View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

26th IEEE International Symposium on Reliable Distributed Systems

Model Checking of Consensus Algorithms

Tatsuhiro Tsuchiya *
Osaka University
1-5 Yamadaoka, Suita, 565-0871 Japan
tatsuhiro@ieee.org

André Schiper †
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
andre.schiper@epfl.ch

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, a new computation model for distributed computing. By making use of the high abstraction level provided by this computation model and by devising a finite representation of unbounded timestamps, we develop a methodology for verifying consensus algorithms in every possible state by model checking.

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 [8, 9, 19]. 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 *consensus*, a fundamental problem in fault-tolerant distributed computing. Consensus not only captures the difficulty related to fault-tolerance in distributed systems — it is also a basic building block that handles failures for solving other agreement problems such as atomic broadcast or group membership [6, 17, 32].

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 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 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 [16, 20, 24], therefore, model checking was not used as a stand alone method, but in conjunction with other mathematical proof techniques.

Our approach presented in this paper is different from the previous work in that it 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*

*This work was done when the first author was visiting EPFL with support from Scientist Exchange Program between JSPS and SNSF.

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

1060-9857/07 $25.00 © 2007 IEEE
DOI 10.1109/SRDS.2007.20

IEEE computer society

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 when the algorithm uses timestamps, because the number of rounds is unbounded. To cope with this problem, we devise a technique for representing infinite combinations of timestamp values as finite representatives. We also develop several optimization techniques. These techniques enable one to apply standard model checking to a class of nontrivial consensus algorithms including *Paxos* [21].

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. Section 2 describes the HO model and the consensus problem. Section 3 briefly explains the concept of model checking. Section 4 shows how one can model check HO model-based consensus algorithms by taking a particular algorithm as an example. Section 5 introduces two optimization techniques. Section 6 describes a technique for representing, as a finite state space, the behavior of a consensus algorithm that uses unbounded timestamps. Section 7 summarizes related work. Section 8 concludes the paper and points out future work.

2. The HO Model and the Consensus Problem

2.1. The HO Model

We consider a distributed system consisting of n processes. Let $\Pi = \{p_1, p_2, \cdots, p_n\}$ be the set of the processes. We assume a communication-closed round computation model, called the *Heard-Of (HO) Model* [9]. The HO model generalizes the asynchronous round model in [13] with some features of [15] and [31]. The two notable features of the HO 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. Each round consists of three parts: *send*, *receive*, and *state transition*. Every process sends messages to all or a subset of processes, then receives the messages sent to it, and finally makes a state transition based on the current state and the messages it received. We refer to the collection of the states of the n processes as a *configuration*.

We denote by $HO(p_i, r)$ ($\subseteq \Pi$) the set of processes from which p_i receives a message in round r: $HO(p_i, r)$ is the "heard of" set of p_i in round r. A transmission fault refers to the situation where $p_j \notin HO(p_i, r)$ while p_j sent (or was supposed to send) a message to p_i in round 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:

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. Discussion of the reason for this specification can be found in [8, 9].

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

Algorithm 1 The *OneThirdRule* algorithm [9]

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 $\lceil \frac{n-1}{3} \rceil$, 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($\overline{x}$)

that is not contained in V. V is an arbitrary set of totally ordered elements. 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.

As a running example, we consider the *OneThirdRule* algorithm [9] (Algorithm 1). 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 [5] and in [29], and by *Fast Paxos* [24]. Each round r starts with the *send* part denoted by S_p^r. Each process p then receives messages from processes (implicit in Algorithm 1). 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 predicate over the collections of sets $(HO(p,r))_{p\in\Pi,r>0}$. It is well known that no deterministic consensus algorithm is possible in pure asynchronous systems prone to failures [14]. In general, therefore, consensus algorithms based on the HO model are intended to work when a certain predicate holds. The *OneThirdRule* algorithm, for example, assumes the following predicate:

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

When the transition part T_p^r is executed, the messages available guarantee that the predicate on the HO sets hold. This predicate ensures that (i) the existence of some round r_0 in which all processes hear of the same set of more than two-thirds of the processes, and (ii) for each process p_i, the existence of some round $r_{p_i}(> r_0)$ in which p_i hears of more than two-thirds of the processes. Round r_0 allows every process p_i to adopt the same value for x_{p_i} at the end of this round, while round r_{p_i} ensures that p_i decides in that round, since p_i can receive the same value from more than $2/3n$ processes. It should be noted that this predicate is only required for termination. Agreement is not violated no matter how bad the HO sets are.

