Model Checking of Consensus Algorithms

Tatsuhiro Tsuchiya[1,2] André Schiper[1*]
[1] École Polytechnique Fédérale de Lausanne (EPFL)
[2] Osaka University

Abstract

We show for the first time that standard model checking allows one to completely verify asynchronous algorithms for solving *consensus*, a fundamental problem in fault-tolerant distributed computing. Model checking is a powerful verification methodology based on state exploration. However it has rarely been applied to consensus algorithms, because these algorithms induce huge, often infinite state spaces. Here we focus on consensus algorithms based on the *Heard-Of model* (HO model, for short), a new computation model for distributed computing. By making use of the high abstraction level provided by this computation model, we develop a methodology for verifying consensus algorithms in every possible state by model checking. This paper describes the proposed verification methodology and the results of applying it to various consensus algorithms.

Keywords: Consensus, Heard-Of (HO) model, model checking, fault-tolerant distributed systems, verification

1 Introduction

Asynchronous fault-tolerant distributed algorithms are typically difficult to design; inherent asynchrony and concurrency make them highly error-prone. The goal of our research is to alleviate this problematic situation by providing a means of automatic verification for these algorithms.

Recently, a new computation model for asynchronous fault-tolerant distributed systems, called the *Heard-Of model* (HO model for short), was proposed [6]. The HO model can capture the synchrony degree and any type of non-malicious faults in a unified manner, and thus provides a general framework for designing and reasoning about fault-tolerant distributed algorithms.

This paper presents our attempt to mechanically verify HO model-based algorithms. Specifically, we focus on algorithms for solving the *consensus problem*, a fundamental problem in fault-tolerant distributed computing. As a verification approach, we use *model checking*. In model checking a system to be verified is first represented as a finite state machine and then verified against a temporal logic specification through state exploration. A remarkable advantage of model checking over other formal verification methods is that it is fully automatic and its application requires no user supervision or expertise in mathematical reasoning.

Although model checking has been widely practiced, there is little work on applying it to the verification of asynchronous distributed algorithms for consensus. A plausible reason for this is that these algorithms induce huge, often infinite, state spaces, thereby

*Research funded by the Swiss National Science Foundation under grant number 200021-111701.

severely limiting the usefulness of model checking techniques. Sources that yield infinite state spaces include unbounded round numbers and unbounded message channels, which are both typical for asynchronous distributed systems/algorithms.

By restricting to finite models with a fixed number of processes and a fixed number of rounds, one could apply standard model checking to the verification of asynchronous consensus algorithms. Clearly, this approach can only be used for detecting errors that manifest themselves in early rounds; nothing conclusive can be obtained if no errors are detected.

In previous work [12, 14, 16], therefore, model checking was not used as a stand alone method, but in conjunction with other mathematical proof techniques. In [14], a shared memory-based randomized consensus algorithm was verified. The authors of [14] separated the algorithm into a probabilistic component and a non-probabilistic component. They applied standard probabilistic model checking techniques to the probabilistic component. For the verification of the non-probabilistic part, whose state space is infinite, they used proof techniques that reduce the verification problem to small problems that can be solved by model checking. In [12, 16], model checking was used for debugging purposes in developing the mathematical proofs for some of the *Paxos* consensus algorithms. The models that were model checked consisted of two or three processes and a small number of rounds [17]. Such small-sized models cannot be used to ensure that the algorithm is correct but are sufficient for detecting simple bugs. The applications of formal verification methods other than model checking to consensus algorithms can be found in [11, 13, 19, 20, 22].

The work presented in this paper is different from the previous work in that our approach does not rely on any other formal verification techniques than model checking. As a result, the verification can be carried out in a fully automatic manner. Also, we fix the number of processes but do not impose any restrictions on the number of rounds; thus our verification is complete in the sense that it verifies the behavior of algorithms in every possible state. *To the best of our knowledge, this is the first time standard model checking allows one to completely verify asynchronous consensus algorithms.*

We should remark that this becomes possible largely due to the high abstraction level provided by the HO model. In the HO model, for example, the computation consists of asynchronous communication-closed rounds where every message sent but not received in the same round is lost. Thus, when model checking HO model-based algorithms, one no longer has to explicitly consider messages buffered in the channels. However, the state space can be infinite especially when the algorithm uses timestamps, because the number of rounds is unbounded. As shown later, we devise a technique for dealing with such infinite state spaces, as well as other modeling and optimization techniques.

Unlike mathematical proving, our approach can only be applied to the case where the number of processes is fixed to a small value and thus, it cannot provide a correctness proof for the general case. On the other hand, our approach is fully automatic and, if the design fails to satisfy a desired property, can produce a counterexample, which is particularly important in finding subtle errors. Both approaches are therefore complementary.

This paper is structured as follows. In Section 2, we describe the HO model and the consensus problem. In Section 3, we briefly explain the concept of model checking as well as NuSMV [8], the model checker used throughout this research. In Section 4, we present how one can model check HO model-based consensus algorithms using a particular algorithm as an example. In Section 5, several techniques are introduced to apply the proposed approach to various algorithms with different characteristics. We conclude the paper with a brief summary and future work in Section 6.

2 The HO Model and the Consensus Problem

2.1 The HO Model

We consider a distributed system consisting of n processes. Let $\Pi = \{p_0, p_1, \cdots, p_{n-1}\}$ be the set of the processes. The *Heard-Of (HO) Model* is a general computational model, suitable for describing any type of systems with benign faults [6]. The two notable features of this model are that (1) synchrony degree and fault model are encapsulated in the same abstract structure, namely the *Heard-Of (HO) sets*, and (2) the notion of faulty component has totally disappeared; instead, only the effects of faults are specified in the form of transmission faults.

In the HO model an algorithm runs in rounds. Without loss of generality, we assume that an algorithm starts at round one. Each round consists of three parts: *send*, *receive*, and *state transition*. Every process sends messages to all processes, then receives a subset of the messages sent, and finally makes a state transition based on the current state and the messages it received. We denote by $HO(p_i, r)(\subseteq \Pi)$ the HO set for process p_i in round r. A process p_i receives the message sent by a process p_j iff $p_j \in HO(p_i, r)$. That is, $HO(p_i, r)$ is the set of processes that p_i has "heard of" in round r. A transmission fault from p_j to p_i in round r is notified by the fact that p_j does not belong to $HO(p_i, r)$.

There can be various reasons for transmission faults. For example, messages may have been lost because they missed a round due to the asynchrony of communication and processing. Process or link faults can also cause transmission faults. The key is that the HO model captures the synchrony degree and faulty components in a unified manner by means of the HO sets, without attributing transmission faults to specific causes.

2.2 The Consensus Problem

The *consensus problem* is recognized as a fundamental problem to solve when one has to design a fault-tolerant distributed system. In this problem, each process is assumed to have a proposed value at the beginning of the algorithm execution and is required to eventually decide on some value. In the HO model the problem is specified by the following three conditions [6]:

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

Agreement No two processes decide differently.

Termination All processes eventually decide.

It should be noted that the termination property requires that all processes decide, since there is no notion of faulty processes in the HO model. A detailed discussion of this issue can be found in [6].

We assume that a process chooses its proposed value from a set V and that each process p_i has a special variable d_i whose domain is $V \cup \{?\}$ where ? is a special value that is not contained in V. Variable d_i is initially ? and p_i decides on a value $v \in V$ by setting d_i to v. By convention, we denote the assignment of v to d_i by DECIDE(v) and omit an explicit reference to d_i in the pseudo-codes presented in this paper. We assume that a process chooses its proposed value from a set V which is an arbitrary set of totally ordered elements.

