Springer Software Quality Journal
Author's Accepted Manuscript

Reductions and Abstractions for Formal Verification of Distributed Round-based Algorithms

Raul Barbosa · Alcides Fonseca · Filipe Araujo

This is a post-peer-review, pre-copyedit version of an article published in Software Quality Journal. The final authenticated version is available online at: http://dx.doi.org/10.1007/s11219-020-09539-6

Abstract Model checking has advanced over the last decades to become an effective formal technique for verifying distributed and concurrent systems. As computers grew in memory and processing capacity, it became possible to exhaustively verify systems with billions of states, making it practical to model and verify real-world protocols and algorithms. However, writing a model is a manual task that potentially introduces defects which the model checker tool finds to fulfill the formal specification (*e.g.*, an incorrect model that fulfills an incomplete specification). Furthermore, this kind of formal verification technique is limited by the well known state-space explosion problem. This paper aims to provide a set of generic template models, appropriate for distributed round-based algorithms, to be used to focus modeling effort on algorithm-specific details. To mitigate state-space explosion, the paper proposes two reduction techniques, named partition symmetry reduction and message order reduction, that exploit symmetries in the state-space to avoid expanding equivalent states. The reusable framework for verifying round-based algorithms and the two proposed reduction techniques provide the means for reducing by orders of magnitude the number of states required to analyse common distributed algorithms.

Keywords Model checking · Distributed algorithms · Formal verification

R. Barbosa
CISUC, Department of Informatics Engineering
University of Coimbra
P-3030 290, Coimbra, Portugal
Tel.: +351 239 790 024
Fax: +351 239 701 266
E-mail: rbarbosa@dei.uc.pt

1 Introduction

Distributed algorithms are at the core of modern computing infrastructures and have been the subject of substantial research. Although much progress has been made in understanding fundamental problems such as consensus, reliable multicast and atomic commit (Lynch, 1996), it is still notoriously difficult to verify the correctness of reliable algorithms and to quantify their performance and reliability.

Several factors make it difficult to analyze distributed systems, notably the potential interleaving of events, asynchronous communication and failures, all of which contribute to a combinatorial explosion of the space of reachable states. Since one must usually cover all possible executions to guarantee correctness and adequate performance, it is often a challenge to ensure that a distributed system is dependable and secure.

Model checking is widely regarded as an effective formal technique for verifying distributed systems and algorithms (Clarke et al., 2018). It consists of checking whether a model fulfills a given specification, through exhaustive verification of the corresponding finite-state automaton. Specifications are given in some form of logic (*e.g.*, Computational Tree Logic, Linear Temporal Logic) and models are written in a formal language to allow for rigorous analysis. While model checking can verify complex systems with billions of reachable states, the practical application is limited by computational constraints.

For a model checker to exhaustively verify whether a finite-state model of a system meets the specified properties, all reachable states must be examined. The state-space explosion problem arises whenever it becomes computationally too expensive to analyze the full state-space. This is one of the main challenges to the application of model checking to dynamic systems. Another fundamental challenge is to manually construct a *small sufficient model* (Holzmann, 2003) that correctly captures all relevant system details while resulting in a tractable number of states for automatic analysis.

This paper addresses these two challenges in the context of distributed systems using round-based algorithms (Lynch, 1996). The main contributions are the following:

- *Two reduction techniques to mitigate the state-space explosion problem.* We propose *partition symmetry reduction* (PSR) to reduce the state-space by exploiting the symmetry that arises when processes choose a non-deterministic initial value. Rather than exploring all possible combinations of values for all processes, only the representative partitions are expanded. This reduces the number of initial transitions performed by the model checker. Furthermore, we propose *message order reduction* (MOR) that exploits the observation that in round-based algorithms all messages are received or missed in the same round in which they are sent. Therefore, if an algorithm's state-transition function is commutative (*i.e.*, processing order is irrelevant) then any possible transition results in the same local state as a single representative order.