It depends on the underlying system model whether a given predicate can be implemented or not. In [19], two algorithms are proposed that implement predicate (1) in systems that alternate between good (synchronous) periods and bad (asynchronous) periods.

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. 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.

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. A *computation path* is defined as an infinite sequence of states $s_0, s_1, \cdots$ 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* [25], 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 [10]. NuSMV is a reimplementation of CMU SMV [25], and is one of the latest and most successful model checkers. Performance comparisons with the SMV family and other model checkers can be found for example in [12, 22].

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*.

NuSMV supports CTL as a temporal specification logic. Here we only use two temporal operators, $\mathbf{AG}$ and $\mathbf{AF}$. The formula $\mathbf{AG}g$ holds in state s if g holds in all states along all computation paths starting from s, while the formula $\mathbf{AF}g$ holds in state s if g holds in some state along all computation paths starting from s.

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. In the next step, the set of states where the given temporal property holds is computed. This is done by recursively computing the state set satisfying each CTL subformula with a backward search from the reachable states. 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.

The time complexity of CTL model checking is $O(|f| \cdot (|S| + |R|))$ where $|f|$ is the total number of subformulas of the given CTL formula f. An optimization can be made if f is of the form of $\mathbf{AG}g$ where g contains no temporal operator. In this case the first step (that is, reachability analysis) suffices to check that formula. In NuSMV the `-AG` option enables this optimization, 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. The *OneThirdRule* algorithm is used as a running example. Figure 1 shows the NuSMV program for this algorithm when $n = 4$.

```
1  MODULE main
2  VAR
3    p1: proc(p1, p2, p3, p4); p2: proc(p1, p2, p3, p4);
4    p3: proc(p1, p2, p3, p4); p4: proc(p1, p2, p3, p4);
5
6  MODULE proc(p1, p2, p3, p4)
7  VAR
8    x: {1, 2, 3, 4}; d: {1, 2, 3, 4, 0};
9  IVAR
10   h1 : boolean; h2 : boolean; h3 : boolean; h4 : boolean;
11 ASSIGN
12   init(d) := 0;
13   next(d) :=
14     case
15       h1 & h2 & h3 & (p1.x = p2.x) & (p1.x = p3.x) : p1.x;
16       h1 & h2 & h4 & (p1.x = p2.x) & (p1.x = p4.x) : p1.x;
17       h1 & h3 & h4 & (p1.x = p3.x) & (p1.x = p4.x) : p1.x;
18       h2 & h3 & h4 & (p2.x = p3.x) & (p2.x = p4.x) : p2.x;
19       1: d;
20     esac;
21   next(x) :=
22     case (h1 & h2 & h3) | (h1 & h2 & h4)
23         |(h1 & h3 & h4) | (h2 & h3 & h4):
24       case
25         h1 & h2 & (p1.x = p2.x) : p1.x;
26         h1 & h3 & (p1.x = p3.x) : p1.x;
27         h1 & h4 & (p1.x = p4.x) : p1.x;
28         h2 & h3 & (p2.x = p3.x) : p2.x;
29         h2 & h4 & (p2.x = p4.x) : p2.x;
30         h3 & h4 & (p3.x = p4.x) : p3.x;
31         h1 & (!h2 | p1.x <= p2.x) & (!h3 | p1.x <= p3.x)
32            & (!h4 | p1.x <= p4.x) : p1.x;
33         h2 & (!h1 | p2.x <= p1.x) & (!h3 | p2.x <= p3.x)
34            & (!h4 | p2.x <= p4.x) : p2.x;
35         h3 & (!h1 | p3.x <= p1.x) & (!h2 | p3.x <= p2.x)
36            & (!h4 | p3.x <= p4.x) : p3.x;
37         h4 & (!h1 | p4.x <= p1.x) & (!h2 | p4.x <= p2.x)
38            & (!h3 | p4.x <= p3.x) : p4.x;
39         1: x;
40       esac;
41      1: x;
42     esac;
```

Figure 1. NuSMV program for the *OneThirdRule* algorithm ($n = 4$)

4.1. Program Variables

Program variables determine the state space S. Since different configurations must be distinguished in S, we need 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 $\{1, 2, \cdots, n\}$ for V. In other words, the elements of $\{1, 2, \cdots, n\}$ can be viewed as symbolic values representing any of at most n distinct values taken from V.