As a running example, we consider the *UniformVoting* algorithm [6] (Algorithm 1). This algorithm can be viewed as a deterministic version of the well-known randomized consensus

Algorithm 1 The *UniformVoting* algorithm [6]

```
1: Initialization:
2:     x_p ∈ V, initially v_p    {v_p is the initial value of p}
3:     vote_p ∈ V ∪ {?}, initially ?

4: Round r = 2φ − 1:
5:     S_p^r :
6:         send ⟨ x_p ⟩ to all processes

7:     T_p^r :
8:         if at least one ⟨v⟩ is received then
9:             x_p := smallest v received
10:        if all the values received are equal to v then
11:            vote_p := v

12: Round r = 2φ:
13:    S_p^r :
14:        send ⟨ x_p , vote_p ⟩ to all processes

15:    T_p^r :
16:        if at least one ⟨ ∗ , v ⟩ with v ≠? is received then
17:            x_p := v
18:        else
19:            x_p := smallest w from ⟨ w , ? ⟩ received
20:        if all the messages received are
21:                        equal to ⟨ ∗ , v ⟩ with v ≠? then
22:            DECIDE(v)
23:        vote_p := ?
```

algorithm by Ben-Or [1]. The *UniformVoting* algorithm shares many features with other HO model-based consensus algorithms. For example, these algorithms run in *phases*, each of which consists of one or more rounds. We let m denote the number of rounds in one phase.[1] For the *UniformVoting* algorithm, one phase is composed of $m = 2$ rounds. Each round r starts with the *send* part denoted by S_p^r. Each process p then receives messages from processes, which defines $HO(p, r)$. Finally, processes execute the *state transition* part denoted by T_p^r.

Since the HO model represents the degree of synchrony and fault model by the HO sets, system's characteristics can be captured by a condition on the HO sets. It is well known that no deterministic consensus algorithm is possible in pure asynchronous systems prone to failures [10]. In general, therefore, consensus algorithms based on the HO model are intended to work when a certain condition holds on the HO sets. The *UniformVoting* algorithm, for example, assumes that both of the following conditions are met.

- *No Split Rounds*: $\forall r > 0, \forall p_i, p_j \in \Pi,\ HO(p_i, r) \cap HO(p_j, r) \neq \emptyset$.
- *Space Uniform Round*: $\exists r_0 > 0, \forall p_i, p_j \in \Pi,\ HO(p_i, r_0) = HO(p_j, r_0)$.

The first condition ensures agreement, while the second condition ensures termination. When the transition part T_p^r is executed, the messages available guarantee that the conditions on the HO sets hold.

In contrast to agreement and termination, integrity is trivially satisfied and this is usually the case for most consensus algorithms. Thus we limit our discussion to the verification of agreement and termination in the rest of this paper. Also, we will not explicitly verify the possibility that the same process makes different decisions in different rounds, because it is straightforward to modify any algorithm to avoid such a situation (adding an extra boolean variable per process is sufficient).

3 Symbolic Model Checking

Model checking is the process of exploring a finite state transition system to determine whether or not a given temporal property holds. Formally a finite state transition system is a 3-tuple (S, I, R) where S is a set of states, I is a set of initial states, and $R \subseteq S \times S$ is a transition relation. R must be total, that is, for every state $s \in S$ there is a state $s \in S$ such that $(s, s') \in R$. A *computation path* is defined as an infinite sequence of states $s_0, s_1, \cdots$

[1] In *Paxos* [15] and in [5], a round is decomposed in phases. "Round" and "phase" are swapped here to use the classical terminology [9].

such that $(s_i, s_{i+1}) \in R$ for any $i \geq 0$. In the process of model checking, a given temporal property is evaluated with respect to all the initial states.

The major problem with model checking is that the state spaces arising from practical problems are often extremely large, generally making exhaustive exploration not feasible. One of the most successful approaches to this problem is the use of *symbolic* representations of the state space. In *symbolic model checking* [18], boolean functions represented by *Binary Decision Diagrams* (BDDs) are used to represent the state space, instead of, for example, explicit adjacency-lists. This can reduce dramatically the memory and time required because BDDs represent many frequently occurring boolean functions very compactly.

We use the NuSMV Version 2 model checker [8]. NuSMV is one of the latest and most successful symbolic model checkers. NuSMV takes a program written in its own input language as input and outputs the verification results for given temporal specifications.

A NuSMV program consists of variables that have finite domains. The set of states, S, is the Cartesian product of these domains. Each valuation to these variables corresponds to a unique state in S. To avoid confusion, we refer to the variables occurring in NuSMV programs as *program variables* and the variables used in HO model-based algorithms as *process variables*.

In the NuSMV input language, the transition relation is described either by parallel assignments to the next version of the program variables, or by propositional constraints on the reachable state set and the transition relation, or both. Assignments of next values to the variables are preceded by the keyword `ASSIGN`. The expression `next(x)` is used to refer to the program variable `x` in the next state. The `TRANS` keyword is used to declare a constraint on the transition relation, while the `INVAR` keyword is used to specify a constraint on the reachable states.

Initial states are assigned by specifying the initial values of the program variables using the expression `init(x)` where `x` is a program variable. Assignments to the initial values must be preceded by `ASSIGN`.

NuSMV supports CTL and LTL as temporal specification logics; however we stick to using CTL because CTL model checking requires much less computational complexity than LTL model checking [23]. Here we only use two temporal operators, AG and AF. The formula AG g holds in state s if g holds in all states along all computation paths starting from s, while the formula AF g holds in state s if g holds in some state along all computation paths starting from s. An atomic state formula is a CTL formula. If g_1 and g_2 are CTL formulae, then so are $\neg g_1$, $g_1 \wedge g_2$, $g_1 \vee g_2$, AF g_1, and AG g_1.

A given CTL formula is evaluated with respect to all the initial states as follows: First, the set of all reachable states is computed by performing a forward search from the set of the initial states. This step does not affect the correctness of the result but often improves the performance, because it can allow the model checker to limit the state search performed thereafter to the reachable states. In the next step, the set of states where the given temporal property holds is computed. This is done by computing in turn the state set satisfying each subformula of the CTL formula in a bottom-up manner. Finally, whether the set obtained contains all initial states is determined. If it contains all the initial states, then the system meets the correctness property.

If the CTL formula is of the form of AG g where g contains no temporal operator, the first step (that is, reachability analysis) suffices to check that formula. In NuSMV the `-AG` option enables this search, making it possible to skip the remaining, time-consuming steps. In this work we always use this option whenever it can be applied.

4 The Proposed Model Checking Approach

In this section we show how one can model the behavior of an HO model-based consensus algorithm as a finite state transition system so that model checking can be applied to the verification of the algorithm.

4.1 Program Variables

Program variables determine the state space S. Since different configurations of process states must be distinguished in S, we need at least program variables that correspond to the process variables.

Some of the process variables usually have V as their domain (see Algorithm 1). Since V can be arbitrarily large, it is necessary to represent it by a set of small size. Since integrity usually trivially holds and at most n distinct values can be proposed at a time, we substitute a set of n values $\{0, 1, \cdots, n-1\}$ for V. In other words, the elements of $\{0, 1, \cdots, n-1\}$ can be viewed as symbolic values representing any of at most n distinct values taken from V.

Additional program variables are used to represent the current round and the HO sets. The HO model-based algorithms we consider here run in phases each of which consists of $m(\geq 1)$ rounds. The variable ro is used to represent the current round. Specifically, the domain of ro is $\{0, 1, \cdots, m-1\}$ and round $m\phi - ro$ for some phase ϕ is represented by ro. Hence the value of ro changes as $m-1, m-2, \cdots, 0, m-1, m-2, \cdots$.[2] The HO set for process p_i is represented by n boolean variables $h_{i,0}, h_{i,1}, \cdots, h_{i,n-1}$ such that $h_{i,j} = true$ iff p_j belongs to the HO set for p_i in the current round.

For the *UniformVoting* algorithm, for instance, the following program variables are used to define the state transition system:

- $x_i \in \{0, 1, \cdots, n-1\} \quad (i = 0, 1, \cdots, n-1)$.
- $vote_i \in \{0, 1, \cdots, n-1\} \cup \{?\} \quad (i = 0, 1, \cdots, n-1)$.
- $d_i \in \{0, 1, \cdots, n-1\} \cup \{?\} \quad (i = 0, 1, \cdots, n-1)$.
- $ro \in \{0, 1\}$
- $h_{i,j} \in \{true, false\} \quad (i = 0, 1, \cdots, n-1,\ j = 0, 1, \cdots, n-1)$.