- *A reusable framework for verifying round-based algorithms.* The paper provides a reusable template composed of a generic and a specific part. The generic part is a model of round-based computation for distributed algorithms and the specific part models the behavior of each concrete algorithm that is to be verified. One who wishes to verify a new round-based algorithm will focus the effort on the specific part of the model. The framework includes an implementation of both reduction techniques (PSR and MOR) written in Promela for the Spin model checker (Holzmann, 2003). The goal is to reduce the effort of practitioners in verifying round-based protocols.
- *A translation methodology for round-based algorithms to Symbolic Model Checking.* To evaluate the impact of both the framework and reduction techniques on symbolic model checking approaches, we show how to encode the algorithms and their specifications as a symbolic satisfiability problem, taking advantage of advances in SMT solvers (Clarke et al., 2001) for model checking.
- *An experimental evaluation of both reduction techniques.* Using well-known protocols in the literature, we apply PSR and MOR to show that the theoretical gains can be obtained in practical model checking applications. In particular, we perform evaluation using the Spin Model Checker and Z3, an SMT solver for the symbolic approach.

Our work addresses algorithms designed according to the round-based computation model. Round-based algorithms (Charron-Bost and Schiper, 2009; Gafni, 1998; Elrad and Francez, 1982) are structured in rounds of computation. At each round every process computes a new message exclusively based upon the local state, sends the message to all processes, waits to receive messages from other processes, executes a local state-transition function based upon incoming messages and advances to the following round. This structure is explored in the present paper to construct small sufficient models for verifying new algorithms.

The remainder of this paper is structured as follows. Section 2 summarizes the related work in the field. Section 3 describes the computation model of round-based algorithms and focuses on the key observations for writing a formal model of such algorithms. Section 4 formalizes two algorithms that are used in the paper as running examples. Sections 5 and 6 formalize the two reduction techniques proposed in this paper and Section 7 provides the proofs of validity of the proposed reductions. Section 8 describes the full implementation of the generic verification framework, in the Promela language, for the Spin model checker. Section 9 provides a translation methodology from the Promela templates to SMT encodings. Section 10 summarizes the main conclusions of the paper and outlines implications for practice.

2 Related work

Temporal logic model checking (Clarke et al., 1986) automates the process of verifying if a model fulfills a given specification. It is a form of deductive

reasoning to formally verify finite state systems with concurrency and message-passing. Correctness properties are expressed using temporal logic and the model checker performs an exhaustive search to look for states in which some property is violated. Exhaustive search without finding any counterexample formally implies that the specification holds.

Exhaustively searching the state-space of an algorithm often leads to combinatorial explosion of possible executions, thereby consuming all available memory or too much computation time. In this context, model checking tools such as Spin make use of partial order reduction (Peled, 1994) to optimize the search process. Partial order reduction exploits the fact that when two concurrent processes access only the local state, the transitions are commutative, *i.e.*, the same state results from choosing to execute the processes in any order. The two reductions proposed in the present paper complement partial order reduction and further eliminate equivalent states from the verification process.

Symbolic model checking (Burch et al., 1992) uses binary decision diagrams to represent state-transitions and is widely used to formally verify hardware systems (Eisner and Peled, 2002). Explicit-state model checking (Holzmann, 2003) exhaustively generates the graph of global states reachable by a system and this approach is most frequently used for verifying distributed and concurrent systems. The present paper addresses explicit model checking.

The key challenges in model checking are the manual construction of a system model and the laborious task of mitigating the state-space explosion problem. To deal with these challenges, one should aim for the notion of smallest sufficient model (Holzmann, 2003). A model must be a faithful representation of the original system, while abstracting away irrelevant details to reduce the state-space. This is often a delicate balance.

Combinatorial growth of the number of reachable system states has sparked interest in research of methods to tackle this problem. Bounded model checking (Tsuchiya and Schiper, 2008), symbolic verification (Clarke et al., 1996), diverse techniques for model abstraction (Clarke et al., 1994, 2000) and partial order reduction (Peled, 1994) all aim to mitigate the state-space explosion problem. Furthermore, explicit model checkers have the ability to use compression techniques to reduce the memory footprint of the verifier Holzmann (2003). The present paper not only builds upon existing techniques (*e.g.*, partial order reduction and compression) but also proposes further reductions that are applicable to the specific case of round-based algorithms (Lynch, 1996).

