*.

[5]Similar to round numbers, phase numbers are not part of local states, but are instead parameters.

Algorithm 1 The *LastVoting* algorithm.

```
Initialization:
  $x_p \in V$, initially $v_p$   {$v_p$ is the initial value of $p$}
  $vote_p \in V \cup \{?\}$, initially ?
  $decide_p \in V \cup \{\bot\}$, initially $\bot$
  $commit_p$ a Boolean, initially false
  $ready_p$ a Boolean, initially false
  $ts_p \in \mathbb{N}$, initially 0

Round $r = 4\phi$ :
  $S_p^r$ :
    send $\langle x_p , ts_p \rangle$ to $Coord(p, \phi)$

  $T_p^r$ :
    if $p = Coord(p, \phi)$ and
    number of $\langle \nu , \theta \rangle$ received $> N/2$ then
      let $\overline{\theta}$ be the largest $\theta$ from $\langle \nu , \theta \rangle$ received
      $vote_p$ := one $\nu$ such that $\langle \nu , \overline{\theta} \rangle$ is received
      $commit_p$ := true

Round $r = 4\phi + 1$ :
  $S_p^r$ :
    if $p = Coord(p, \phi)$ and $commit_p$ then
      send $\langle vote_p \rangle$ to all processes

  $T_p^r$ :
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
      $x_p := v$ ; $ts_p := \phi + 1$

Round $r = 4\phi + 2$ :
  $S_p^r$ :
    if $ts_p = \phi + 1$ then
      send $\langle ack \rangle$ to $Coord(p, \phi)$

  $T_p^r$ :
    if $p = Coord(p, \phi)$ and
    number of $\langle ack \rangle$ received $> N/2$ then
      $ready_p$ := true

Round $r = 4\phi + 3$ :
  $S_p^r$ :
    if $p = Coord(p, \phi)$ and $ready_p$ then
      send $\langle vote_p \rangle$ to all processes

  $T_p^r$ :
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
      $decide_p := v$
    if $p = Coord(p, \phi)$ then
      $ready_p$ := false
      $commit_p$ := false
```

hears of a strict majority of processes at the first and third rounds of ϕ_0, and c_0 is heard by all processes in rounds $4\phi_0 + 1$ and $4\phi_0 + 3$.

Theorem 3.1 *Any run of the LastVoting algorithm satisfies the Integrity, Agreement, and Irrevocability conditions. Moreover, Termination is guaranteed by the communication-coordinator predicate $\mathcal{P}_{LastVoting}$:*

$$\begin{aligned}
&\exists \phi_0 \geq 0, \exists c_0 \in \Pi, \forall p \in \Pi : \\
&\qquad |HO(c_0, 4\phi_0)| > N/2 \ \wedge\ |HO(c_0, 4\phi_0 + 2)| > N/2 \\
&\quad \wedge\ c_0 = Coord(p, \phi_0) \ \wedge\ c_0 \in HO(p, 4\phi_0 + 1) \ \wedge\ c_0 \in HO(p, 4\phi_0 + 3)
\end{aligned}$$

Hence, the LastVoting *algorithm solves Consensus under $\mathcal{P}_{LastVoting}$.*

We have formally proved this theorem in the interactive proof assistant Isabelle. The following sections explain our encoding of the algorithm, and of its proof, in Isabelle.

4 Representing Heard-Of Algorithms in Isabelle

The HO model is attractive for formal, tool-supported verification of distributed algorithms because many problems, such as Consensus, can be studied over a coarse-grained abstraction of runs. Tsuchiya and Schiper [17, 18] have used model checking techniques to validate some of the algorithms presented in [4]. In [3], we have similarly used TLC [20], the model checker for Lamport's specification language TLA$^+$ [12], to validate certain HO algorithms. However, it is well known that model checking suffers from the state-space explosion problem. In the case of algorithms such as *LastVoting*, even symbolic model checkers will typically be able to handle finite instances of no more than three or four processes even if using the reduction in Proposition 2.3, and considering only coarse-grained representations of runs.

In this article, we report on the formal verification of Theorem 3.1 in the interactive proof assistant Isabelle/HOL [15]. In contrast to model checking, we obtain a formal correctness proof for the *LastVoting* algorithm, for an arbitrary number of processes. However, carrying out the proof in Isabelle requires strong guidance by the user who has to encode the arguments that underly a correctness proof in the logic of the theorem prover. The tool checks the soundness of each step, ensuring that no corner case has been overlooked. It also provides limited automation for finding proofs that would be considered "obvious" by a human reader. Like most proof assistants, Isabelle relies on a small trusted kernel that is ultimately responsible for certifying theorems, and this architecture makes Isabelle proofs highly trustworthy. The second author had previous experience with the verification with a variant of the *Paxos* algorithm in Isabelle/HOL [9], based on a traditional, fine-grained representation of executions. Because the algorithms are quite similar, any significant gain in the effort necessary to carry out the correctness proof in Isabelle can be attributed to the use of coarse-grained runs in the HO model.

4.1 Isabelle/HOL

Isabelle [15] is a generic framework for mechanizing logics. The user must encode the syntax of the target logic in the simply typed λ-calculus. The proof system should be represented in natural deduction style as sequents

$$[\![P_1;\ \ldots;\ P_n]\!] \Longrightarrow Q$$

where P_i and Q are propositions written in the syntax of the target logic. The Isabelle kernel provides elementary functions for applying proof rules (i.e., combining sequents), and theorems can only be created by the application of these functions. The soundness of reasoning thus hinges on the correct implementation of the kernel and the soundness of the presentation of the proof system; both are ensured by standard software-engineering techniques such as tests and code reviews. Isabelle also comes with many automated proof methods such as first-order reasoners, a rather efficient rewriting engine, and a decision procedure for linear arithmetic, which can be instantiated for target logics. The correctness of the results obtained by these proof methods is certified through applications of the kernel functions.

For concrete applications, users do not encode a logic of their own but use one of the predefined logics that come with a rich library of definitions and theorems and that instantiate the generic proof machinery for immediate use. A verification project consists of several *theories* that contain definitions (modeling given problems), as well as statements and proofs of theorems. Traditionally, proofs were written as applications of tactics that reduced the statement of a theorem to subproblems, until these could be established by an already available theorem or some automatic proof method. More recently, Isabelle has introduced a structured proof language called Isar [19], in which a user writes a proof in a language resembling standard mathematical prose. Although Isar proofs are more verbose – in particular, each subproblem is stated explicitly whereas it is derived implicitly in tactic applications – they are easier to read and to maintain, and we used them throughout our verification of *LastVoting*.

Isabelle/HOL is the encoding in Isabelle of higher-order logic, following Church's Simple Theory of Types. It is the most widely used object logic in Isabelle and has been used for numerous verification projects including the proof of mathematical theorems, encodings of programming language semantics, and the verification of security protocols, to name just a few examples. We give a brief introduction to some of the features of Isabelle/HOL that we use in the following; extensive documentation is available from the Isabelle Web site.[6] In the following, we will no longer distinguish between Isabelle and Isabelle/HOL.