The NuSMV code for declaration of these variables in the case $n = 3$ is shown below. The value ? is represented as `-1` to avoid type conflicts.

```
MODULE main
VAR
--  Process variables
  x0 : {0, 1, 2};
  x1 : {0, 1, 2};
  x2 : {0, 1, 2};

  vote0 : {0, 1, 2, -1}; -- -1 stands for ?
  vote1 : {0, 1, 2, -1};
  vote2 : {0, 1, 2, -1};

  d0 : {0, 1, 2, -1}; -- -1 stands for ?
```

[2] For Algorithm 1, we have $m = 2$. In this case, $ro = 1$ represents rounds 1, 3, 5, etc.; $ro = 0$ represents rounds 2, 4, 6, etc.

```
  d1 : {0, 1, 2, -1};
  d2 : {0, 1, 2, -1};

--  Round
  ro : {0, 1};

-- HO sets: hij = true iff i hears of j
  h00 : boolean; h01 : boolean; h02 : boolean;
  h10 : boolean; h11 : boolean; h12 : boolean;
  h20 : boolean; h21 : boolean; h22 : boolean;
```

Using these variables we construct the state transition system (S, I, R) as follows. A state in S represents: (i) one of the rounds constituting a phase, (ii) the states of the processes at the beginning of that round, and (iii) the HO sets in the round. The set of initial state in I contains all states that correspond to round one. A transition $(s, s') \in R$ exists iff s' represents: (i) the round next to the round represented by s, (ii) the process states yielded by a round of algorithm execution from the process states and the HO sets represented by s, and (iii) HO sets that can occur in the round represented by s'. In the rest of this section, we describe how one can specify the state transition system in the context of NuSMV.

4.2 Representing Algorithms

Here we explain how to describe the behavior of an algorithm by specifying the transition of the values of program variables except $h_{i,j}$. The representation of HO sets using $h_{i,j}$ will be discussed in Section 4.3.

Initial States

The initial states of the state transition system represent the states of the processes when an algorithm starts. Process variables are initialized as specified in a given algorithm. In NuSMV, if no initial value is assigned to a variable, that variable can take any value in its domain in the initial state. For the case of the *UniformVoting* algorithm, this applies to the variables x_i, because a process can propose any value in V. The value of ro is initially $m - 1$, where m is the number of rounds comprising a phase. Below is the NuSMV code fragment that specifies the initial states for the *UniformVoting* algorithm.

```
ASSIGN
  init(vote0) := -1; init(d0) := -1;
  init(vote1) := -1; init(d1) := -1;
  init(vote2) := -1; init(d2) := -1;
  init(ro) := 1;
```

Algorithm Execution

Now we show how to express the transition of the values of these variables. First, let us consider ro. Remember that ro is used to show that the current state represents round $m\phi - ro$ for some phase ϕ. Thus ro changes cyclically as $m-1, m-2, \cdots, 1, 0, m-1, m-2, \cdots$. That is, $ro' = m - 1$ if $ro = 0$; $ro' = ro - 1$ otherwise.[3] In NuSMV this can be specified using a `case` expression as follows:

[3]Remember that a primed variable is used to refer to a variable of the next state (see Section 3). Thus ro' represents the value of ro at the next state. In the NuSMV language, `next` is used to represent a next state variable.

```
next(ro) :=
  case
  ro = m - 1 : m - 2;
  ro = m - 2 : m - 3;
  ...
  ro = 1 : 0;
  1 : ro - 1;
  esac;
```

A `case` expression returns the value of the first expression on the right hand side of ':', such that the corresponding condition on the left hand side evaluates to `1` (*true*).

The process variables (x_i, $vote_i$, d_i for the *UniformVoting* algorithm) are updated along with the execution of the algorithm. The state of a process p at the beginning of round $r+1$ is determined from its HO set $HO(p,r)$ and the states of all the processes at the beginning of round r (the messages sent by a process in round r are determined by its state at the beginning of round r). Hence the new value of a process variable at the next state can be represented as an expression over the process variables and $h_{i,j}$. When $r = m\phi - k$ $(0 \leq k \leq n-1)$, we denote by $f(v,k)$ the new value for a process variable v.

For the *UniformVoting* algorithm, for example, when the current state represents the beginning of round $2\phi - 1$, the value of x_i at the next state is represented by $f(x_i, 1)$ shown below (see lines 5–9 of Algorithm 1):

$$f(x_i,1) \quad := \quad \begin{cases} x_i & \forall j : h_{i,j} = false \\ \min\limits_{h_{i,j}=true} \{x_j\} & \text{otherwise} \end{cases}$$

In the NuSMV input language, the expression ($f(x_0, 1)$) can be defined as follows (here $n = 3$ is assumed):

```
DEFINE
fx0_1 :=
  case
  !h00 & !h01 & !h02 : x0;
  h00 & (!h01|(x0 <= x1)) & (!h02|(x0 <= x2)): x0;
  h01 & (!h00|(x1 <= x0)) & (!h02|(x1 <= x2)): x1;
  1 : x2;
  esac;
```

For any process variable v, its value in the next state is $f(v, ro)$, since round $m\phi - ro$ is the round that corresponds to the current state. That is, the transition of the value of v is specified as $v' = f(v, ro)$. The following code fragment, which specifies the next value of x_0 in the *UniformVoting* algorithm, shows how one can describe this in the NuSMV input language.

```
next(x0) :=
  case
  ro = 1 : fx0_1;
  1 : fx0_0;
  esac;
```

4.3 Representing Conditions on HO Sets

Algorithms based on the HO model are correct when a certain condition holds on the HO sets. There are two ways to represent such a condition in our verification method. The first approach is to explicitly incorporate it into the state transition system, by imposing some constraints on the behavior of the HO sets.

In the second approach, the condition on HO sets is not modeled in the state transition system; instead, the condition is incorporated into the temporal logic formula to be verified so that only computational paths that meet that condition will be examined. In this case, since the HO set changes arbitrarily, so does the value of program variable $h_{i,j}$.

The second approach is more flexible since it allows one to verify the algorithm under various conditions without altering the state transition system. For example, if one wants to verify that p_i will eventually decide if the HO sets meet a condition c_1 in some round, then the property is expressed in CTL as AG $(c_1 \rightarrow$ AF $(d_i \neq ?))$.

However, CTL is not so expressive that this approach can always be feasible. For example, suppose that only the computation paths where the HO sets invariantly meet some condition c_2 are of interest. In this case, the property to be verified can be expressed in LTL as G $c_2 \rightarrow$ F $(d_i \neq ?)$;[4] but CTL is not capable to express this property. In such a case, we must resort to the first approach.

The condition of interest is often a conjunction of subconditions. In that case, the two approaches can be taken at the same time for different subconditions. For the *UniformVoting* algorithms, the two (sub)conditions described in Section 2.2 are of particular interest. The first condition, which states that no round is split, specifies an invariant condition. Thus the first approach is used to model this property. Specifically, we impose the following constraint on the reachable states:

$$\bigwedge_{0 \leq i < j \leq n-1} \big((h_{i,0} \wedge h_{j,0}) \vee \cdots \vee (h_{i,n-1} \wedge h_{j,n-1})\big)$$

This can be performed using the `INVAR` keyword. Below is the fragment of the NuSMV program which specifies this constraint for the case $n = 3$:

```
INVAR
     ((h00 & h10) | (h01 & h11) | (h02 & h12))
   & ((h00 & h20) | (h01 & h21) | (h02 & h22))
   & ((h10 & h20) | (h11 & h21) | (h12 & h22))
```

For the second condition to hold, on the other hand, a uniform round only needs to occur in some round after the algorithm started. This property can be expressed in CTL, as shown in Section 4.4.

4.4 Verification

Here we show the results of a verification experiment for Algorithm 1 using the model described above. If no notice is provided, these results are those obtained when the number of processes n was set to three. All measurements in this paper were performed on a Windows XP machine with a 1.66GHz Intel T2300 CPU and 1.5Gb memory.