Other authors have proposed to use specific abstractions for fault-tolerant distributed algorithms (Aminof et al., 2018). More generally, symmetry reduction techniques have proven useful to reduce the state-space explosion problem (Emerson and Sistla, 1996). The key idea, which we also adopt in the present paper, is to exploit symmetries existing in the global system state to mark equivalent states. This is usually performed by pruning the search whenever a state is revisited. Several practical implementations exist, including a prototype implementation named SymmSpin (Bosnacki Dragan and Holenderski, 2002) for the Spin model checker. The reduction techniques proposed in the present paper exploit symmetry in the state space beforehand, by avoiding

the generation of symmetric states *a priori*, rather than waiting for a revisited state.

Our work builds upon the notion of communication-closed rounds of communication (Elrad and Francez, 1982) in which processes compute local state transitions based upon the messages received from other processes. Because of failures and asynchrony, we model message omissions from the receiving side, similarly to the heard-of model (Charron-Bost and Schiper, 2009) that can be used to model any kind of benign failure.

At this point, it is worthwhile discussing round-based probabilistic algorithms, as well as asynchronous systems. A practical way of using asynchronous systems is to use local clocks to delete late messages, as in the *timed asynchronous* model (Cristian and Fetzer, 1999). We can follow the same approach in round-based algorithms and model late messages as omissions, e.g., in asynchronous round-based broadcast protocols (Srikanth and Toueg, 1987). A problem with this approach is that correct processes that have not crashed may fail to send messages in time, thus making the failure model more complex.

One should notice that model checking does not suit probabilistic algorithms well, as these may not be live against an adversary that always picks an inconvenient random value. Hence, our technique would either not apply or require some modifications for well-know algorithms, like the Ben-Or's consensus algorithm (Ben-Or, 1983).

In some specific systems, where variables have symmetric roles, e.g., where $a = 2$, $b = 1$ and $c = 0$ represents a similar state to $a = 1$, $b = 2$ and $c = 0$, the partition symmetry approaches may also simplify liveness or safety analysis. This could be the case of counter machines (Minsky, 1961).

Under the heard-of model of received and missed messages, it is possible to reduce the state-space by visiting less executions as long as the sequence of events is preserved (Chaouch-Saad et al., 2009). The authors prove a reduction theorem, built upon the fact that rounds are communication-closed, by distinguishing between fine-grained and coarse-grained executions. A coarse-grained model is shown to be an abstraction of the fine-grained model, with less state-transitions.

3 Communication model

A system is composed of n processes exchanging messages through directed communication channels and computations consist of *rounds*. Round-based algorithms (Gafni, 1998; Elrad and Francez, 1982) follow a well-defined pattern: at each round every process computes a new message exclusively based upon the local state, sends the message to all processes, waits to receive messages from other processes, executes a local state-transition function based upon incoming messages and advances to the following round.

Under this communication model two practical consequences are relevant. If a message is received during some round, then it must have been sent during that round. If a message is missed during some round, then it is discarded (in-

tentionally, if needed, by dropping the message if it arrives at a later round). Hence, message omissions and process crashes/hangs are given uniform treatment without the need to consider the root cause (Santoro and Widmayer, 2005).

An implementation of round-based communication requires strong guarantees that are abstracted using the notion of round-by-round fault detector (Gafni, 1998). Every process has a local module that provides, for each round, the set of processes from which no message should be expected. The root cause for such failures is abstracted away, thereby unifying process failures, link failures and network asynchrony. This communication model is widely adopted and there is vast literature on specific challenges such as round-based consensus (Marić et al., 2017; García-Pérez et al., 2018; Raynal, 2018).