Types in Isabelle. Type constructors include *bool*, the type $'a \to 'b$ of total functions[7] with arguments of type $'a$ and results of type $'b$, and the product type $'a * 'b$. The function arrow is right-associative: $'a \to 'b \to 'c$ is parsed as $'a \to ('b \to 'c)$. Functional values are defined as λ-terms, and function application is written as juxtaposition of the function and its argument(s). The function and product type constructors are polymorphic: type variables such as $'a$ and $'b$ can be instantiated by arbitrary types. Type inference is usually implicit, but type constraints can be given in the form $v :: ty$ where v is a term and ty a type.

Sets are identified with their characteristic predicates: the type $'a\ set$ is synonymous in Isabelle with the type $'a \to bool$. Records are similar to products but have named fields; for every field f of a record Isabelle defines a selector function, also called f, to access the field. Other operations on records include record construction and (functional) record update, written in the form

$$(\!| f = valf, g = valg |\!) \quad \text{and} \quad rec(\!| g := valg' |\!)$$

for a record with fields f and g, assuming that $valf$, $valg$, and $valg'$ are terms of appropriate type and that rec denotes an already created object of record type.

Isabelle comes with a facility for defining inductive data types. For example,

datatype *nat* =
  Zero ("0")
| *Suc nat*

defines the Peano numbers as an inductive data type with a nullary constructor *Zero* (written 0) and a unary constructor *Suc*. In order to ensure that an inductive data type is well-formed, all occurrences of the type being defined

[6]`http://isabelle.in.tum.de/`

[7]The actual syntax of Isabelle/HOL uses $\Rightarrow$ for function types; we write $\to$ for coherence with the standard mathematical notation used in the previous sections of this article.

(such as *nat* in the example) as arguments of the constructors must be positive (so, one could not have a constructor $C\ (nat \rightarrow nat)$ in the above definition).

Inductive type constructors can be polymorphic as well. For example, the type $'a\ option$ is defined as a data type with a nullary constructor *None* and a unary constructor $Some\ 'a$. It can be understood as augmenting type $'a$ by an additional "undefined" value. Isabelle also defines the function

$$the :: 'a\ option \rightarrow 'a$$

such that $the(Some\ x) = x$, for any x of type $'a$. The option type is used to represent partial functions in a logic of total functions: type $'a \rightharpoonup 'b$ is a synonym for the type $'a \rightarrow ('b\ option)$; a function of such a type returns *None* for an argument outside the domain of the partial function, and $Some\ y$ (for an appropriate value y) otherwise.

Locales. An Isabelle *locale* is a module whose interface consists of a signature (a number of operators and their types) and certain assumptions (logical formulas). For example, a locale of partial orders would introduce the signature containing the sole operator

$$leq :: 'a \rightarrow 'a \rightarrow bool$$

and assumptions stating the reflexivity, anti-symmetry, and transitivity of *leq*. Locales serve to structure logical theories: inside a locale, the operators indicated in the signature are considered as constants, and the assumptions as axioms. A locale can define further operators and prove certain properties. For example, one could define a strict variant *less* of *leq* and prove transitivity properties about *less* and *leq*. These generic properties are made available to *interpretations*, which instantiate the locale by concrete operators. For example, the partial order locale could be interpreted by instantiating *leq* by the standard ordering $\leq$ on natural numbers. At this point, the user must prove that the locale assumptions are indeed satisfied by the instance. All derived operators and facts are then available to the interpretation.

4.2 A Generic Model of CHO Machines in Isabelle

We make use of Isabelle's *locale* mechanism to represent CHO algorithms.[8] Elementary theorems and proof rules are proved generically within the locale. Models of concrete algorithms are obtained as instances of the locale, inheriting all of its properties.

In the Isabelle model, we represent the set Π of processes by a type variable $'proc$. Similarly, the type variables $'pst$ and $'msg$ serve to represent the sets of local process states and messages, which will be defined concretely for particular algorithms. We formalize (coarse-grained) runs of algorithms as sequences of

[8]Non-coordinated HO algorithms are a special case of CHO algorithms that place no constraints on the coordinator collection.

locale *CHOAlgorithm* $=$
fixes
  $initState :: 'proc \to 'pst \to bool$ **and**
  $sendMsg :: nat \to 'proc \to 'proc \to 'pst \to 'proc \to 'msg$ **and**
  $nextState :: nat \to 'proc \to 'pst \to ('proc \rightharpoonup 'msg) \to 'proc \to 'pst \to bool$ **and**
  $commPred :: (nat \to 'proc\ HO) \to (nat \to 'proc\ coord) \to bool$
assumes
  $finiteProc : finite\ (UNIV :: 'proc\ set)$

Figure 1: An Isabelle locale for representing CHO algorithms.

rounds: Proposition 2.3 justifies that it is enough to verify a problem Σ, such as Consensus, for coarse-grained runs if Σ depends only on the causality of events. We therefore define the type

types $('proc, 'pst)\ run\ =\ nat \to 'proc \to 'pst$

to represent runs as infinite sequences of arrays of process states.[9] Types used for the representation of heard-of and coordinator collections are defined as

types
  $'proc\ HO\ =\ 'proc \to 'proc\ set$
  $'proc\ coord\ =\ 'proc \to 'proc.$

The definition of an Isabelle locale representing CHO algorithms appears in Fig. 1. It takes four parameters and states one assumption. The parameter *initState* represents a predicate (boolean function) characterizing the initial states for every process. Similarly, the parameters *sendMsg* and *nextState* formally represent the message-sending functions S_p^r and the state-transition relation T_p^r, in their versions for CHO algorithms as introduced in Section 2.6. The parameter *commPred* represents the communication-coordinator predicate and is evaluated over heard-of and coordinator collections.

The locale *CHOAlgorithm* assumes that the set of all processes is finite; it is non-empty because any type in HOL must be inhabited.

Runs of a CHO algorithm start in a configuration where each process is in an initial state:

definition *initConfig* **where**
  $initConfig\ cfg\ \equiv\ \forall p.\ initState\ p\ (cfg\ p).$

Similarly, we introduce a predicate that relates two successive configurations *cfg* and *cfg'* at round r of a run: every process p updates its local states according to the relation *nextState*, given assignments of heard-of sets and coordinators.

[9]Unlike the definitions of runs of (C)HO algorithms in Sections 2.1 and 2.6, we omit the vectors $\vec{\mu}_p^r$ of received messages in the Isabelle representation of runs. We do not need to represent messages explicitly in the following since we only verify properties of states that occur in runs.

definition *nextConfig* **where**
$nextConfig\ r\ cfg\ (HO :: 'proc\ HO)\ (coord :: 'proc\ coord)\ cfg' \ \equiv$
$\forall p.\ nextState\ r\ p\ (cfg\ p)\ (rcvdMsgs\ p\ (HO\ p)\ coord\ cfg\ (sendMsg\ r))$
$\qquad (coord\ p)\ (cfg'\ p)$

where the utility function *rcvdMsgs* computes the vector of messages that process p receives from the processes in its heard-of set, given their message-sending function:

definition *rcvdMsgs* **where**
$rcvdMsgs\ p\ ho\ coord\ cfg\ send\ \equiv$
$\lambda q.\ if\ \ q \in ho\ then\ Some(send\ q\ p\ (cfg\ q)\ (coord\ q))\ else\ None.$

The definitions of *nextConfig* and *rcvdMsgs* make use of some features of higher-order logic, in particular partial instantiation of function parameters, to obtain concise specifications.