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

- $x_i \in \{1, 2, \cdots, n\} \quad (i = 1, 2, \cdots, n)$.
- $d_i \in \{1, 2, \cdots, n\} \cup \{?\} \quad (i = 1, 2, \cdots, n)$.

In Figure 1 these variables are declared in lines 7–8. The value ? is represented as `0` to avoid type conflicts.

Using these variables we construct the state transition system (S, I, R) as follows. A state in S represents a configuration at the beginning of a round. The set I of initial states contains all configurations that correspond to round one. A transition $(s, s') \in R$ exists iff s' represents the configuration that can be yielded by a round of algorithm execution from the configuration represented by s.

In addition to these program variables, we use NuSMV's *input variables* to represent the HO sets. Input variables are not part of the state transition system; technically they are existentially quantified out when computing transitions. An input variable can take any value of its domain and no constraints can be imposed on the value. The HO set for process p_i is represented by n boolean input variables $h_{i,1}, h_{i,1}, \cdots, h_{i,n}$ such that $h_{i,j} = true$ iff p_j belongs to the HO set for p_i in the current round. These n input variables for p_i are declared under the keyword `IVAR` in lines 9–10 in Figure 1.

4.2. Representing Algorithms

Since the initial states of the state transition system represent the configurations when an algorithm starts, process variables are initialized as specified in a given algorithm (see line 12 in Figure 1). 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 *OneThirdRule* algorithm, this applies to the variables x_i, because a process can propose any value in V.

The program variables (i.e., x_i and d_i) 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 program variable at the next state can be represented as an expression over the program variables and $h_{i,j}$. By convention, we use a primed variable to refer to the value of a variable at the next state. In Figure 1, d'_i and x'_i (i.e., the next state values of d_i and x_i) are specified by the `case` statements in lines 13– 20 and in lines 21 – 42, respectively (`x`, `d`, `next(x)`, `next(d)`, and `p`j`.x` refer to x_i, d_i, x'_i, d'_i, and x_j).

4.3. Verification

We first discuss the verification of agreement. Agreement is expressed in CTL as follows:

$$\mathbf{AG}\ agreement \qquad \text{(CTL 1)}$$

where $agreement := \bigwedge_{1\le i<j\le n} \big((d_i{\neq}?)\wedge(d_j{\neq}?) \rightarrow d_i = d_j\big)$.

We were able to check that this CTL formula holds when n is up to seven. The running time and the size of the state spaces are shown in Table 1. This and all subsequent measurements in this paper were performed on a Windows XP machine with a 1.66GHz Intel T2300 CPU and 1.5Gb memory. Time data was collected using `timeit.exe` in Windows 2003 Server Resource Kit and averaged over 10 runs.

To verify termination, the finite state transition system needs to be extended to represent the required predicate on HO sets. Since we are interested in the situation where predicate (1) holds (see Section 2.2), it is necessary 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 $b_1, \cdots, b_n$ 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:

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

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

where a' and b'_i are the values of a and b_i at the next state.

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

$$\mathbf{AG}\big((b_1\wedge\cdots\wedge b_n)\rightarrow \mathbf{AF}\ termination\big) \qquad \text{(CTL 2)}$$

where $termination := (d_1 \neq ?) \wedge \cdots \wedge (d_n \neq ?)$.

We should remark that adding such auxiliary variables enlarges the state space. For example, when $n = 6$, the number of the reachable states grows from $53,064$ to $849,408$. Table 1(a) shows the relationships between n and the time and space needed for model checking.

5. Optimizations

5.1. Optimizing Agreement Verification

The performance of agreement verification may be improved by slightly modifying the modeling proposed in Section 4. The idea behind this optimization is to split the model checking problem into two subproblems: (1) checking that no two processes decide differently in the same round, and (2) checking that no process decides a value different from the value decided in earlier rounds. If agreement is verified by solving these two problems, the n program variables d_i $(i = 1, 2, \cdots, n)$ can be omitted from the NuSMV program. Instead, we introduce a new program variable d with domain $\{1, \cdots, n\} \cup \{?\}$ to record the decision value. That is, the value of d is a value decided before the current round ($d = ?$ if no decision has been made yet).