Algorithm 1: Structure of processes in round-based algorithms.

```
loop forever:
    begin_round()
    msg = compute_message()
    send_to_all(msg)
    wait_to_receive()
    state_transition()
    end_round()
end
```

Algorithm 1 shows the structure of a process according to the round-based pattern of distributed communication. Every process has a local state that depends on the particular protocol being modeled. Six different functions access the local state and compute the necessary transitions. It is worthy to note that only two of these functions are protocol-specific: the `compute_message` function that determines the new message based exclusively upon the local state; and the `state_transition` function that reads all received messages and computes the new local state for the process.

In this computation model, failures and asynchrony can be specified in terms of heard-of sets (Charron-Bost and Schiper, 2009), which are the sets of processes from which every process receives a message in every round. Formally, considering Π the set of n processes, the set $\mathrm{HO}(p, r) \in \Pi$ contains the processes from which p receives a message in round r. In our models, the same notion is formalized as an `rm` vector (*i.e.*, a representation of the *received messages* during each round). Under these conditions, in any sequence of rounds (finite or infinite) the local states of processes are uniquely determined by initial non-deterministic values selected and by the sets of received messages.

4 Protocols

We illustrate our reduction techniques with the help of two protocols: the FloodSet (Lynch, 1996) and Herlihy's n-process consensus (Herlihy, 1991).

Algorithm 2: *FloodSet* Consensus algorithm in a synchronous system (Lynch, 1996)

```
// Algorithm for process p_i ∈ g
On initialization:
    Values_i^1 = {v_i}
    Values_i^0 = {}
end
In round r, for 1 ≤ r ≤ f + 1:
    // Send only values not sent yet
    B-multicast(g, Values_i^r \ Values_i^{r-1})
    Values_i^{r+1} = Values_i^r
    On B-deliver(V_j) from some process p_j:
        Values_i^{r+1} = Values_i^r ∪ V_j
    end
end
After f + 1 rounds:
    d_i = min(Values_i^{f+1})
end
```

The FloodSet consensus, in Algorithm 2, tolerates up to f process stopping failures and runs in $f + 1$ rounds. Initially, each process picks its own value v_i and sends this value to the peers. Since faulty processes might deliver their values to a subset of the correct processes before stopping, the main challenge is to propagate such values to every non-faulty process. For this end, each process uses best-effort multicast ($B - multicast$), which, in the worst case, might need up to $f+1$ rounds to cope with f failures. Once processes converge in the set of values ($Values$), they can use a deterministic function over the set, like the minimum function, to determine the agreed value.

Algorithm 3: *Compare&Swap*

```
Data: register r, old value in the register, new value to set to the register
Result: Return the value of the register r at call time and conditionally change it
previous = r
if previous = old then
    r = new
end
return previous
```

Before introducing the n-process consensus, we review the $Compare\&Swap$ primitive. Although this is not apparent in Algorithm 3, this primitive is atomic; intuitively, we can think as if no two processes could be inside the function at the same time. Processes compete to change the value of register r: if they arrive on time, the comparison $previous = old$ will be true and the change will occur; otherwise it will not.

Based on the compare-and-swap operation, the n-process consensus, in Algorithm 4, returns either the $input$ or the $first$ value set on the $Compare\&Swap$

Algorithm 4: n-process consensus with compare and swap (Herlihy, 1991)

Data: Proposed value *input*; *shared* register
Result: The agreed value
$first = Compare\&Swap(shared, \perp, input)$
if $first == \perp$ **then**
  return *input*
else
  return $first$
end

register. Only the first process arriving will manage to change the register according to its *input*; all the following will see the *input* set by the first one. Assuming the availability of the *Compare&Swap* primitive, this algorithm is *wait-free*: non-faulty processes can finish the algorithm in a finite number of steps, regardless of other process behaviours.

5 Partition symmetry reduction

A frequent pattern in distributed algorithms is the non-deterministic selection of an initial value for each process. Notable examples are the consensus problem and leader election. Each of the n processes begins by selecting one out of n possible values, giving rise to n^n distinct possibilities during initialization, even before the distributed algorithm is executed.