We can now define the runs of an CHO algorithm, relative to heard-of and coordinator collections *HOs* and *coords*, as infinite sequences of configurations $c_0 c_1 \ldots$ such that the following conditions hold:

- c_0 satisfies *initConfig*,
- the predicate *nextConfig* holds for any pair (c_r, c_{r+1}) of successive configurations, with respect to the heard-of and coordinator assignments at that round, and
- the heard-of and communicator collections satisfy the communication-coordinator predicate *commPred*.

definition *CHORun* **where**
$CHORun\ rho\ HOs\ coords\ \equiv$
$\quad initConfig\ (rho\ 0)$
$\wedge\ (\forall r. nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r)))$
$\wedge\ commPred\ HOs\ coords.$

4.3 Elementary lemmas

A certain number of lemmas can be derived at the level of the Isabelle locale; these are then available for all instantiations of the locale. For later use, we derive some consequences of the finiteness assumption for the set of processes, and convenience rules for inductive reasoning over runs.

Finiteness and cardinality lemmas. By assumption *finiteProc* (cf. Fig. 1), any set of processes is finite. Moreover, the range of a partial function from processes to an arbitrary type $'a$ is finite.[10] This is expressed by the two following lemmas whose proof in Isabelle is straightforward. We make them available to the automatic proof procedures of Isabelle so that they do not need to be invoked manually.

[10]The standard library of Isabelle/HOL already contains an analogous lemma for total functions.

lemma *finite_procset*: *finite* $(S :: {'proc\ set})$

lemma *finite_ran*: *finite* $(ran\ (f :: {'proc} \rightharpoonup {'a}))$

A frequent argument in the correctness proofs of many fault-tolerant distributed algorithms is that any two majority sets must have a non-empty intersection. This property is expressed by the two following lemmas.

lemma *majorities_intersect*:
  assumes $card(UNIV :: {'proc\ set}) < card(S :: {'proc\ set}) + card(T :: {'proc\ set})$
  shows $S \cap T \neq \{\}$

lemma *majoritiesE*:
  assumes $card(S :: {'proc\ set}) > card(UNIV :: {'proc\ set})\ div\ 2$
  and $card(T :: {'proc\ set}) > card(UNIV :: {'proc\ set})\ div\ 2$
  obtains p **where** $p \in S$ **and** $p \in T$

The second lemma is a consequence of the first one; it asserts that given two sets S and T each of which contains a strict majority of processes, there exists some process p that is an element of both S and T (the **obtains** keyword ensures that p is an *eigenvariable* that does not occur in the expressions S and T).

Reasoning about runs. Many properties of runs are proved by case distinction on the current action, or by induction. The two following proof rules, automatically verified by Isabelle, provide the basis for such reasoning. For example, rule *CHORun_induct* states that, given a run of a CHO algorithm, a property $P\ n$ is true for an arbitrary $n \in \mathbb{N}$ if $P\ 0$ follows from the initialization predicate, and if P can be shown to hold of any successor round $Suc\ r$ whenever P holds of round r and the configurations for rounds r and $Suc\ r$ are related by the relation *nextConfig*.

lemma *CHORun_Suc*:
  assumes $CHORun\ rho\ HOs\ coords$
  and $\bigwedge r.\ nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r)) \implies P\ r$
  shows $P\ n$

lemma *CHORun_induct*:
  assumes $CHORun\ rho\ HOs\ coords$
  and $initConfig\ (rho\ 0) \implies P\ 0$
  and $\bigwedge r.\ [\![P\ r;\ nextConfig\ r\ (rho\ r)\ (HOs\ r)\ (coords\ r)\ (rho\ (Suc\ r))]\!]$
      $\implies P\ (Suc\ r)$
  shows $P\ n$

4.4 Modeling a Concrete Algorithm: *LastVoting*

We now instantiate the generic Isabelle locale *CHOAlgorithm* for the *LastVoting* algorithm introduced in Section 3. We begin by declaring an anonymous type *Proc* of processes that is assumed to be finite. The number of processes will be denoted by N.

typedecl *Proc*
axioms *procFinite*: *finite* (*UNIV* :: *Proc set*)
abbreviation $N \;\equiv\; card\ (UNIV :: Proc\ set)$

The algorithm proceeds in *phases* that consist of 4 rounds, and we call *steps* the rounds that constitute a phase. Accordingly, we define the following operators *phase* and *step* to compute the phase and step numbers of a given round:

definition *phase* **where** $phase\ (r :: nat) \;\equiv\; r\ div\ 4$

definition *step* **where** $step\ (r :: nat) \;\equiv\; r\ mod\ 4$.

In close correspondence with the informal description (see Algorithm 1), the local state of a process is declared as the following record; the type variable *'val* represents the type of values that processes decide on:

record *'val pstate* =

x :: *'val*	– current value held by process
vote :: *'val option*	– last vote cast by process, if any
commt :: *bool*	– Boolean “commit” flag
ready :: *bool*	– Boolean “ready” flag
timestamp :: *nat*	– time stamp of current value
decide :: *'val option*	– value the process decided, if any

Four kinds of messages are sent by processes, and these are represented by an Isabelle data type as follows:

datatype *'val msg* =

ValStamp 'val nat	– message carrying a value and a time stamp
| *Vote 'val*	– vote for a certain value
| *Ack*	– acknowledgement message
| *Null*	– empty message

We define characteristic predicates *isValStamp*, *isVote*, and *isAck* that recognize the type of a message and selector functions *val* and *stamp* that retrieve the value and time stamp components of messages of appropriate types. We also define the following utility functions that retrieve the set of processes *q* from a partial vector of message such that *q* sent a *ValStamp* message, the highest time stamp carried by any such message, and the set of processes from which an acknowledgement was received.

definition *valStampsRcvd* **where**
$valStampsRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \;\equiv$
$\{q\ .\ \exists v\ ts.\ msgs\ q = Some\ (ValStamp\ v\ ts)\}$

definition *highestStampRcvd* **where**
$highestStampRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \;\equiv$
$Max\ \{ts\ .\ \exists q\ v.\ msgs\ q = Some\ (ValStamp\ v\ ts)\}$

definition *acksRcvd* **where**
$acksRcvd\ (msgs :: Proc \rightharpoonup 'val\ msg) \;\equiv$
$\{q\ .\ msgs\ q = SomeAck\}$

```
definition initState where initState p st ≡
    vote st = None ∧ ¬(commt st) ∧ ¬(ready st)
  ∧ timestamp st = 0 ∧ decide st = None
definition send0 where send0 r p q st crd ≡
  if q = crd then ValStamp (x st) (timestamp st) else Null
definition next0 where next0 r p st msgs crd st' ≡
  if p = crd ∧ card(valStampsRcvd msgs) > N div 2
  then ∃p v. msgs p = Some(ValStamp v (highestStampRcvd msgs))
          ∧ st' = st(|vote := Some v, commt := True|)
  else  st' = st
definition send1 where send1 r p q st crd ≡
  if p = crd ∧ commt st then Vote (the (vote st)) else Null
definition next1 where next1 r p st msgs crd st' ≡
  if msgs crd ≠ None ∧ isVote (the (msgs crd))
  then st' = st(|x := val (the (msgs crd)), timestamp := Suc(phase r)|)
  else  st' = st
definition send2 where send2 r p q st crd ≡
  if timestamp st = Suc(phase r) ∧ q = crd then Ack else Null
definition next2 where next2 r p st msgs crd st' ≡
  if p = crd ∧ card (acksRcvd msgs) > N div 2
  then st' = st(|ready := True|)
  else  st' = st
definition send3 where send3 r p q st crd ≡
  if p = crd ∧ ready st then Vote (the (vote st)) else Null
definition next3 where next3 r p st msgs crd st' ≡
    if msgs crd ≠ None ∧ isVote (the (msgs crd))
    then decide st' = Some (val (the (msgs crd)))
    else  decide st' = decide st
  ∧ if p = crd
    then ¬(ready st') ∧ ¬(commt st')
    else  ready st' = ready st ∧ commt st' = commt st
  ∧ x st' = x st ∧ vote st' = vote st ∧ timestamp st' = timestamp st
```