Case 1: No Split Rounds

We first model checked the algorithm assuming only no split rounds (see Section 2.2). We started with verification of agreement. This property is expressed in CTL as follows:

$$\text{AG } \mathit{agreement} \tag{CTL 1}$$

where

$$\mathit{agreement} := \bigwedge_{0 \leq i < j \leq n-1} \big((d_i \neq ?) \wedge (d_j \neq ?) \rightarrow d_i = d_j\big)$$

[4] Operators G and F mean *globally* and *sometime in the future*, respectively.

Table 1: Time required for verification (*UniformVoting*)

	$n = 3$	$n = 4$
Agreement (CTL 1)	0.2sec	11.0sec
Termination (CTL 2)	1.9sec	14min5sec
Termination (CTL 3)	1.9sec	13min58sec
# reachable states	21,350	1.58658×10^7

The CTL formula is expressed in the NuSMV input language as follows:

```
DEFINE
  agreement :=
    ((d0 != -1) & (d1 != -1) -> (d0 = d1))
  & ((d0 != -1) & (d2 != -1) -> (d0 = d2))
  & ((d1 != -1) & (d2 != -1) -> (d1 = d2));

SPEC
  AG agreement
```

NuSMV completed the verification instantly, showing that the property holds.

Next, we checked termination. The first CTL formula we tried is:

$$\text{AF } \mathit{termination} \qquad \text{(CTL 2)}$$

where

$$\mathit{termination} := (d_0 \neq ?) \wedge \cdots \wedge (d_{n-1} \neq ?)$$

NuSMV checked this formula in a few seconds, making a verdict that it does not hold with the following counterexample.

State 1:

$x_0 = 0,\ x_1 = 0,\ x_2 = 2,\ vote_0 = ?,\ vote_1 = ?,\ vote_2 = ?,\ d_0 = ?,\ d_1 = ?,\ d_2 = ?,\ ro = 1,\ \boldsymbol{h}_0 = (true, false, true),\ \boldsymbol{h}_1 = (true, false, true),\ \boldsymbol{h}_2 = (false, false, true)$

$\downarrow$

State 2:

$x_0 = 0,\ x_1 = 0,\ x_2 = 2,\ vote_0 = ?,\ vote_1 = ?,\ vote_2 = 2,\ d_0 = ?,\ d_1 = ?,\ d_2 = ?,\ ro = 0,\ \boldsymbol{h}_0 = (true, true, false),\ \boldsymbol{h}_1 = (true, true, false),\ \boldsymbol{h}_2 = (true, false, true)$

$\downarrow$

State 1:

$\downarrow$

$\vdots$

This counterexample clearly shows an execution that alternates between State 1 and State 2; i.e., the No Split Rounds condition does not suffice to ensure termination.

Case 2: No Split Rounds and a Space Uniform Round

We changed the CTL formula so that it asserts that when a space uniform round occurs, the termination will be met eventually. The following expression represents that the current round is space uniform.

$$uniformity := \bigwedge_{1 \leq i \leq n-1} \big((h_{0,0} = h_{i,0}) \wedge \cdots \wedge (h_{0,n-1} = h_{i,n-1})\big)$$

Thus the CTL formula to be verified is:

$$\mathrm{AG}(uniformity \rightarrow \mathrm{AF}\ termination) \qquad \text{(CTL 3)}$$

NuSMV proved within two seconds that this formula holds.

Table 1 summarizes the time required for model checking the *UniformVoting* algorithm. (The time used for producing counterexamples is not included.) The number of reachable states was calculated using the `-r` option of NuSMV. When $n = 5$, verification was not be completed because of a memory shortage.

5 Extensions and Optimizations

5.1 Coordinator-Based Algorithms

Many algorithms for consensus are coordinator-based algorithms. The HO model can be naturally extended to include the notion of a coordinator. In a coordinator-based algorithm, the behavior of a process depends not only on its current state and the received messages, but also on the identity of the coordinator of that process. Here we show how to deal with coordinator-based algorithms.

To represent the coordinator, we add a new program variable per process. Let $coord_i$ be this variable for process p_i. The domain of $coord_i$ is $\{0, 1, \cdots, n-1\}$. Variable $coord_i$ can be treated in the same way as HO sets: The value of $coord_i$ could be nondeterministically changed in a fully arbitrary manner or some constraints could be imposed. Probably the most typical constraint is that the coordinator of a process never changes during a phase. In this case the coordinator can change only at the end of round $m\phi - 0$. Thus this constraint is described as:

$$(ro \neq 0) \rightarrow \big((coord'_0 = coord_0) \wedge \cdots \wedge (coord'_{n-1} = coord_{n-1})\big)$$

In the NuSMV language, this can be specified using the `TRANS` keyword, as shown below ($n = 3$).

```
TRANS
  (ro != 0) ->
    ((next(coord0) = coord0) & (next(coord1) = coord1) & (next(coord2) = coord2))
```

It is also often the case that the coordinator of a process is deterministically determined based on the current phase number. The well-known rotating coordinator strategy is such an example, in which case process $p_{(\phi-1) \bmod n}$ is the coordinator for all the processes in phase ϕ. To represent the rotating coordinator, it suffices to set the initial value of $coord_i$ to 0 and to explicitly specify the next state value as follows:

$$coord'_i = \begin{cases} (coord_i + 1) \bmod n & ro = 0 \\ coord_i & otherwise \end{cases}$$

The corresponding code fragment is shown below ($n = 3$):

Algorithm 2 The *CoordUniformVoting* algorithm [6]

1: **Initialization:**
2: $x_p \in V$, initially v_p $\{v_p$ *is the initial value of* $p\}$
3: $vote_p \in V \cup \{?\}$, initially ?

4: **Round** $r = 3\phi - 2$**:**
5: S_p^r :
6: **if** $p = Coord(p, \phi)$ **then**
7: send $\langle\, x_p \,\rangle$ to all processes

8: T_p^r :
9: **if** some message $\langle\, v \,\rangle$ is received
10: from $Coord(p, \phi)$ **then**
11: $x_p := v$

12: **Round** $r = 3\phi - 1$**:**
13: S_p^r :
14: send $\langle\, x_p \,\rangle$ to all processes

15: T_p^r :
16: **if** all the values received are equal to v **then**
17: $vote_p := v$

18: **Round** $r = 3\phi$**:**
19: S_p^r :
20: send $\langle\, vote_p \,\rangle$ to all processes

21: T_p^r :
22: **if** at least one $\langle\, v \,\rangle$ with $v \neq ?$ is received **then**
23: $x_p := v$
24: **if** all the messages received are
25: equal to $\langle\, v \,\rangle$ with $v \neq ?$ **then**
26: DECIDE(v)
27: $vote_p := ?$

```
ASSIGN
  init(coord0) := 0; init(coord1) := 0; init(coord2) := 0;

  next(coord0) :=
    case
    ro = 0 : (coord0 + 1) mod 3;
    1: coord0;
    esac;

  next(coord1) :=
    ...
```

As an example of verification of a coordinator-based algorithm, we show the results of model checking the *CoordUniformVoting* algorithm [6] (see Algorithm 2), which is a coordinator-based variant of the *UniformVoting* algorithm. In the *CoordUniformVoting* algorithm, the coordinator of each process is assumed not to vary during a phase. The coordinator of process p in phase ϕ is denoted as $Coord(p, \phi)$ in the pseudo-code.

In order to meet agreement, the *CoordUniformVoting* algorithm assumes that no round is split, as in the case of *UniformVoting*. Thus the agreement property can be checked just as described in Section 4.

Termination is intended to be satisfied if for some phase ϕ, round $3\phi - 2$ is *uniformly and well coordinated*, that is, in that round, all the processes agree on the coordinator and can hear from it. Let *uwc* be an expression that evaluates to true iff when the current round is uniformly and well coordinated; that is,

$$uwc := \Big(\bigwedge_{1 \leq i \leq n-1} (coord_0 = coord_i) \Big) \wedge \Big(\bigwedge_{0 \leq i \leq n-1} \big((coord_0 = i) \rightarrow (h_{0,i} \wedge \cdots \wedge h_{n-1,i}) \big) \Big)$$

Using this expression, the following CTL formula can be obtained which specifies the liveness property of interest.

$$\mathrm{AG}\big(((ro = 2) \wedge uwc) \rightarrow \mathrm{AF}\ termination \big) \qquad \text{(CTL 4)}$$

Note that the term $(ro = 2) \wedge uwc$ evaluates to true at a state s iff s corresponds to round $3\phi - 2$ for some phase ϕ and that round is uniformly and well coordinated.