We propose a technique hereby referred to as *partition symmetry reduction* to expand fewer initial combinations of values. To illustrate with an example, consider a set of 4 processes that may select one out of 4 possible initial values. Full state-space analysis would result in $4^4 = 256$ possibilities. However, if the set of values is unordered, there is no need to consider all permutations. Instead, one may expand only the partitions of n. In number theory, a partition of natural number n is one way to write the number as a sum of natural numbers (Bóna, 2002). When $n = 4$ the five possible partitions are $1+1+1+1$, $2+1+1$, $2+2$, $3+1$ and 4. Respectively, the 4 processes choose distinct values, two processes choose the same value and two processes choose distinct values, two processes choose the same value and the other two choose another value, three processes choose the same value and the other process chooses a distinct value, and the 4 processes choose the same value. Hence, a system with $n = 4$ has $4^4 = 256$ permutations with repetition while having only five partitions.

The partition function $p(n)$ yields the number of partitions of natural number n and was notably studied by Euler. Natural numbers here correspond to the positive integers. The asymptotic formula

$$p(n) \sim \frac{1}{4n3^{\frac{1}{2}}} e^{\pi(\frac{2n}{3})^{\frac{1}{2}}} \tag{1}$$

has been known for over a century (Hardy and Ramanujan, 1918; Erdős, 1942) and it is trivial to prove that

$$\lim_{n\to\infty} \frac{p(n)}{n^n} = 0 \tag{2}$$

implying that symmetry reduction by expanding only the partitions of n, rather than the full set of permutations with repetition, results in a much smaller state-space.

Figure 1 presents a complete model of Herlihy's wait-free consensus protocol for n processes (Herlihy, 1991). The presented model is written for the Spin model checker (Holzmann, 2003) and is therefore formalized in the Promela language.

Fig. 1 Wait-free n-process consensus protocol formalized in Promela.

```
#define n 5

byte r;
byte ghost;

inline compare_and_swap(r, old, new, result) {
    atomic {
        previous = r;
        if
        :: previous == old -> r = new
        :: else -> skip
        fi;
        result = previous
    }
}

proctype Process(byte input) {
    byte previous, first, decision;
    compare_and_swap(r, 0, input, first);
    if
    :: first == 0 -> decision = input
    :: else -> decision = first
    fi;
    ghost = decision;
    assert(ghost == decision)
}

init {
    byte v, i;
    atomic {
        for(i : 1 .. n) {
            select(v : 1 .. n);
            run Process(v)
        }
    }
}
```

The `init` process in Figure 1 formally specifies the initialization procedure frequently constructed when modeling distributed algorithms. A loop iterates from 1 through n and the `select` statement chooses an initial value v for each process that is started with the `run` statement. Clearly, non-deterministic

selection of n values for n processes results in full state-space analysis for all permutations.

Figure 2 provides an implementation of the proposed partition symmetry reduction technique. The `init` process computes the possible partitions of n processes and serves as a drop-in replacement for the `init` process presented in Figure 1. It can be reused in similar contexts and, conceptually, it could be built into an existing model checker by extending the modeling language.

Fig. 2 Formalization of partition symmetry reduction in the Promela language.

```
init {
    byte partition[n];  // holds the current partition
    byte v, i = 1;      // i points to the partition's end
    byte count;         // counts processes in each group
    partition[0] = n;   // the initial partition is just [n]
    atomic {
        do
        :: partition[0] != 1 ->  // if more partitions exist
           v = 1;                // compute the next partition
           do
           :: partition[i-1] == 1 ->
              i--;               // index of last element > 1
              v++                // elements to be updated
           :: else -> break
           od;
           partition[i-1]--;     // update partition elements
           do
           :: partition[i-1] < v ->
              v = v - partition[i-1];
              partition[i] = partition[i-1];
              i++
           :: else -> break
           od;
           i++;
           partition[i-1] = v
        :: true ->               // use the current partition
           v = i;                // begin with highest value
           count = 0;
           do
           :: count < partition[v-1] ->
              run Process(v);    // start and count processes
              count++            // in group that has value v
           :: else ->
              v--;
              count = 0;
              if
              :: v == 0 -> break // end with lowest value
              :: else -> skip
              fi
           od;
           assert(_nr_pr == n+1);
           break
        od
    }
}
```