Figure 2: Isabelle representation of the *LastVoting* algorithm.

With these preliminary definitions, the Isabelle specification of the *LastVoting* algorithm appears in Fig. 2. It closely parallels the pseudo-code description that appears as Algorithm 1 in Section 3, translating the description of each step into a relation between two states *st* and *st'*, given the coordinating process and the partial vector of messages received.

The overall message sending function and next-state relation are defined as compositions of the operators introduced in Fig. 2.

```
definition sendMsg where
  sendMsg r ≡
  if step r = 0 then send0 r
  else if step r = 1 then send1 r
  else if step r = 2 then send2 r
  else send3 r
```

```
definition nextState where
  nextState r ≡
  if step r = 0 then next0 r
  else if step r = 1 then next1 r
  else if step r = 2 then next2 r
  else next3 r
```

It remains to define the communication-coordinator predicate assumed for the *LastVoting* algorithm. As mentioned in Section 3, it is assumed that coordinators remain unchanged for an entire phase, i.e., they may change only between step 3 of a phase and step 0 of the subsequent phase. Second, we formalize the predicate of Theorem 3.1, which requires the existence of a phase ϕ such that:

- all processes agree on the same coordinator c for phase ϕ,
- c hears from a strict majority of processes in steps 0 and 2 of phase ϕ, and
- every process hears from c in steps 1 and 3 of phase ϕ.

These requirements are expressed by the following predicate:

```
definition LV_commPred where
  LV_commPred HOs coords ≡
    (∀r. step r ≠ 3 ⇒ coords(Suc r) = coords r)
  ∧ (∃ph c. (∀p. coords (4 * ph) p = c)
          ∧ card (HOs (4 * ph) c) > N div 2
          ∧ card (HOs (Suc(Suc(4 * ph))) c) > N div 2
          ∧ (∀p. c ∈ HOs (Suc(4 * ph)) p ∩ HOs (Suc(Suc(Suc(4 * ph)))) p))
```

We now have all elements for defining the *LastVoting* algorithm as an instance of the generic locale for CHO algorithms described in Section 4.2. This is achieved by the following Isabelle command; the proof of the locale assumption is discharged by invoking the axiom asserting finiteness of type *Proc*.

```
interpretation CHOAlgorithm initState sendMsg nextState LV_commPred
by (unfold_locales, rule procFinite)
```

```
lemma LV_induct:
  assumes HORun rho HOs coords
  and ∀p. initState p (pstates (rho 0) p) ⟹ P 0
  and ⋀ r.
          ⟦ P r; step r = 0; phase (Suc r) = phase r; step (Suc r) = 1;
            ∀p. next0 r p (rho r p)
                       (rcvdMsgs p (HOs r p) (coords r) (rho r) (send0 r))
                       (coords r p) (rho (Suc r) p) ⟧
          ⟹  P (Suc r)
  and ...      – analogous assumptions for next1, next2, and next3
  shows P n
```

Figure 3: Induction rule for *LastVoting*.

Using the definition of the next-state relation, we specialize the proof rules about runs of HO algorithms presented in Sect. 4.3 for the concrete instance of *LastVoting*. For example, Fig. 3 shows the derived induction rule for executions of the *LastVoting* algorithm.

5 Formal Verification of *LastVoting* in Isabelle

We now give an overview of the formal proof in Isabelle of the correctness of the *LastVoting* algorithm, asserted in Theorem 3.1. In Section 5.1, we begin by proving some elementary facts about timestamps, then present in Section 5.2 a number of "obvious" lemmas about runs of the algorithm. Based on these lemmas, we derive Integrity (Section 5.3), Irrevocability and Agreement (Section 5.4), and Termination (Section 5.5).

Throughout, we reason about (coarse-grained) runs of the *LastVoting* algorithm: each lemma carries the hypothesis *CHORun rho HOs coords*, which we leave implicit in the following. For each lemma, we give its statement in ordinary mathematical language as well as its formal counterpart in Isabelle. We then outline its proof, closely following the reasoning performed in the Isabelle proof script.

5.1 Facts about timestamps

We begin by proving some elementary facts about the timestamps held by the processes executing the *LastVoting* algorithm. Our first lemma states, roughly, that the timestamp of a process at any state during a run is bounded by the current phase. The subsequent Lemma 5.2 asserts that timestamps never decrease.

Lemma 5.1 *The timestamp of any process p at round r is bounded by the current phase ϕ at steps 0 and 1, and by $\phi + 1$ at steps 2 and 3.*

$$\textit{timestamp}\ (\textit{rho}\ r\ p) \le (\textit{if}\ \ \textit{step}\ r < 2\ \textit{then}\ \textit{phase}\ r\ \textit{else}\ \textit{Suc}(\textit{phase}\ r))$$

Proof: The lemma is proved by induction on the run ρ, using lemma *LV_induct* of Figure 3. The assertion is obviously true initially. It is preserved by all transitions. In particular, only *next*1 may modify the timestamp, and it updates it to *Suc*(*phase r*) while passing to step 2 of the current phase. The Isabelle proof consists of a single interaction directing Isabelle to apply the induction rule and expand the definitions of *initState* and of the system transitions *next*0, ..., *next*3. □

Lemma 5.2 *The timestamp of a process never decreases.*

1. $\textit{timestamp}\ (\textit{rho}\ r\ p) \le \textit{timestamp}\ (\textit{rho}(\textit{Suc}\ r)\ p)$
2. $r \le r' \Longrightarrow \textit{timestamp}\ (\textit{rho}\ r\ p) \le \textit{timestamp}\ (\textit{rho}\ r'\ p)$

Proof:

1. The proof considers the possible actions; only action *next*1 is of interest because it may modify the timestamp. By Lemma 5.1, the timestamp of process p is bounded by the phase ϕ, whereas the new timestamp (if it is updated) equals $\phi + 1$. This suffices.
2. The second assertion follows from the first one by induction on natural numbers. □

5.2 Auxiliary lemmas

Before we prove the main correctness properties of algorithm *LastVoting*, we state a few facts about its runs. These facts would typically not appear in a published correctness proof because they would be considered as "immediately obvious" from the pseudo code presentation of the algorithm. Isabelle does not allow us to take such shortcuts: formally, they are established by inductive reasoning over runs.