Table 2: Time required for verification (*CoordUniformVoting* and *CoordUniformVoting*^{rc})

	CoordUniformVoting		*CoordUniformVoting*^{rc}	
	$n = 3$	$n = 4$	$n = 3$	$n = 4$
Agreement (CTL 1)	0.2sec	14.0sec	0.2sec	16.0sec
Termination (CTL 4)	0.8sec	17min5sec	0.8sec	16min39sec
# reachable states	2.46645×10^6	5.21465×10^{10}	274,050	8.14789×10^8

We were able to model check this algorithm and its variant that uses the rotating coordinator strategy (denoted by *CoordUniformVoting*^{rc}) with up to $n = 4$. It should be noted that in the latter algorithm, all the processes always agree on the same coordinator; thus the first conjunct of the *uwc* formula is always true. The running time and the size of the state spaces are shown in Table 2.

An interesting finding is that although the number of reachable states for *CoordUniform Voting*^{rc} is much smaller than *CoordUniformVoting*, the time needed for verification was almost the same for these two algorithms. The reason for this is that the size of Binary Decision Diagrams (BDDs) arising in the process of model checking was similar in both cases. In symbolic model checking, state spaces and transition relations are symbolically represented by BDDs, and the execution time critically depends on the size of these BDDs. Although a small state space can often be represented by a small BDD, this is not necessarily always the case.

5.2 Expressing Complex Conditions on HO Sets

As another example of a consensus algorithm, let us consider the *OneThirdRule* algorithm [6] (see Algorithm 3). A notable feature of this simple algorithm is that it can solve consensus in a single round in favorable circumstances where enough processes propose the same value. A similar structure is shared by the algorithms proposed in [2] and in [21], and *Fast Paxos* [16].

For this algorithm, the corresponding finite state transition system can be constructed just the way described in Section 4. Indeed, the resulting state transition system is even simpler in some sense. Since each phase of this algorithm consists of one single round, the program variable *ro* is no longer necessary. In addition, the `INVAR` part, which was used to represent the absence of split rounds for the *UniformVoting* algorithm, can be omitted

Algorithm 3 The *OneThirdRule* algorithm [6]

Initialization:
  $x_p \in V$, initially v_p { v_p *is the initial value of* p }

Round r**:**
  S_p^r :
    send $\langle\, x_p \,\rangle$ to all processes

  T_p^r :
    if $|HO(p,r)| > 2n/3$ **then**
      if the values received, except at most $[\frac{n-1}{3}]$, are equal to $\overline{x}$ **then**
        $x_p := \overline{x}$
      else
        $x_p :=$ smallest x received
      if more than $2n/3$ values received are equal to $\overline{x}$ **then**
        DECIDE($\overline{x}$)

Table 3: Time required for verification (*OneThirdRule*)

	$n=4$	$n=5$	$n=6$	$n=7$
Agreement (CTL 1)	0.1sec	0.5sec	7.7sec	5min41sec
# reachable states	4.27295×10^7	1.50324×10^{11}	3.64653×10^{15}	5.66894×10^{20}
Termination (CTL 5)	0.4sec	10.5sec	6min15sec	NA
Termination (CTL 6)	0.2sec	0.7sec	41sec	NA
# reachable states	6.39631×10^7	1.91092×10^{11}	5.83709×10^{16}	NA

too, since the *OneThirdRule* algorithm does not require any conditions to be met in order to ensure the agreement condition.

To verify termination, on the other hand, the finite state transition system needs to be extended to represent the required condition. The *OneThirdRule* algorithm terminates when the following condition holds:

$$\begin{array}{l} \exists r_0 > 0, \exists \Pi_0 \subseteq \Pi \text{ s.t. } |\Pi_0| > 2n/3, \forall p_i \in \Pi : \\ (HO(p_i, r_0) = \Pi_0) \wedge (\exists r_{p_i} > r_0 : |HO(p, r_{p_i})| > 2n/3) \end{array} \tag{1}$$

To verify that this condition is sufficient for the algorithm to terminate, one has to limit the scope of verification to the computation paths where the rounds r_0 and $r_{p_i} (p_i \in \Pi)$ occur. This can be done by introducing $n+1$ boolean program variables. With these variables, even if two states correspond to the same configuration of process states, it is possible to distinguish them depending on whether they already experienced the rounds r_0 and/or r_{p_i}.

Let a and $b_0, \cdots, b_{n-1}$ be these new variables. They are initially $false$. Variables a and b_i are used to record that the rounds r_0 and r_{p_i} have occurred, respectively. The transitions of the values of these variables are represented as follows:

$$a' = \begin{cases} true & \neg a \wedge \\ & \displaystyle\bigvee_{\Pi_0 \subseteq \Pi : |\Pi_0| > \frac{2}{3}n} \Big(\bigwedge_{p_i \in \Pi_0} (h_{0,i} \wedge \cdots \wedge h_{n-1,i}) \\ & \qquad \wedge \displaystyle\bigwedge_{p_i \in \Pi - \Pi_0} (\neg h_{0,i} \wedge \cdots \wedge \neg h_{n-1,i}) \Big) \\ a & \text{otherwise} \end{cases}$$

$$b_i' = \begin{cases} true & a \wedge \displaystyle\bigvee_{\Pi_0 \subseteq \Pi : |\Pi_0| > \frac{2}{3}n} \bigwedge_{p_j \in \Pi_0} h_{i,j} \\ b_i & \text{otherwise} \end{cases}$$

Using these variables, the following CTL formula can be obtained which states that all the processes will eventually decide provided that condition (1) holds:

$$\mathrm{AG}\big((b_0 \wedge \cdots \wedge b_{n-1}) \rightarrow \mathrm{AF}\ termination\big) \tag{CTL 5}$$

This benefit cannot come without cost; adding such auxiliary variables enlarges the state space. For example, when $n = 4$, the number of the reachable states grows from 4.27295×10^7 to 6.39631×10^7. Table 3 shows the relationships between n and the time and space needed for model checking.

5.3 Verifying Termination by Checking Reachability

Here we introduce an optimization technique for speeding up the verification of the termination property. As can be seen in the performance results shown so far, checking termination usually incurs much more execution time than checking agreement.

This is due to the difference in the CTL formulae used to represent the two properties. The agreement property is specified by AG *agreement*. As stated in Section 3, if a CTL formula is of the form AG g where g is a state formula that contains no temporal operator, then it can be verified simply by reachability analysis, which is the very first step performed in the process of model checking in NuSMV.

The idea of the proposed optimization is to represent the termination property as a CTL formula of this form. This optimization can be made when one wants to verify that a consensus algorithm terminates by the end of the round where some specific condition holds on HO sets.

For example, in [6], it is claimed that in the *OneThird-Rule* algorithm a process p_i will make a decision at the latest by the end of round r_{p_i} (see formula (1) in Section 5.2). The program variable b_i evaluates to true iff the current state corresponds to the end of round r_{p_i} or later (see again Section 5.2). Thus, instead of $\mathrm{AG}\big((b_0 \wedge \cdots \wedge b_{n-1}) \rightarrow \mathrm{AF}\ \mathit{termination}\big)$, the termination property can also be verified by checking the following CTL formula.

$$\mathrm{AG}\big((b_0 \wedge \cdots \wedge b_{n-1}) \rightarrow \mathit{termination}\big) \qquad \text{(CTL 6)}$$

We were able to check that this CTL formula holds when n is up to six, with much less execution time as shown in Table 3.

5.4 Finite Representation of Unbounded Timestamps

For all the consensus algorithms discussed so far, process variables have finite domains. On the contrary, some consensus algorithms use timestamps to record the phase number when some event happened, such as an update of an estimate of the decision value (e.g., [5, 15]). In asynchronous systems there is no bound on the phase number; thus the domain of these timestamps is a set of non-negative integers $\mathbb{N}$. In this case, clearly, possible process states are infinite.