Subproblem (1) can be solved even without using d. Let D_i be the value that process p_i decides in the current round. As stated in Section 4.2, this value can be represented by an expression over $h_{i,j}$ and program variables. No two processes decide differently in the current round iff the following formula evaluates to true:

$$agreement_1 := \bigwedge_{1\le i<j\le n} \big((D_i{\neq}?)\wedge(D_j{\neq}?) \rightarrow D_i = D_j\big)$$

Variable d is used in solving subproblem (2). Suppose that no two processes decided differently in any of the earlier rounds. Then, in the current round no process makes a decision different from the value already decided iff the following formula evaluates to true:

$$agreement_2 := \bigwedge_{1\le i\le n} \big((D_i{\neq}?)\wedge(d{\neq}?) \rightarrow D_i = d\big)$$

As a result, agreement can be expressed by the following CTL formula:

$$\mathbf{AG}(agreement_1 \wedge agreement_2) \qquad \text{(CTL 3)}$$

Table 1. Time required for verification (*OneThirdRule*)

(a) Without optimizations

	$n=4$	$n=5$	$n=6$	$n=7$
Agreement (CTL 1)	0.0sec	0.5sec	7.5sec	5min34sec
# reachable states	652	4480	53064	1.00701×10^6
Termination (CTL 2)	0.3sec	7.1sec	5min53sec	NA
# reachable states	976	5695	849408	NA

(b) With optimizations

	$n=4$	$n=5$	$n=6$	$n=7$
Agreement (CTL 3)	0.0sec	0.1sec	7.4sec	5min30sec
# reachable states	2.01851×10^7	1.07710×10^{11}	3.22102×10^{15}	4.66762×10^{20}
Termination (CTL 4)	0.2sec	2.7sec	40sec	NA
# reachable states	976	5695	849408	NA

This optimization requires a slight modification of the NuSMV programs: It requires $h_{i,j}$ to be declared as a variable, instead of an input variable, because NuSMV does not allow CTL specifications to contain input variables. Although this modification blows up the number of reachable states, it does not directly affect the performance of model checking.

As shown in Table 1(b), the effect of this optimization is not very tangible for the *OneThirdRule* algorithm. On the other hand, it worked well for the algorithm presented in Section 6. The results for this algorithm will be shown later.

5.2. Optimizing Termination Verification

Here we introduce an important optimization technique for speeding up the verification of the termination property. As can be seen in Table 1, checking termination requires much more execution time than checking agreement.

This is mainly due to the difference in the CTL formulae used to represent the two properties. The agreement property is specified by $\mathbf{AG}\ agreement$. As stated in Section 3, if a CTL formula is of the form $\mathbf{AG}\ g$ where g 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 easily be implemented 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 the *OneThird-Rule* algorithm, we claimed in Section 2.2 that a process p_i makes a decision at the latest by the end of round r_{p_i} (see predicate (1) in Section 2.2). The program variable b_i evaluates to true iff the current state corresponds to the end of round r_{p_i} or later (see Section 4.3). Thus, instead of $\mathbf{AG}\big((b_1 \wedge \cdots \wedge b_n) \rightarrow \mathbf{AF}\ termination\big)$, the termination property can also be verified by checking the following CTL formula:

$$\mathbf{AG}\big((b_1 \wedge \cdots \wedge b_n) \rightarrow termination\big) \qquad \text{(CTL 4)}$$

As shown in Table 1(b), this technique allowed us to check termination with much less execution time.

6. Model Checking Consensus Algorithms with Unbounded Timestamps

To solve consensus, existing algorithms often incorporate one or more of the following features:

- Execution in phases, each of which consists of multiple rounds.[1]
- A coordinator process used to orchestrate each phase.
- Timestamps used to record the phase number when some event happened, such as an update of an estimate of the decision value.

For example, *Paxos* [21] and the Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm [6] use all the three features.

By introducing additional program variables, the model checking approach presented in Sections 4 and 5 can be extended to incorporate the first two features. Specifically, when a phase consists of m rounds, the current round can be expressed by program variable ro with domain $\{0, 1, \cdots, m-1\}$, such that the current round is $m\phi - ro$ for some phase ϕ (≥ 1).[2] The coordinator of a process p_i is represented by program variable $coord_i$ with domain $\{1, 2, \cdots, n\}$. For more details see [33].

[1] In [6] and [21], a round is decomposed in phases. "Round" and "phase" are swapped here to use the classical terminology [13].

[2] $ro = m-1$ represents rounds $1, m+1, 2m+1, 3m+1$, etc.; $ro = m-2$ represents rounds $2, m+2, 2m+2, 3m+2$, etc.