First, we verify that the function used to determine the vote of a process in step 0 is actually well-defined. Because Isabelle/HOL is a logic of total functions, the definition of *highestStampRcvd* (based on the maximum *Max* of a set) always returns some natural number. However, we must prove that that value satisfies the expected properties.

Lemma 5.3

1. *The set of timestamps contained in a partial vector of messages is finite.*

$$\textit{finite}\ \{ts\ .\ \exists q\ v.\ (\textit{msgs} :: \textit{Proc} \rightharpoonup \textit{'val msg})\ q = \textit{Some}(\textit{ValStamp}\ v\ ts)\}$$

2. *If at least one message of kind ValStamp has been received then some message carries a maximal timestamp.*

   $valStampsRcvd\ msgs \neq \{\}$
   $\quad\Longrightarrow\ \exists p\ v.\ msgs\ p = Some(ValStamp\ v\ (highestStampRcvd\ msgs))$
   $msgs\ q = Some(ValStamp\ w\ ts)$
   $\quad\Longrightarrow\ ts \leq highestStampRcvd\ msgs$

Proof:

1. Follows easily from the instance of lemma *finite_ran* (cf. Section 4.3) for the *LastVoting* algorithm.

2. These assertions are a consequence of the first one because the *Max* operator yields the maximal element of a non-empty finite set of integers. □

The following facts state at which steps of the algorithm the coordinator and the different fields of a process state may change. These facts are proved in a single interaction with Isabelle by considering the different possible transitions and expanding the definitions of *next*0, ..., *next*3.

Lemma 5.4

1. *Coordinators change only at step* 3.

   $step\ r \neq 3\ \Longrightarrow\ coord\ (Suc\ r)\ p = coord\ r\ p$

2. *Votes change only at step* 0.

   $step\ r \neq 0\ \Longrightarrow\ vote\ (rho\ (Suc\ r)\ p) = vote\ (rho\ r\ p)$

3. *The* commit *field of a process changes only at steps* 0 *and* 3.

   $step\ r \notin \{0, 3\}\ \Longrightarrow\ commt\ (rho(Suc\ r)\ p) = commt\ (rho\ r\ p)$

4. *Timestamps change only at step* 1.

   $step\ r \neq 1\ \Longrightarrow\ timestamp\ (rho\ (Suc\ r)\ p) = timestamp\ (rho\ r\ p)$

5. *The x field of a process changes only at step* 1.

   $step\ r \neq 1\ \Longrightarrow\ x\ (rho\ (Suc\ r)\ p) = x\ (rho\ r\ p)$

The remaining “obvious” lemmas express certain invariants about process states and transitions. They are again proved by induction on the runs of the algorithm, but require some more guidance by the user. An alternative to proving these lemmas separately would be to state and prove a global invariant about the algorithm, but we find the present proof more readable.

Lemma 5.5 *The "commit" flag of a process p is set only if the step number is at least* 1*, the vote of process p is non-null, and if there is a strict majority of processes, including p, that consider p to be the coordinator for the current round.*

$$\begin{aligned}&\mathit{commt}\ (\mathit{rho}\ r\ p) \implies \\ &\qquad 1 \leq \mathit{step}\ r \wedge \mathit{coords}\ r\ p = p \wedge \mathit{vote}\ (\mathit{rho}\ r\ p) \neq \mathit{None} \\ &\quad \wedge\ \mathit{card}\ \{q.\ \mathit{coords}\ r\ q = p\} > N\ \mathit{div}\ 2\end{aligned}$$

Proof: The assertion is true initially because the commit flag is not set. Steps 1 and 2 trivially preserve the assertion because they change neither the vote nor the coordinator, step 3 also preserves it because it resets the commit flag of coordinators. It remains to consider step 0. By induction hypothesis, the commit flag cannot be set before the transition. By the definition of action *next*0, it will be set for process p only if p considers itself to be the coordinator and if it has received a majority of messages of kind *ValStamp* – implying, by the definition of *send*0, that a majority of processes consider p to be their coordinator. Moreover, process p sets its vote to a non-null value in the same transition. This establishes the assertion. □

Lemma 5.6 *Process p has a fresh timestamp only at step* 2 *or beyond and only if its coordinator has its "commit" flag set and the current x value of p agrees with the vote of its coordinator.*

$$\begin{aligned}&\mathit{timestamp}\ (\mathit{rho}\ r\ p) = \mathit{Suc}\ (\mathit{phase}\ r) \implies \\ &\qquad \mathit{step}(\mathit{rho}\ r) \geq 2 \wedge \mathit{commt}\ (\mathit{rho}\ r\ (\mathit{coords}\ r\ p)) \\ &\quad \wedge\ x\ (\mathit{rho}\ r\ p) = \mathit{the}\ (\mathit{vote}\ (\mathit{rho}\ r\ (\mathit{coords}\ r\ p)))\end{aligned}$$

Proof: The assertion is true initially because the timestamp of all processes equals 0. It is preserved by step 0 because Lemma 5.1 ensures that the timestamp cannot be fresh. It is preserved by step 2 because none of the state components that occur in the assertion change during that step, and by step 3 because Lemma 5.1 again asserts that the timestamp cannot be fresh in the successor state. It remains to consider step 1. Using Lemma 5.1, it follows that the timestamp cannot be fresh in the state before the transition; it must therefore be set by the transition, and the definitions of *next*1 and *send*1 ensure that in this case the coordinator of process p must have its "commit" flag set, and that the new x value of p is set to the vote of the coordinator. Moreover, neither the coordinator nor its vote or the status of its "commit" flag change during this step, by Lemma 5.4. □

Lemma 5.7 *The "ready" flag of a process p is set only at step* 3*. Also, p in this case considers itself as the coordinator, and there exists a strict majority of processes that consider p as their coordinator and that have a fresh timestamp.*

$$\begin{aligned}&\mathit{ready}\ (\mathit{rho}\ r\ p) \implies \\ &\qquad \mathit{step}\ r = 3 \wedge \mathit{coords}\ r\ p = p \\ &\quad \wedge\ \mathit{card}\{q.\ \mathit{coords}\ r\ q = p \wedge \mathit{timestamp}\ (\mathit{rho}\ r\ q) = \mathit{Suc}\ (\mathit{phase}\ r)\} > N\ \mathit{div}\ 2\end{aligned}$$

Proof: The only interesting case is that of step 2 because it is the only one that sets the "ready" flag. By the definition of *next*2, process p does this only if it considers itself a coordinator and if it has received a strict majority of *Ack* messages. By definition of *send*2, this implies that a majority of processes consider p to be their coordinator and have a fresh timestamp. □

Lemma 5.8 *A process p changes its decision value only if it is at step* 3 *and if the "ready" and "commit" flags of its coordinator are set. Moreover, the new decision value is the vote of its coordinator.*

$$\begin{aligned}&\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r)\ p) \neq \mathit{decide}\ (\mathit{rho}\ r\ p) \implies \\ &\quad \mathit{step}\ r = 3 \wedge \mathit{ready}\ (\mathit{rho}\ r\ (\mathit{coords}\ r\ p)) \wedge \mathit{commt}\ (\mathit{rho}\ r\ (\mathit{coords}\ r\ p)) \\ &\ \wedge \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r)\ p) = \mathit{Some}\ (\mathit{the}\ (\mathit{vote}\ (\mathit{rho}\ r\ (\mathit{coords}\ r\ p))))\end{aligned}$$