As an illustrative example, let us take the *LastVoting* algorithm (Algorithm 4) [6]. This algorithm follows the basic line of the *Paxos* algorithm [15]. In the *LastVoting* algorithm, the agreement condition is never violated no matter how bad the HO sets are. Each process p_i uses exactly one timestamp ts_i. The timestamp is updated to the current phase number when a process receives an estimate of the decision value from the coordinator in round $4\phi - 2$ (see lines 20–21). The timestamp value is used in round $4\phi - 3$ by the coordinator to select the most recently updated estimate value (lines 11–13). It is also used for a process to decide whether to reply an ack to the coordinator in round $4\phi - 1$ (lines 24–25); the process sends an ack if its timestamp represents the current phase. Here we address the problem of modeling the behavior of such an algorithm as a finite state transition system.

Typically the ways of using a timestamp in consensus algorithms are limited only to: (i) setting it to the current phase number, (ii) arithmetically comparing it with another timestamp, and (iii) checking if it equals the current phase number. The behavior of these algorithms therefore depends on, rather than their actual values, (a) the relative order of pairs of timestamps and (b) whether the values equal the current phase number or not. Also, the timestamp values never exceed the current phase number, since they are initially set to zero. These observations lead us to the following simple representation.

Algorithm 4 The *LastVoting* algorithm [6]

```
1: Initialization:
2:     x_p ∈ V, initially v_p   {v_p is the initial value of p}
3:     vote_p ∈ V ∪ {?}, initially ?
4:     commit_p a Boolean, initially false
5:     ready_p a Boolean, initially false
6:     ts_p ∈ ℕ, initially 0

7: Round r = 4φ − 3:
8:     S_p^r :
9:         send ⟨x_p , ts_p⟩ to Coord(p, φ)

10:    T_p^r :
11:        if p = Coord(p, φ) and
           number of ⟨ν , θ⟩ received > n/2 then
12:            let θ̄ be the largest θ from ⟨ν , θ⟩ received
13:            vote_p := one ν such that ⟨ν , θ̄⟩ is received
14:            commit_p := true

15: Round r = 4φ − 2:
16:    S_p^r :
17:        if p = Coord(p, φ) and commit_p then
18:            send ⟨vote_p⟩ to all processes

19:    T_p^r :
20:        if received ⟨v⟩ from Coord(p, φ) then
21:            x_p := v ; ts_p := φ

22: Round r = 4φ − 1:
23:    S_p^r :
24:        if ts_p = φ then
25:            send ⟨ack⟩ to Coord(p, φ)

26:    T_p^r :
27:        if p = Coord(p, φ) and
           number of ⟨ack⟩ received > n/2 then
28:            ready_p := true

29: Round r = 4φ:
30:    S_p^r :
31:        if p = Coord(p, φ) and ready_p then
32:            send ⟨vote_p⟩ to all processes

33:    T_p^r :
34:        if received ⟨v⟩ from Coord(p, φ) then
35:            DECIDE(v)
36:        if p = Coord(p, φ) then
37:            ready_p := false
38:            commit_p := false
```

Let $ts_i (0 \leq i \leq N-1)$ denote a timestamp used by the algorithm under consideration. N is the total number of the timestamp variables. For the *LastVoting* algorithm, $N = n$ because each process has a single timestamp. We represent the values of these timestamps using N program variables $ats_0, ats_1, \cdots, ats_{N-1}$ such that $ats_i \in \{0, 1, \cdots, N\}$ as follows. If ts_i is not equal to the current phase number and is the jth smallest value in $\bigcup_{0 \leq i \leq N-1}\{ts_i\}$, then ats_i is set to $j-1$. If ts_i represents the current phase, on the other hand, then ats_i is set to N.

When $N = 3$, for example, $(ts_0, ts_1, ts_2) = (10, 100, 25)$ is represented as $(ats_0, ats_1, ats_2) = (0, 3, 1)$ if 100 is the current phase number; otherwise as $(ats_0, ats_1, ats_2) = (0, 2, 1)$. Similarly, $(ts_0, ts_1, ts_2) = (10, 10, 100)$ is represented as $(ats_0, ats_1, ats_2) = (0, 0, 3)$ if the current phase is phase 100; otherwise as $(ats_0, ats_1, ats_2) = (0, 0, 1)$.

Clearly, if $(ats_0, \cdots, ats_{N-1})$ corresponds to $(ts_0, \cdots, ts_{N-1})$, then (a) for any i, j, the relative order between ats_i and ats_j coincides with that between ts_i and ts_j, and (b) for any i, the timestamp ts_i is equal to the current phase number iff $ats_i = N$.

Now we show how the transition of ats_i can be specified in NuSMV. We assume that the algorithm under consideration does no update timestamp values in the last round of any phase, which is the case for the *LastVoting* algorithm. This property makes the representation of the transition simple; it can easily be generalized, however, to the case where the property does not necessarily hold.

In the initial states, every ats_i is set to zero. Since ts_i is initially zero, it is clear that the values of ats_i correctly represent ts_i in the initial states.

The transition of ats_i is specified by the conjunction of four constraints. Two of these constraints involve symbol $cp_i (0 \leq i \leq N-1)$, which represents the expression over program variables that evaluates to true iff the value of ts_i in the next state (that is, ts_i') is equal to the phase number in the next state. The expression for cp_i will be derived from the algorithm later. Here we assume that the expression is available.

Now suppose that in the current state the values of ats_i correctly represent ts_i as de-

scribed above. Then, the following two constraints guarantee that (a) for any i, j, the order between ats'_i and ats'_j matches that between ts'_i and ts'_j and (b) for any i, $ats'_i = N$ iff ts'_i is the current phase number (in the next state):

1. For any $i, j (i \neq j)$ such that $cp_i = cp_j = false$, the relative order between ats_i and ats_j is maintained in the next state; that is,

$$\bigwedge_{i,j:0\leq i<j\leq N-1} \big((\neg cp_i \wedge \neg cp_j) \rightarrow \big((ats_i = ats_j \rightarrow ats'_i = ats'_j) \\ \wedge (ats_i < ats_j \rightarrow ats'_i < ats'_j) \wedge (ats_i > ats_j \rightarrow ats'_i > ats'_j)\big)\big)$$

2. For any i, $ats'_i = N$ iff $cp_i = true$; that is,

$$\bigwedge_{0\leq i\leq N-1} \big(cp_i \leftrightarrow (ats'_i = N)\big).$$

Given that the ordering of ats_i coincides with the ordering of ts_i, the remaining two constraints shown below ensure that when $ats_i \neq N$, $ats_i = j - 1$ holds iff ts_i is the jth smallest value in $\bigcup_{0\leq i\leq N-1}\{ts_i\}$:

3. $ats_i = 0$ for some i, unless all ats_i are N; that is,

$$(ats_0 \neq N \vee \cdots \vee ats_{N-1} \neq N) \rightarrow (ats_0 = 0 \vee \cdots \vee ats_{N-1} = 0)$$

4. There is no "gap" between $ats_i (\neq N)$ and $ats_k (\neq N)$ that are consecutive in value. Formally,

$$\bigwedge_{i,j:0\leq i,j\leq N-1} \Big((ats_i < ats_j \wedge ats_j \neq N) \rightarrow \bigvee_{0\leq k\leq N-1, k\neq i} (ats_i + 1 = ats_k)\Big)$$

Since the values of ats_i correctly represent those of ts_i in the initial states, these four constraints (1) to (4) inductively guarantee the correct correspondence between ats_i and ts_i in every reachable state. Figure 1 shows these four constraints expressed in the tool language when $N = 3$.

The expression cp_i can be derived from the consensus algorithm. For the *LastVoting* algorithm, it is obtained as follows (see lines 15–21):

$$cp_i := \big((ro \neq 0) \wedge (ats_i = n)\big) \\ \vee\Big((ro = 2) \wedge \bigvee_{0\leq j\leq n-1} \big(h_{i,j} \wedge (coord_i = j) \wedge (coord_j = j) \wedge commit_j\big)\Big)$$

In words, cp_i evaluates to true iff ts_i has already been updated to the current phase number (remember $N = n$) or is updated in the current round $4\phi - 2$ as a result of the reception of a message from its coordinator. The two disjuncts of the right-hand side of the above formula respectively represent these two conditions.