On the other hand, timestamps are far more difficult to deal with. 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 2) [9], which follows the basic line of the *Paxos* algorithm. This algorithm uses exactly one timestamp ts_i for each process p_i.

The *LastVoting* algorithm runs in phases and each phase ϕ consists of four rounds $3\phi-3, 3\phi-2, 3\phi-1, 3\phi$. The coordinator of process p in phase ϕ is denoted as $Coord(p, \phi)$ in the pseudo-code. The selection of coordinators is done outside of the algorithm. That is, the algorithm itself does not impose any restrictions on the value of $Coord(p, \phi)$. Thus, for example, we can have multiple coordinators in the same phase.

The timestamp ts_i is updated to the current phase number when a process p_i 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. In Section 6.1 we address the problem of representing ts_i with a finite number of states.

Algorithm 2 The *LastVoting* algorithm (Algorithm *à la Paxos*) [9]

```
 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)
                    {Coord(p, φ) is the coordinator of p in phase 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
```

6.1. Finite Representation of Unbounded Timestamps

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.

Let ts_i $(1 \leq i \leq N)$ 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 $ats_1, \cdots, ats_N$, 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 $\bigcup_{1\leq i\leq N}\{ts_i\}$, 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_1, ts_2, ts_3) = (10, 100, 25) is represented as $(ats_1, ats_2, ats_3) = (0, 3, 1)$ if 100 is the current phase number; otherwise as $(ats_1, ats_2, ats_3) = (0, 2, 1)$. Similarly, $(ts_1, ts_2, ts_3) = (10, 10, 100)$ is represented as $(ats_1, ats_2, ats_3) = (0, 0, 3)$ if the current phase is phase 100; otherwise as $(ats_1, ats_2, ats_3) = (0, 0, 1)$.

Clearly, if $(ats_1, \cdots, ats_N)$ corresponds to $(ts_1, \cdots, ts_N)$, 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 $ats_i = N$.

As shown below, the transition of ats_i can be concisely specified in the form of propositional constraints. Symbolic model checking allows one to directly impose these constraints on the transition relation or the reachable states. For presentation purposes, we assume that the algorithm under consideration does not 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 even simpler; 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 $(1 \leq i \leq N)$, which represents the expression over program variables that evaluates to true iff the value of ts_i in the next state, denoted by ts_i',[3] 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 described 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,

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

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

$$\bigwedge_{1\leq i\leq N} \big(cp_i \leftrightarrow (ats_i' = N)\big).$$

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

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

$$(ats_1 \neq N \vee \cdots \vee ats_N \neq N) \\ \rightarrow (ats_1 = 0 \vee \cdots \vee ats_N = 0)$$

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

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

[3] Remember that a primed variable is used to refer to the next state value (see Section 4.2).

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 2 shows these four constraints expressed in the tool language, where $N = 3$ and `pi.ats` and `pi.cp` refer to ats_i and cp_i. The `TRANS` keyword is used to declare the constraints on the transition relation (constraints (1) and (2)), while the `INVAR` keyword is used to specify the constraints on the reachable states (constraints (3) and (4)).

6.2. Model Checking the *LastVoting* Algorithm

The expression cp_i can be derived from the consensus algorithm. For the *LastVoting* algorithm, it is obtained as follows (see lines 15–21 of Algorithm 2):

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

where ro and $coord_i$ represent the current round and p_i's coordinator, respectively (see Section 5.2 and the beginning of Section 6). 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 is never violated. Table 2 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 :

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

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 4.3 and 5.2. 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:

$$\mathbf{AG}(\mathit{good\_phase} \rightarrow \mathbf{AF}\; \mathit{termination}) \qquad \text{(CTL 5)}$$

$$\mathbf{AG}(\mathit{good\_phase} \rightarrow \mathit{termination}) \qquad \text{(CTL 6)}$$