Proof: A change of decision can occur only at step 3, and the definitions of *next*3 and *send*3 ensure that the new decision value is the vote of the coordinator, and that the "ready" flag of the coordinator c_p of p is set. By Lemma 5.7, this implies that there exists a strict majority of processes that have c_p as their coordinator and that have a fresh timestamp. Letting q denote an arbitrary element of this (non-empty) majority and using Lemma 5.6, it follows that the coordinator c_q of q has its "commit" flag set. Since $c_q = c_p$, this proves the assertion. □

Lemma 5.9 *If some process p updates its decision value at round r then at all following rounds r+k there exists a strict majority of processes whose timestamp is greater than the phase corresponding to round k.*

$$\begin{aligned}&\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r)\ p) \neq \mathit{decide}\ (\mathit{rho}\ r\ p) \implies \\ &\mathit{card}\ \{q.\ \mathit{timestamp}\ (\mathit{rho}\ (r+k)\ q) > \mathit{phase}\ r\} > N\ \mathit{div}\ 2\end{aligned}$$

Proof: Since process p decides at round r, Lemma 5.8 implies that the "ready" flag of its coordinator is set at round r. By Lemma 5.7, there exists a majority of processes whose timestamp at round r equals *Suc* (*phase* r). Moreover, timestamps never decrease (Lemma 5.2), which implies the assertion. □

5.3 Integrity

The proof of the Integrity property relies on an invariant that asserts that all values that appear in the x, *vote* or *decide* fields of any process during a run are among the initial values of the x fields. In the Isabelle formulation of this invariant, we make use of some notation about functions: $\circ$ denotes function composition, and f ' S denotes the image of set S under the function f.

Lemma 5.10 *All values appearing in the x, vote or decide field of any process, at any round, are among the initial x values chosen by the processes or None (i.e., not set) in the case of vote and decide.*

$$\begin{aligned}&\quad \mathit{range}\ (x \circ (\mathit{rho}\ r)) \subseteq \mathit{range}\ (x \circ (\mathit{rho}\ 0)) \\ &\wedge \mathit{range}\ (\mathit{vote} \circ (\mathit{rho}\ r)) \subseteq \{\mathit{None}\} \cup \mathit{Some}\ `\ \mathit{range}\ (x \circ (\mathit{rho}\ 0)) \\ &\wedge \mathit{range}\ (\mathit{decide} \circ (\mathit{rho}\ r)) \subseteq \{\mathit{None}\} \cup \mathit{Some}\ `\ \mathit{range}\ (x \circ (\mathit{rho}\ 0))\end{aligned}$$

Proof: The assertion clearly holds initially because the *vote* and *decide* fields are initialized to *None*. It is easily seen to be preserved by all transitions because values are only copied: no new value is introduced at any step of the algorithm. The Isabelle proof is quite tedious because we have to show that certain expressions yield the expected value. For example, when step 1 updates the x field of some process p, Lemma 5.5 is used to infer that the coordinator of p has a vote different from *None*, which must therefore be among the initial x values by the induction hypothesis. □

The Integrity property is an immediate consequence of the third conjunct of Lemma 5.10.

Theorem 5.11 (Integrity) *If process p decides some value v, then v is the initial x value of some process q.*

$$decide\ (rho\ n\ p) = Some\ v \implies \exists q.\ v = x\ (rho\ 0\ q)$$

5.4 Irrevocability and Agreement

We establish the Irrevocability and Agreement properties of the *LastVoting* algorithm by proving a series of lemmas that build on the "obvious" facts of Sections 5.1 and 5.2. Again, we find this presentation of the proofs more readable than the alternative approach of stating a global system invariant that implies these properties.

Lemma 5.12 *No two different processes have their "commit" flag set at any round.*

$$commt\ (rho\ r\ p) \wedge commt\ (rho\ r\ p') \implies p = p'$$

Proof: If processes p and p' have their "commit" flag set, Lemma 5.5 tells us that there is a majority of processes that consider p (resp., p') as their coordinator. Applying lemma *majoritiesE*, we obtain some process q in the intersection of these two majority sets. Since q considers both p and p' as its coordinator, they must be equal. □

As a concrete example for a proof in Isabelle/Isar, we reproduce the Isabelle input for the proof of Lemma 5.12 in Figure 4. At each proof step, we state the assertion that we wish to prove, and the assumptions necessary to prove it. The **by** clauses invoke Isabelle's proof methods (*blast*, *auto*, and *simp* in this example), indicating when necessary auxiliary theorems to be used (*commitE* is Lemma 5.5 and *majoritiesE* was mentioned in Section 4.3). This example is quite representative of the level of detail that is necessary to actually carry out these proofs in Isabelle.

Lemma 5.13 *No two different processes have their "ready" flag set at any round.*

$$ready\ (rho\ r\ p) \wedge ready\ (rho\ r\ p') \implies p = p'$$

```
lemma committedProcsEqual:
  assumes run: CHORun rho HOs coords
  and cmt: commt (rho r p) and cmt′: commt (rho r p′)
  shows p = p′
proof –
  from run cmt have card {q . coords r q = p} > N div 2
    by (blast elim: commitE)
  moreover
  from run cmt′ have card {q . coords r q = p′} > N div 2
    by (blast elim: commitE)
  ultimately
  obtain q where p = coords r q and coords r q = p′
    by (auto elim: majoritiesE)
  thus p = p′ by simp
qed
```

Figure 4: Lemma 5.12 and its proof in Isabelle/Isar.

Proof: The proof is analogous to that of Lemma 5.12, using Lemma 5.7 instead of Lemma 5.5. ☐

The following lemma restricts the values a coordinator may vote for: if there is a majority of processes whose timestamp is beyond *ts*, then a coordinator may only vote a value held by one of these processes. This is a consequence of the choice of a timing criterion for determining the vote.

Lemma 5.14 *Assume that a majority set M of processes hold timestamps beyond ts, and that the "commit" flag of process p is set. Then the vote of process p agrees with the x value of some process $q \in M$.*

$$\llbracket card\{q.\ timestamp\ (rho\ r\ q) \geq ts\} > N\ div\ 2;\ commt\ (rho\ r\ p)\rrbracket \implies \exists q.\ timestamp\ (rho\ r\ q) \geq ts\ \wedge\ vote\ (rho\ r\ p) = Some\ (x\ (rho\ r\ q))$$

Proof: The assertion holds initially because no process has its "commit" flag set. If process p commits at step 0, it has received messages of kind *ValStamp* from a majority M' of processes and updates its vote to some value v that comes with a highest timestamp. By Lemma 5.3 and the definition of *send*0, this value v indeed agrees with the x field of some process q that has a maximal timestamp among the processes in M'. Moreover, M and M' have a non-empty intersection, hence there exists some process $q' \in M \cap M'$. Since q has a maximal timestamp among processes in M', we obtain

$$timestamp\ (rho\ r\ q) \geq timestamp\ (rho\ r\ q')$$

and because of the definition of M we know

$$timestamp\ (rho\ r\ q') \geq ts.$$

Taken together, the assertion follows.