When $n = 3$ and $n = 4$, we were able to prove that the agreement property of the *LastVoting* algorithm holds no matter how the HO sets are. Table 4 shows the time needed for model checking and the size of the state spaces.

The *LastVoting* algorithm is supposed to terminate at the end of a phase $\phi_0 > 0$ such that :

$$\exists c_0 \in \Pi, \forall p \in \Pi, \forall k \in \{0, 1, 2, 3\} : \\ (|HO(c_0, 4\phi - 3)| > n/2) \wedge (|HO(c_0, 4\phi - 1)| > n/2) \\ \wedge (c_0 = Coord(p, \phi_0)) \wedge (c_0 \in HO(p, \phi_0))$$

```
TRANS
   ((!cp0 & !cp1) -> ((ats0 = ats1 -> next(ats0) = next(ats1))
     & (ats0 < ats1 -> next(ats0) < next(ats1)) & (ats0 > ats1 -> next(ats0) > next(ats1))))
  &((!cp0 & !cp2) -> ((ats0 = ats2 -> next(ats0) = next(ats2))
     & (ats0 < ats2 -> next(ats0) < next(ats2)) & (ats0 > ats2 -> next(ats0) > next(ats2))))
  &((!cp1 & !cp2) -> ((ats1 = ats2 -> next(ats1) = next(ats2))
     & (ats1 < ats2 -> next(ats1) < next(ats2)) & (ats1 > ats2 -> next(ats1) > next(ats2))))

TRANS
  (cp0 = (next(ats0) = 3)) & (cp1 = (next(ats1) = 3)) & (cp2 = (next(ats2) = 3))

INVAR
  ((ats0 != 3) | (ats1 != 3) | (ats2 != 3)) -> ((ats0 = 0) | (ats1 = 0) | (ats2 = 0))

INVAR
   (((ats0 < ats1) & (ats1 != 3)) -> ((ats0 + 1 = ats1) | (ats0 + 1 = ats2)))
  &(((ats0 < ats2) & (ats2 != 3)) -> ((ats0 + 1 = ats1) | (ats0 + 1 = ats2)))
  &(((ats1 < ats0) & (ats0 != 3)) -> ((ats1 + 1 = ats0) | (ats1 + 1 = ats2)))
  &(((ats1 < ats2) & (ats2 != 3)) -> ((ats1 + 1 = ats0) | (ats1 + 1 = ats2)))
  &(((ats2 < ats0) & (ats0 != 3)) -> ((ats2 + 1 = ats0) | (ats2 + 1 = ats1)))
  &(((ats2 < ats1) & (ats1 != 3)) -> ((ats2 + 1 = ats0) | (ats2 + 1 = ats1)))
```

Figure 1: Constraints specifying timestamps ts_i

Table 4: Time required for verification (*LastVoting* and $LastVoting^{rc}$)

	LastVoting		$LastVoting^{rc}$	
	$n = 3$	$n = 4$	$n = 3$	$n = 4$
Agreement (CTL 1)	2.3sec	2min36sec	2.9sec	3min41sec
# reachable states	1.68311×10^9	1.73436×10^{14}	2.37487×10^8	3.86427×10^{12}
Termination (CTL 7)	4min5sec	NA	3min13sec	NA
Termination (CTL 8)	4.5sec	3min5sec	4.4sec	4min6sec
# reachable states	1.86288×10^9	1.75119×10^{14}	2.79977×10^8	4.16511×10^{12}

We successfully verified for the case $n \leq 4$ that the termination property holds if such a phase ϕ_0 occurs, using the techniques described in Sections 5.2 and 5.3. More specifically, we introduced n auxiliary variables to define an expression *good_phase* that evaluates to true iff phase ϕ_0 has occurred. This allows us to assert termination by either of the following two CTL formulae:

$$\mathrm{AG}(\mathit{good\_phase} \rightarrow \mathrm{AF}\ \mathit{termination}) \quad \text{(CTL 7)}$$

$$\mathrm{AG}(\mathit{good\_phase} \rightarrow \mathit{termination}) \quad \text{(CTL 8)}$$

Table 4 shows the time needed to verify these two formulae.

We also model checked the *LastVoting* algorithm with the rotating coordinator strategy (denoted by $LastVoting^{rc}$). The results are shown in Table 4.

Other Examples

Here we show the results of model checking two consensus algorithms (Algorithm 5 and Algorithm 6) proposed in [7] and [9]. These algorithms and the *LastVoting* algorithm share some characteristics as follows. First, both algorithms never violate the agreement condition no matter how bad the HO sets are. Second, all processes can decide by the end of a "good" phase that satisfies a certain condition. Thus CTL 7 and CTL 8 can be used to assert the termination property for these two algorithms, if the definition of *good_phase* is replaced with a new one corresponding to the algorithm to be verified.

Algorithm 5 Algorithm *Pa* [7]

```
 1: Initialization:
 2:     x_p ∈ V, initially v_p   {v_p is the initial value of p}
 3:     vote_p ∈ V ∪ {?}, initially ?
 4:     voteToSend_p a Boolean, initially false
 5:     ts_p ∈ ℕ, initially 0

 6: Round r = 3φ − 2:
 7:     S_p^r :
 8:         send ⟨x_p , ts_p⟩ to Coord(p, φ)

 9:     T_p^r :
10:         if p = Coord(p, φ) and
             number of ⟨ν , θ⟩ received > n/2 then
11:             let θ̄ be the largest θ from ⟨ν , θ⟩ received
12:             vote_p := one ν such that ⟨ν , θ̄⟩ is received
13:             voteToSend_p := true

14: Round r = 3φ − 1:
15:     S_p^r :
16:         if p = Coord(p, φ) and voteToSend_p then
17:             send ⟨vote_p⟩ to all processes

18:     T_p^r :
19:         if received ⟨v⟩ from Coord(p, φ) then
20:             x_p := v ; ts_p := φ

21: Round r = 3φ:
22:     S_p^r :
23:         if ts_p = φ then
24:             send ⟨ack, x_p⟩ to all processes

25:     T_p^r :
26:         if ∃v, number of received ⟨ack, v⟩ > n/2 then
27:             DECIDE(v)
28:         voteToSend_p := false
```

Table 5: Time required for verification (*Pa*)

	$n = 3$	$n = 4$
Agreement (CTL 1)	0.9sec	1min39sec
# reachable states	6.44807×10^8	1.28554×10^{14}
Termination (CTL 7)	9.2sec	61min46sec
Termination (CTL 8)	1.5sec	1min57sec
# reachable states	1.25521×10^9	1.29868×10^{14}

Algorithm 5 (which is referred to as Algorithm *Pa* in [7]) is a direct adoption of the *Paxos* algorithm (augmented with some optimizations) to the HO model. As the *LastVoting* algorithm, this algorithm uses a total of n timestamps (i.e., $N = n$). All processes decide by the end of a phase $\phi_0 > 0$ such that:

$$\begin{aligned}&\forall k \in \{0, 1, 2\} : \\ &\quad (\forall p \in \Pi : (|HO(p, \phi_0 - k)| > n/2) \wedge (Coord(p, \phi_0) \in HO(p, \phi_0 - k))) \\ &\quad \wedge (\forall p, q \in \Pi : Coord(p, \phi_0) = Coord(q, \phi_0))\end{aligned}$$

Table 5 summarizes the time required for model checking.

Algorithm 6 is due to Dwork, Lynch, and Stockmeyer [9], which we refer to as the *DLS* algorithm. This algorithm adopts the rotating coordinator strategy; i.e., $Coord(p, \phi) = p_{(\phi-1) \bmod n}$ for any $p \in \Pi$. Each phase consists of a lock-release phase (round $4\phi - 3$) and a trying phase (rounds $4\phi - 2, 4\phi - 1, 4\phi$). In the trying phase a process may *lock* a value that has been proposed by some process. A phase number is associated with every lock as a timestamp (see lines 37–39). Since n processes can propose at most n distinct values, each process needs at most n timestamp variables. Hence $N = n^2$.