The proposed implementation of partition symmetry reduction, in the Promela language, presented in Figure 2, consists of an outer `do`-loop that calculates the possible partitions of n processes in lexicographic order. It begins with the partition composed of n itself and iteratively computes the fol-

lowing partitions. The `do`-loop has two alternatives: choosing line 8 calculates the next partition; choosing line 26 uses the current partition to run the processes. These two alternatives are non-deterministic because the guards may overlap, *i.e.*, at each execution of the `do`-loop the model checker can either select to compute the next partition (in line 8 the guard evaluates to `true` while there are more partitions left) or select to use the current partition (in line 26 the guard is always `true`). The `assert` statement in line 41 formalizes a trivial correctness check imposing that the number of processes equals $n+1$ (n processes initialized plus the `init` process).

The initial partition, which consists of n itself, leads to all processes selecting the same value. The last possible partition consists of $1+1+...+1$ adding up to n and leads to all processes selecting distinct values. Non-deterministic transitions allow anything in between these two extremes to be executed.

In Figure 1 the `init` process runs a new process in line 33 by passing as argument the non-deterministic value v. Using partition symmetry reduction, in Figure 2 the `init` process runs a new process in line 31 by passing the value v of the partition in which that process was non-deterministically placed. Therefore, the `init` processes in Figures 1 and 2 are interchangeable and correspond, respectively, to disabling and enabling partition symmetry reduction.

Partition symmetry reduction is valid under the condition that the set of values selected by processes is unordered. This condition is met by Herlihy's wait-free consensus protocol given the fact that initial values have no semantics and process identifiers are never used to distinguish behavior among different processes. These characteristics are common to a wide range of distributed and concurrent algorithms and, hence, the proposed reduction is valid for those algorithms. Constructing a static analyser to automatically check whether an algorithm meets these conditions is an interesting research question for future work.

5.1 Experimental evaluation

To compare full state-space analysis with partial symmetry reduction, we executed exhaustive verification runs using the Spin model checker with n ranging from 2 to 9 processes. The results are presented in Table 1.

The results were obtained using Spin version 6.5.0 running on a 2.7GHz Intel Core i7 with 16GB of 2133MHz RAM. The verifier generated by Spin was compiled with the *collapse* directive, to reduce memory by compressing the state vector, and with the *safety* directive, to optimize both speed and memory by disabling cycle detection. The verifier was compiled with the `-O2` optimization flag and Spin's partial order reduction was enabled.

Table 1 compares the results of partition symmetry reduction with full state-space analysis of Herlihy's wait-free consensus protocol for n processes. The columns on the left hand side show how the number of states, the memory used for states and the verification time grow with the reduction technique

Table 1 Verification results for Herlihy's wait-free consensus protocol.

	Partition symmetry reduction			Full state-space			
n	States	Memory	Time	States	Memory	Time	Ratio
2	99	0.3 MB	∼0 s	183	0.3 MB	∼0 s	54.10%
3	1028	0.3 MB	∼0 s	8314	0.6 MB	∼0 s	12.36%
4	10882	0.7 MB	∼0 s	4.9×10^5	18.8 MB	∼0 s	2.22%
5	8.9×10^4	3.6 MB	∼0 s	3.4×10^7	1.4 GB	30 s	0.26%
6	7.7×10^5	29.7 MB	∼0 s	—	—	—	—
7	5.6×10^6	245.0 MB	4 s	—	—	—	—
8	4.2×10^7	2.0 GB	42 s	—	—	—	—
9	2.8×10^8	14.3 GB	439 s	—	—	—	—

enabled (using the `init` from Figure 2) whereas the columns on the right hand side show the same results without reduction (using the complete code from Figure 1 unmodified). The rightmost column provides the ratio between the number of states required by disabling/enabling the reduction.