We now prove that the assertion is preserved at step 1. Indeed, assume that there is a majority M of processes holding timestamps beyond ts after the step 1 transition (during which some processes may update their timestamps). We consider two cases: if no process in M updated its local state during the step 1 transition, then all processes in M already held timestamps beyond ts before the transition, and since the vote of p doesn't change during step 1 (Lemma 5.4), the assertion follows from the induction hypothesis. Otherwise, let $q \in M$ be some process that updates its local state during step 1. Then the definitions of $next1$ and $send1$ ensure that q sets its x field to the vote of the coordinator, and the assertion again follows.

Step 2 preserves the assertion because neither timestamps nor votes nor commit values change during this transition (Lemma 5.4), and step 3 preserves it because the commit flag cannot be set in the post-state by Lemma 5.5. □

The following lemma gives the crucial argument for establishing Irrevocability and Agreement: if some process p decides value v during phase ϕ then any process q whose timestamp is greater than ϕ must hold v in its x field.

Lemma 5.15 *Assume that process p decides at round r and that process q, at some later round $r + k$, has a timestamp strictly greater than the phase at round r. Then the value of the x field of process q at round $r + k$ agrees with the decision value of p.*

$$\begin{array}{l}[\![decide\ (rho\ (Suc\ r)\ p) \neq decide\ (rho\ r\ p); \\ \ timestamp\ (rho\ (r+k)\ q) > phase\ r]\!] \\ \Longrightarrow\ x\ (rho\ (r+k)\ q) = the\ (decide\ (rho\ (Suc\ r)\ p))\end{array}$$

Proof: We will write ϕ for *phase* r, i.e., the phase at which process p decides, and v for *decide* $(rho\ (Suc\ r)\ p)$, i.e., the decision value.

The assertion of the lemma is proved by induction on $k \in \mathbb{N}$, simultaneously for all processes q. For $k = 0$, by the assumption that process p decides at round r and Lemma 5.8 it follows that v must be the vote of its coordinator, whose "commit" flag is set. Moreover, the assumption on the timestamp of q and Lemma 5.1 implies that the timestamp of q equals $\phi+1$, and by Lemma 5.6, the x value of q agrees with the vote of its coordinator, whose "commit" flag is also set. Now, Lemma 5.12 implies that the coordinators of p and q are the same, and the assertion follows.

For the inductive step, we need only consider step 1 because the other steps leave the timestamps and x values unchanged. If process q does not update its local state during step 1, the assertion is trivially preserved. Otherwise, the definitions of $next1$ and $send1$ imply that process q updates its x field to the vote of its coordinator c_q, which has its "commit" flag set. Moreover, by Corollary 5.9 there is a majority M of processes whose timestamps at round $r+k$ are greater than ϕ, and Lemma 5.14 implies that the vote of c_q at round $r + k$ agrees with the x field of some process $q' \in M$. By the induction hypothesis (for process q'), the x field of process q' must hold value v, which proves the assertion. □

It follows that if two processes p and q decide at rounds r and $r+k$, then their decision values agree.

Lemma 5.16 *Assume that process p decides at round r, and that process q decides at round $r+k$, for some $k \in \mathbb{N}$. Then they decide the same values.*

$$\begin{array}{l} [\![\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r)\ p) \neq \mathit{decide}\ (\mathit{rho}\ r\ p); \\ \ \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ (r+k))\ q) \neq \mathit{decide}\ (\mathit{rho}\ (r+k)\ q);]\!] \\ \Longrightarrow\ \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ (r+k))\ q) = \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r)\ p) \end{array}$$

Proof: By Lemma 5.8, process q decides on the vote of its coordinator c_q, which has its “commit” flag set. Also, since process p decided at round r (say, in phase ϕ_r), by Lemma 5.9 a majority M of processes at round $r+k$ hold timestamps beyond ϕ_r. Lemma 5.14 ensures that c_q voted for the value held in the x field of some process in M, which by Lemma 5.15 must be the decision value of process p. □

Lemma 5.17 *Any process that holds a non-null decision value must have decided that value in the past.*

$$\begin{array}{l} \mathit{decide}\ (\mathit{rho}\ r\ p) = \mathit{Some}\ v \\ \Longrightarrow\ \exists r' < r.\ \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r')p) \neq \mathit{decide}\ (\mathit{rho}\ r'\ p) \\ \qquad\qquad\quad \wedge\ \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r')p) = \mathit{Some}\ v \end{array}$$

Proof: Assuming to the contrary that for all $r' < r$, whenever the decision value of p changed, it was not set to *Some* v, the lemma is proved by induction on r, observing that $\mathit{decide}\ (\mathit{rho}\ 0\ p) = \mathit{None}$. □

We now have established all arguments needed to prove the Irrevocability and Agreement properties for the *LastVoting* algorithm.

Theorem 5.18 (Irrevocability) *Once a process decides a value, it remains decided on that value.*

$$\mathit{decide}\ (\mathit{rho}\ r\ p) = \mathit{Some}\ v\ \Longrightarrow\ \mathit{decide}\ (\mathit{rho}\ (r+k)\ p) = \mathit{Some}\ v$$

Proof: Lemma 5.17 implies that there exists some round $r' < r$ at which process p decided value v. It suffices to prove that $\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r'+k)\ p) = \mathit{Some}\ v$ for all k, which is shown by induction on $k \in \mathbb{N}$. The induction basis is obvious. For the induction step, assume to the contrary that

$$\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r' + \mathit{Suc}\ k)\ p) \neq \mathit{Some}\ v$$

Hence, process p changes its decision value at round $\mathit{Suc}\ r'+k$, and Lemma 5.16 (for $q = p$) implies that

$$\mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r' + \mathit{Suc}\ k)\ p) = \mathit{decide}\ (\mathit{rho}\ (\mathit{Suc}\ r')\ p) = \mathit{Some}\ v,$$

and a contradiction is reached. □

Theorem 5.19 (Agreement) *No two processes ever decide differently.*

$$\llbracket decide\ (rho\ r\ p) = Some\ v;\ decide\ (rho\ r'\ q) = Some\ w \rrbracket \implies v = w$$

Proof: By Lemma 5.17 there exist rounds r_p and r_q at which p and q decide v and w, respectively. Without loss of generality, assume that $r_p \leq r_q$, then Lemma 5.16 implies that $decide\ (rho\ (Suc\ r_q)\ q) = decide\ (rho\ (Suc\ r_p)\ p)$, hence $v = w$. □

5.5 Termination

The proof of Termination makes essential use of the communication-coordinator predicate $\mathcal{P}_{LastVoting}$ given in Theorem 3.1. This predicate assumes that there exists some phase ϕ and some process c that is the coordinator of all processes during phase ϕ and such that c receives messages from a majority of processes in steps 0 and 2 of phase ϕ, while all processes receive messages from c in steps 1 and 3 of phase ϕ. This assumption ensures that all processes will have decided at the end of phase ϕ.