The following two modifications were made to the original algorithm: (1) In Algorithm 6, the lock-release phase is performed before the trying phase, while the trying phase comes first in the original algorithm. (2) In Algorithm 6, processes send acks to all processes in round 4ϕ (see lines 44–45), while in the original algorithm, processes send acks only to the coordinator.

In [9] it was proven that the algorithm terminates when a majority of processes are correct and the *Global Stabilization Time* (GST) is reached. GST is defined as a time such that all messages sent from correct processes at that time or afterwards are correctly

Algorithm 6 The *DLS* algorithm [9] $(\forall p \in \Pi : Coord(p, \phi) = p_{(\phi-1) \bmod n})$

Initialization:
  $PROPER_p \in 2^V$, initially $\{v_p\}$
    $\{v_p$ *is the initial value of* $p\}$
  $LOCKS_p \in 2^{V \times \mathbb{N}}$, initially $\emptyset$
  $vote_p \in V \cup \{?\}$, initially ?
  $lock_p$, a Boolean, initially `false`

Round $r = 4\phi - 3$**:**
  S_p^r :
    send $\langle\langle PROPER_p \rangle\rangle$ to all processes
    send $\langle\, LOCKS_p \,\rangle$ to all processes

  T_p^r :
    for all $\langle\langle P \rangle\rangle$ received **do**
      $PROPER_p := PROPER_p \cup P$
    for all $\langle L \rangle$ received **do**
      for all $(v, l) \in LOCKS_p$ such that
          $\exists (w, h) \in L$ with $v \neq w$, $l \leq h$ **do**
        $LOCKS_p := LOCKS_p - \{(v, l)\}$

Round $r = 4\phi - 2$**:**
  S_p^r :
    send $\langle\langle PROPER_p \rangle\rangle$ to all processes
    if $LOCKS_p = \emptyset$ **then**
      send $\langle PROPER_p \rangle$
    else
      if $|LOCKS_p| = 1$ **then**
        send $\langle \{v\} \rangle$ such that $LOCKS_p = \{(v, ts)\}$

  T_p^r :
    for all $\langle\langle P \rangle\rangle$ received **do**
      $PROPER_p := PROPER_p \cup P$
    if $p = Coord(p, \phi)$ **and**
      $\exists v$, more than $n/2$ $\langle L \rangle$ received contain v **then**
      $vote_p := v$

Round $r = 4\phi - 1$**:**
  S_p^r :
    send $\langle\langle PROPER_p \rangle\rangle$ to all processes
    if $p = Coord(p, \phi)$ and $vote_p \neq ?$ **then**
      send $\langle\, vote_p \,\rangle$ to all processes

  T_p^r :
    for all $\langle\langle P \rangle\rangle$ received **do**
      $PROPER_p := PROPER_p \cup P$
    if $\langle\, v \,\rangle$ received from $Coord(p, \phi)$ **then**
      $LOCKS_p := LOCKS_p - \{(v, *)\}$
      $LOCKS_p := LOCKS_p \cup \{(v, \phi)\}$
      $vote_p := v$; $lock_p :=$ `true`

Round $r = 4\phi$**:**
  S_p^r :
    send $\langle\langle PROPER_p \rangle\rangle$ to all processes
    if $lock_p =$ `true` **then**
      send $\langle\, ack, vote_p \,\rangle$ to all processes

  T_p^r :
    for all $\langle\langle P \rangle\rangle$ received **do**
      $PROPER_p := PROPER_p \cup P$
    if $\exists v$, number of received $\langle ack, v \rangle > n/2$ **then**
      DECIDE(v)
    $vote_p :=?$; $lock_p =$ `false`

delivered. In [6] it is shown that this termination requirement can be drastically weakened. Specifically, Algorithm 6 terminates by the end of a phase ϕ_0 such that:

$$\exists \Pi_0 \subseteq \Pi \text{ s.t. } |\Pi_0| > n/2 : \\ (\forall k \in \{0, 1, 2, 3\}, \forall p \in \Pi : HO(p, \phi_0 - k) = \Pi_0) \wedge (Coord(p, \phi_0) \in \Pi_0)$$

When $n = 3$, we were able to verify that agreement and termination hold. Table 6 shows the time needed for verification.

6 Conclusions

In this paper, we presented a model checking approach for verifying HO model-based consensus algorithms. First we showed how one can model the behavior of the *Uniform-Voting* algorithm as a finite state transition system. Also we demonstrated that CTL formulae can be used to specify the agreement and termination properties and that with the NuSMV model checker these properties can be verified with practical time. We also presented the results of model checking other consensus algorithms, in conjunction with the techniques for applying our approach to these algorithms.

Our approach is different from previous attempts to apply formal verification methods to asynchronous consensus algorithms in at least one of the following two aspects: (1) Our approach relies only on standard model checking techniques and thus the verification is fully automatic, and (2) our approach is complete in the sense that it verifies the behavior

Table 6: Time required for verification (*DLS*)

	$n = 3$
Agreement (CTL 1)	24min41sec
# reachable states	3.6656×10^8
Termination (CTL 7)	NA
Termination (CTL 8)	29min12sec
# reachable states	4.86804×10^8

of consensus algorithms in every possible state. This is, to our knowledge, the first time standard model checking allows one to completely verify asynchronous consensus algorithms.

A notable technique that we devised is the finite representation of unbounded timestamps. This allows us to use existing powerful technologies for finite state space exploration to verify algorithms having infinite state spaces. In fairness, it should be noted that there exist model checking methods that can deal with infinite integer variables in a general form by means of, for example, Presburger arithmetic [3, 4]. Although these methods can often be applied only to small systems, a qualitative comparison should be made in the future.

Acknowledgments

We would like to thank Abdul Haseeb for discussions on this work.

References

[1] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proc. Second ACM Symp. on Principles of Distributed Computing (PODC-2)*, pages 27–30, 1983.

[2] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *Proc. 6th International Conference on Parallel Computing Technologies (PaCT '01)*, volume 2127 of *Lecture Notes in Computer Science*, pages 42–50, London, UK, 2001. Springer-Verlag.

[3] T. Bultan, R. Gerber, and C. League. Composite model-checking: Verification with type-specific symbolic representations. *ACM Trans. Softw. Eng. Methodol.*, 9(1):3–50, 2000.

[4] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Trans. on Progr. Languages and Syst.*, 21(4):747–789, 1999.

[5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, 1996.

[6] B. Charron-Bost and A. Schiper. The Heard-Of model: Unifying all benign failures. *Technical Report, LSR-REPORT-2006-004, EPFL*, July 2006.

[7] B. Charron-Bost and A. Schiper. Improving Fast Paxos: Being optimistic with no overhead. In *Proc. of 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Riverside, CA, USA, Dec. 2006 (to appear).

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Proc. of 14th Conf. on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, Copenhagen, Denmark, July 2002. Springer.

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

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

[11] R. Fuzzati, M. Merro, and U. Nestmann. Distributed consensus, revisited. *Technical Report, LSR-REPORT-2006-003, EPFL*, Sept. 2006.

[12] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[13] J. Helin and P. Kellomäki. Invariants come from templates. In *Proc. 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '05)*, pages 90–97, Lisbon, Portugal, 2005.

[14] M. Z. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *Proc. of 13th Conf. on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 194–206, Paris, France, July 2001. Springer.

[15] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, 1998.

[16] L. Lamport. Fast Paxos. *Techical Report, MSR-TR-2005-112, Microsoft*, 2005.

[17] L. Lamport. Personal Communication, 2006.

[18] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[19] T. Ne Win, M. D. Ernst, S. J. Garland, D. Kırlı, and N. Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004.

[20] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *Proc. 14th Conf. on Concurrency Theory (CONCUR'03)*, volume 2761 of *Lecture Notes in Computer Science*, pages 393–407, Marseille, France, Sept. 2003. Springer.

[21] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conference (EDCC-4)*, volume 2485 of *Lecture Notes in Computer Science*, pages 44–61, Toulouse, France, Oct. 2002. Springer.

[22] A. Pogosyants, R. Segala, and N. A. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: A case study. *Distributed Computing*, 13(3):155–186, 2000.

[23] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3):733–749, July 1986.