Table 2 shows the time needed to verify these two formulae.

```
TRANS
   ((!p1.cp & !p2.cp) -> ((p1.ats = p2.ats -> next(p1.ats) = next(p2.ats)) &
     (p1.ats < p2.ats -> next(p1.ats) < next(p2.ats)) & (p1.ats > p2.ats -> next(p1.ats) > next(p2.ats))))
 & ((!p1.cp & !p3.cp) -> ((p1.ats = p3.ats -> next(p1.ats) = next(p3.ats)) &
     (p1.ats < p3.ats -> next(p1.ats) < next(p3.ats)) & (p1.ats > p3.ats -> next(p1.ats) > next(p3.ats))))
 & ((!p2.cp & !p3.cp) -> ((p2.ats = p3.ats -> next(p2.ats) = next(p3.ats)) &
     (p2.ats < p3.ats -> next(p2.ats) < next(p3.ats)) & (p2.ats > p3.ats -> next(p2.ats) > next(p3.ats))))

TRANS
  (p1.cp = (next(p1.ats) = 3)) & (p2.cp = (next(p2.ats) = 3)) & (p3.cp = (next(p3.ats) = 3))

INVAR
  ((p1.ats != 3) | (p2.ats != 3) | (p3.ats != 3)) -> ((p1.ats = 0) | (p2.ats = 0) | (p3.ats = 0))

INVAR
   (((p1.ats < p2.ats) & (p2.ats != 3)) -> ((p1.ats + 1 = p2.ats) | (p1.ats + 1 = p3.ats)))
 & (((p1.ats < p3.ats) & (p3.ats != 3)) -> ((p1.ats + 1 = p2.ats) | (p1.ats + 1 = p3.ats)))
 & (((p2.ats < p1.ats) & (p1.ats != 3)) -> ((p2.ats + 1 = p1.ats) | (p2.ats + 1 = p3.ats)))
 & (((p2.ats < p3.ats) & (p3.ats != 3)) -> ((p2.ats + 1 = p1.ats) | (p2.ats + 1 = p3.ats)))
 & (((p3.ats < p1.ats) & (p1.ats != 3)) -> ((p3.ats + 1 = p1.ats) | (p3.ats + 1 = p2.ats)))
 & (((p3.ats < p2.ats) & (p2.ats != 3)) -> ((p3.ats + 1 = p1.ats) | (p3.ats + 1 = p2.ats)))
```

Figure 2. Constraints specifying timestamps ts_i

6.3. Other Consensus Algorithms

In addition to the *LastVoting* algorithm, we model checked several consensus algorithms that use unbounded timestamps, including:

- The rotating coordinator version of the *LastVoting* algorithm. This version is different from the original *LastVoting* in that the coordinator is deterministically chosen based on the phase number; i.e., $Coord(p_i, \phi) = p_{((\phi-1) \bmod n)+1}$ for any $p_i \in \Pi$. Table 2 shows the results of model checking this algorithm (denoted by *LastVoting*^{rc}).[4]
- The algorithm of Dwork, Lynch, and Stockmeyer for benign faults [13]. In this algorithm each process has at most n timestamps, thus resulting in $N = n^2$.
- A variant of *Paxos* [7].
- Two variants of *FastPaxos* [7]. These algorithms improve the original *FastPaxos* algorithm by merging fast rounds and ordinary rounds.

For almost all of these algorithms, the proposed approach successfully handled the cases up to $n = 4$. Some of these results can be found in our technical report [33].

6.4. Comparison with Existing Model Checking Techniques

To demonstrate the benefits of using our technique, we compare it with the following two methods:

Method 1 By imposing an upper bound on the phase number, timestamps are treated as usual process variables, instead of using the technique proposed in this section.

Method 2 This method is the same as Method 1, except that *bounded model checking* [11] is used, instead of ordinary Binary Decision Diagram (BDD)-based symbolic model checking. The idea of bounded model checking is to search a counterexample of length up to a given bound. This bounded version of the model checking problem is reduced to the propositional satisfiability problem (SAT), and can thus be solved by SAT methods rather than BDDs. Since NuSMV supports bounded model checking as well as ordinary symbolic model checking, we used it in our experiment with ZChaff [26] as a SAT solver.

Figure 3 shows the results of using these methods for verifying the agreement property of the *LastVoting* algorithm. In this experiment CTL 1 was used as the specification of agreement. From the results, it can be seen that Method 1 exhibited much better performance than Method 2. Compared with our proposed method, however, it runs faster only for a few first phases. More importantly, verification based on a fixed number of phases cannot be used to ensure that the consensus algorithm is correct even for a small n. This is in contrast to our approach, which explores all possible states of the algorithm.

7. Related Work

Not much research exists regarding the application of model checking to asynchronous consensus algorithms. In [20], a shared memory-based randomized consensus al-

[4] Although the number of reachable states of *LastVoting*^{rc} was much smaller than *LastVoting*, verifying *LastVoting*^{rc} took similar or even longer time. A possible reason for this is that the size of Binary Decision Diagrams (BDDs) generated in the process of model checking was similar in both cases.