Theorem 5.20 (Termination) *There exists some round r at which all processes have decided.*

$$\exists r.\ \forall p.\ decide\ (rho\ r\ p) \neq None$$

Proof: From the communication-coordinator predicate, there exist $\phi \in \mathbb{N}$ and a process c such that all of the following conditions hold:

1. $\forall p.\ coords\ (4 * \phi)\ p = c$
2. $|HOs\ (4 * \phi)\ c| > N\ div\ 2$
3. $|HOs\ (4 * \phi + 2)\ c| > N\ div\ 2$
4. $\forall p.\ c \in HOs\ (4 * \phi + 1)\ p$
5. $\forall p.\ c \in HOs\ (4 * \phi + 3)\ p$

Moreover, the communication-coordinator predicate implies that coordinators do not change during phases, and therefore condition 1 implies

6. $\forall p.\ coords\ (4 * \phi + s)\ p = c$ for all $s \in \{0, 1, 2, 3\}$.

Assumptions 1 and 2 imply that c receives *ValStamp* messages from a majority of processes at step 0 of phase ϕ and, by definition of *next*0, has its "commit" flag set at step 1. It therefore sends a vote to all processes at step 1, which by assumptions 6 and 4 is received by all processes. According to the definition of *next*1, every process updates its timestamp to $\phi + 1$ at that step.

All processes therefore send acknowledgments at round $4 * \phi + 2$, and the assumptions 6 and 3 ensure that c receives a majority of acknowledgements at that step and therefore sets its "ready" flag, according to the definition of *next*2. It sends its vote in step 3 of phase ϕ to all processes, and assumptions 6 and 5 ensure that this message is received by all processes, which therefore set their decision field to the vote received, according to the definition of *next*3. It follows that $\mathit{decide}\ (4 * \phi + 4)\ p \neq \mathit{None}$ for all p, and this proves the theorem. □

6 Conclusion and Future Work

Distributed algorithms are reputedly difficult to design and to verify. Several published algorithms have been found to be erroneous or have been applied in contexts for which they were not designed because the underlying hypotheses were not correctly specified. Formal techniques of development and verification offer notation to describe algorithms and state their properties, as well as support for verification techniques based on precise semantic analyses. So far, algorithm designers have been reluctant to adopt these methods, especially in the context of distributed systems and fault-tolerance, probably because they are often tedious to apply and do not scale well to interesting algorithms.

We believe that the use of formal techniques is crucial in distributed computing, but that the models used for reasoning about distributed algorithms must be adapted in order to address the scalability issue. In this paper, we have studied the Heard-Of model, a computational model of distributed algorithms suggested by Charron-Bost and Schiper, with respect to its amenability to formal verification. The main advantage of this model from this point of view is the fact that many properties of algorithms can be proved over a coarse abstraction of runs in which the *round* of the overall system is the basic entity. Compared to a fine-grained representation of runs, we do not need to represent the network state at all in a run – every channel can be considered empty at the beginning of each round since rounds are communication-closed layers –, and we have much fewer events to consider and reason about. As already noted by Tsuchiya and Schiper [17, 18], it therefore becomes possible to derive finite-state models of instances of HO algorithms for a fixed finite number of processes and to apply finite-state model checking techniques. Indeed, the local state of a process in a fine-grained representation of an HO algorithm must necessarily contain the round number in order to determine whether an incoming message was sent for the current round of the process. Moreover, differences between round numbers of different processes at a system configuration can become arbitrarily large in HO algorithms, implying that no bounded representation of round numbers is possible. In the coarse-grained abstraction, all processes execute the same round, and there is no need to verify that a message is fresh. Besides, many algorithms, such as *LastVoting*, do not require exact round numbers, and finite-state representations can be obtained.

In this paper, we have focused on theorem proving techniques that allow a designer to formally prove the correctness of an algorithm. As a concrete case study, we have verified the *LastVoting* algorithm, which is the HO version of Lamport's *Paxos*. Because the algorithms are quite similar, our proof can be compared to a previous verification effort [9] of *DiskPaxos*, also in Isabelle/HOL. The Isabelle sources of the *LastVoting* proof are at least five times shorter, while including much more explanatory text, containing the development of a generic Isabelle model of HO algorithms that should be reusable in different contexts, and covering liveness as well as safety properties. Subjectively, we find the proof of *LastVoting* incomparably more readable than the proof of *DiskPaxos*, which relies on an invariant structured in six conjuncts that collectively take up about five typeset pages.

It is a remarkable property of *Paxos* and *LastVoting* that they are always safe, that is to say, safety requires no particular assumption. This is simply captured in the HO model by the fact that no communication predicate at all is needed for the proof of Integrity, Agreement, and Irrevocability of *LastVoting*, which makes the proof of these safety conditions easier. Theorem 3.1 states a sufficient communication predicate for achieving liveness.

Our long-term goal is to develop a proof library for the verification of distributed algorithms that will help formal methods in general and theorem proving in particular become as accepted in distributed computing as it is becoming in the programming language and semantics community.

In particular, the generic locale for CHO algorithms should be extended to provide a basic library of communication predicates, their relationships, and their consequences. We are working towards a formalization of the reduction result of Section 2.5, including a precise characterization of the properties that are guaranteed to be preserved. Further work could be directed towards proof support for malicious failures in distributed algorithms.

References

[1] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[3] M. Chaouch-Saad, B. Charron-Bost, and S. Merz. A reduction theorem for the verification of round-based distributed algorithms. In O. Bournez, editor, *3rd Workshop on Reachability Problems (RP'09)*, volume 5797 of *Lecture Notes in Computer Science*, pages 93–106, Palaiseau, France, 2009. Springer.

[4] B. Charron-Bost and A. Schiper. The Heard-Of model: Computing in distributed systems with benign failures. *Distributed Computing*, 2009. To appear.

[5] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[6] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[7] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science Comp. Prog.*, 2(3), Apr. 1982.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[9] M. Jaskelioff and S. Merz. Proving the correctness of Disk Paxos. Archive of Formal Proofs, `http://afp.sourceforge.net/entries/DiskPaxos.shtml`, 2005.

[10] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[11] L. Lamport. The part-time parliament. *ACM Trans. Comp. Syst.*, 16(2):133–169, 1998.

[12] L. Lamport. *Specifying Systems*. Addison-Wesley, Boston, Mass., 2002.

[13] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *6th Symp. Princ. Dist. Comp. (PODC'87)*, pages 137–151. ACM, 1987.

[15] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] N. Santoro and P. Widmayer. Time is not a healer. In *6th Symp. Theor. Aspects of Comp. Sci. (STACS 1989)*, volume 349 of *Lecture Notes in Computer Science*, pages 304–313, Paderborn, Germany, 1989. Springer.

[17] T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. In *26th IEEE Symp. Reliable Distributed Systems (SRDS 2007)*, pages 137–148, Beijing, China, 2007. IEEE Computer Society.

[18] T. Tsuchiya and A. Schiper. Using bounded model checking to verify consensus algorithms. In G. Taubenfeld, editor, *22nd Intl. Symp. Dist. Comp. (DISC 2008)*, volume 5218 of *Lecture Notes in Computer Science*, pages 466–480, Arcachon, France, 2008. Springer.

[19] M. Wenzel and L. Paulson. Isabelle/Isar. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*, pages 41–49. Springer, Heidelberg, 2006.

[20] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.