Table 2. Time required for verification (*LastVoting* and *LastVoting*^{rc})

(a) Without optimizations

	LastVoting		*LastVoting*^{rc}	
	$n = 3$	$n = 4$	$n = 3$	$n = 4$
Agreement (CTL 1)	2.1sec	2min29sec	2.7sec	3min36sec
# reachable states	3.28732×10^6	2.64643×10^9	463842	5.8964×10^7
Termination (CTL 5)	3min37sec	NA	1min18sec	NA
# reachable states	3.63844×10^6	2.67210×10^9	546831	6.35545×10^7

(b) With optimizations

	LastVoting		*LastVoting*^{rc}	
	$n = 3$	$n = 4$	$n = 3$	$n = 4$
Agreement (CTL 3)	1.9sec	1min59sec	3.5sec	6min10sec
# reachable states	6.41646×10^8	5.17275×10^{13}	9.24457×10^7	1.19097×10^{12}
Termination (CTL 6)	4.5sec	2min58sec	4.2sec	4min1sec
# reachable states	3.63844×10^6	2.67210×10^9	546831	6.35545×10^7

gorithm, proposed by Aspnes and Herlihy [1], was verified. The authors of [20] separated the algorithm into a probabilistic component and a non-probabilistic component. They applied standard probabilistic model checking techniques to the probabilistic component. In the verification of the non-probabilistic part, whose state space is infinite, they relied on proof techniques that reduce the verification problem to small problems that can be solved by model checking.

In [16] and [24], model checking was used for debugging purposes in developing TLA specifications for the *Disk Paxos* algorithm and the *Fast Paxos* algorithm. The models that were model checked consisted of two or three processes and a small number of rounds [23]. Such small-sized models cannot be used to assure the correctness of the algorithms but are useful for detecting simple bugs.

In [35], automatic verification was applied to optimistically terminating consensus (OTC), an abstraction for a *single* phase with "full-information" exchange of a coordinator-based algorithm. The termination of the consensus algorithm and the selection of a coordinator are not handled. The objective of [35] was to automatically discover one phase that can be extended to full algorithms. Our approach differs from [35] in many aspects. For example, our approach can verify the *entire* consensus algorithm, including the verification of the termination property.

In [18], a *synchronous* consensus algorithm proposed in [2] was analyzed with a real-time model checker for the case $n = 3$.

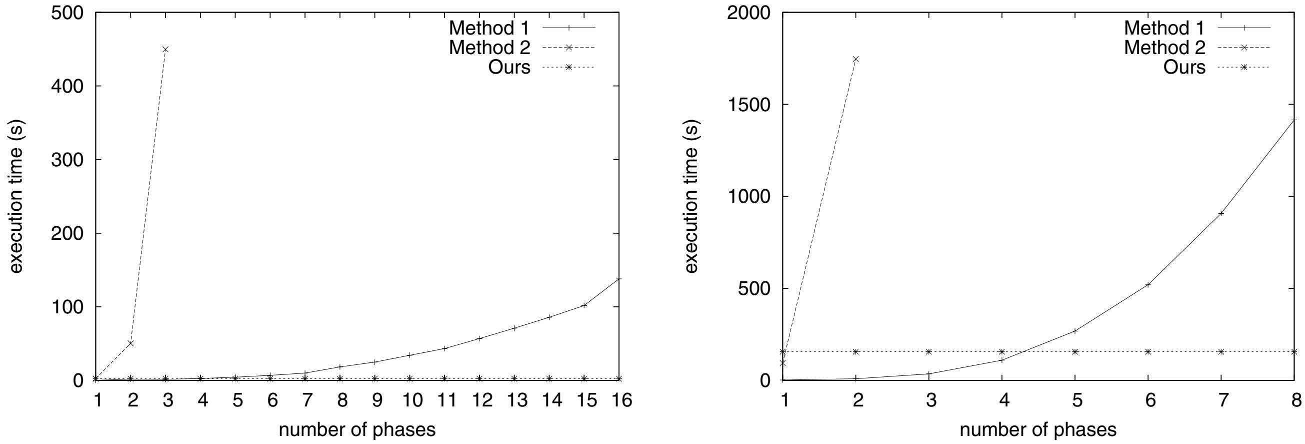
The applications of formal verification methods other than model checking to consensus algorithms can be found in, for example, [27, 28, 30].

8. Concluding Remarks

In this paper, we presented a model checking approach for verifying HO model-based 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.

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 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.

As future research, we will try to extend our techniques to be able to handle a larger number of processes. For example, as observed in [4], the structure of agreement protocols may exhibit various types of symmetries. Exploiting these symmetries to reduce the state space can be an interesting topic. However, from a practical point of view $n = 3$ (for *Paxos*/*LastVoting*) and $n = 4$ (for *OneThirdRule*) are usually sufficient: in practice, replication with strong consistency is restricted to a small number of replicas. Note that we were also able to model check an algorithm with conditions of type $|HO(p, r)| > n/2$ for $n = 5$. The result was obtained for the *UniformVoting* algorithm [9], an algorithm that can be viewed as a multi-valued deterministic version of the well-known randomized consensus algorithm by Ben-Or [3]. Moreover, we model checked algorithms under development and discovered bugs. These bugs were already discovered with $n = 3$.



(a) $n = 3$ (b) $n = 4$

Figure 3. Performance of verifying models with bounded phases. The horizontal axis indicates the number of phases model checked. The vertical axis represents the time used to verify the agreement of the *LastVoting* algorithm.

The verification of lower level algorithms (algorithms that implement communication predicates for HO based algorithms, or that implement failure detectors for failure detector based algorithms) also deserves further study. We have started working on the verification of such algorithms for HO predicates: we applied real-time model checking to one of the predicate implementations proposed in [19] and obtained preliminary results on its timeliness properties [34].

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

References

[1] J. Aspnes and M. P. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, 1990.

[2] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of ACM*, 41(1):122–152, 1994.

[3] 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.

[4] P. Bokor, A. Pataricza, M. Serafini, and N. Suri. Exploiting symmetry of distributed FT protocols to ease model checking. In *Supplemental Proc. Int'l Conf. on Dependable Systems and Network (DSN 2007)*, pages 358–359, Edinburgh, UK, June 2007.

[5] 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 *LNCS*, pages 42–50, London, UK, 2001. Springer-Verlag.

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

[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)*, pages 287–295, Riverside, CA, USA, Dec. 2006. IEEE CS Press.

[8] B. Charron-Bost and A. Schiper. Harmful dogmas in fault tolerant distributed computing. *SIGACT News*, 38(1):53–61, Mar. 2007.

[9] B. Charron-Bost and A. Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Failures. Technical Report LSR-REPORT-2007-001, EPFL, 2007.

[10] 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 *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[11] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[12] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. 22(3):161–180, 1996.

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

[14] 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.

[15] E. Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC-17)*, pages 143–152, New York, NY, USA, 1998. ACM Press.

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

[17] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Trans. on Software Engineering*, 27(1):29–41, 2001.

[18] M. Hendriks. Model checking the time to reach agreement. In P. Pettersson and W. Yi, editors, *3rd International Conference on the Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, volume 3829 of *LNCS*, pages 98–111. Springer–Verlag, 2005.

[19] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Proc. Int'l Conf. on Dependable Systems and Network (DSN 2007)*, pages 92–101, Edinburgh, UK, June 2007. IEEE CS Press. Full version in technical report LSR-REPORT-2006-006, EPFL, 2006.

[20] 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 *LNCS*, pages 194–206, Paris, France, July 2001. Springer.

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

[22] L. Lamport. Real-time model checking is really simple. In *Proc. of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 162–175. Springer, Oct. 2005.

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

[24] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, Oct. 2006.

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

[26] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of 39th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[27] 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.

[28] 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 *LNCS*, pages 393–407, Marseille, France, Sept. 2003. Springer.

[29] 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 *LNCS*, pages 44–61, Toulouse, France, Oct. 2002. Springer.

[30] 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.

[31] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings of the 6th Annual Symposium on Theoretical Aspects of Computer Science (STACS'89)*, volume 346 of *LNCS*, pages 304–313, Paderborn, Germany, Feb. 1989. Springer.

[32] A. Schiper and S. Toueg. From Set Membership to Group Membership: A Separation of Concerns. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(1):2–12, Jan.-March 2006.

[33] T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. Technical Report LSR-REPORT-2006-005, EPFL, Dec. 2006.

[34] T. Tsuchiya and A. Schiper. An automatic real-time analysis of the time to reach consensus. 2007. Submitted for publication.

[35] P. Zieliński. Automatic verification and discovery of Byzantine consensus protocols. In *Proc. Int'l Conf. on Dependable Systems and Network (DSN 2007)*, pages 72–81, Edinburgh, UK, June 2007. IEEE CS Press.