There are two main observations from the results in Table 1. First, for a system with $n = 5$ the state-space is three orders of magnitude smaller by using partition symmetry reduction. This is very much in line with the theoretical observation that $5^5 = 3125$ permutations with repetition, while $p(5) = 7$ possible partitions for 5 processes, yielding the exact ratio of $7 \div 3125 = 0.224\%$ which is close to the 0.26% ratio obtained using Spin.

Second, it was possible to exhaustively verify systems composed of up to $n = 9$ processes using partition symmetry reduction, while $n = 6$ exhausted the 16GB of available memory using full state-space verification without completing the verification process. Once main memory is fully occupied the swapping strategy renders the model checking process impractically slow. Hence, for Herlihy's protocol, the proposed reduction allows systems with $n = 9$ processes, in present workstations, and nearly doubles the number of processes that are verifiable by default.

6 Message order reduction

Most distributed algorithms are designed to handle non-deterministic order of message exchanges, as this is the norm in real systems. It is frequently the case that the order in which messages are sent, received and processed is irrelevant for the final result. This is the basis for the technique of *message order reduction*.

Regardless of the order in which messages are transmitted, it is preferable to reach a deterministic result, instead of leaving it to chance. Therefore, most algorithms ensure this property by design. Round-based algorithms provide a convenient abstraction built around this property. At each round, every process computes the message to be sent, sends the message to all processes, waits to receive all messages and executes a local state-transition function to compute the new state. When the order of message transmission is irrelevant, one may select any specific order as representative of all orders.

Figure 3 presents a complete model of the algorithm for synchronous consensus described in Lynch's book (Lynch, 1996). The presented model, for the Spin model checker, is a direct formalization of the algorithm written in the Promela language.

Fig. 3 Synchronous consensus algorithm formalized in Promela.

```
#define n 4
#define f 1

chan channels[n] = [n] of { byte };
byte round = 1, nr_faulty = 0, nr_active = 0, decision;

proctype Process(byte v; chan c) {
    byte values = 1 << (v-1);
    byte r = 1, t;
    do
    :: r >= 1 && r <= f+1 ->
       t = 0;
       do
       :: t < n ->
          if
          :: atomic { nr_faulty < f -> nr_faulty++; }
end:         (false)
          :: true -> skip
          fi;
          channels[t] ! values;
          t++
       :: else -> break
       od;
       do
       :: nempty(c) ->
          c ? t;
          values = values | t
       :: round != r -> break
       od;
       r++
    :: r > f+1 -> break
    od;
    t = 0;
    do
    :: values & (1 << t) -> break
    :: else -> t++
    od;
    v = t+1;
    decision = v;
    assert(decision == v);
    nr_active--
}

init {
    byte r, i;
    atomic {
        for(i : 1 .. n) {
            select(r : 1 .. n);
            run Process(r, channels[i-1]);
            nr_active++
        }
    }
    do
    :: timeout && (nr_active - nr_faulty) >= 1 -> round++
    :: (nr_active - nr_faulty) == 0 -> break
    od
}
```

One of the main causes of the complexity of verifying a model like the one presented in Figure 3 rests in the order that messages are transmitted and processed by receiving nodes. Each of the n processes has a channel for incoming messages. The `do`-loop in lines 13–23 sends the current message to all processes (containing the values received). Given that this loop executes concurrently across all processes, any interleaving is possible and the model checker expands a vast number of states.

However, as can be observed both in Figure 3 and in Algorithm 2, regardless of the order of message transmission the local states of processes are the same at the end of every round (only failures and especially omissions have an effect). On delivery of V_j from some process p_j the set of values becomes the union of previously received values with the newly received values (line 27 in Figure 3). This operation is clearly commutative. Hence, the proposed message order reduction is valid for Lynch's synchronous consensus protocol. In fact, most protocols are designed this way to ensure the end result is independent of the order of transmission.

The code in Figure 3 initializes n processes in lines 46–52. Each process has a channel `c` to receive incoming messages. The set of values received from other processes is stored using bitwise operations in the `values` variable, which is initialized in line 8 with the non-deterministic value initially selected for the process. At each round, processes may fail by blocking in the end label in line 17, or otherwise send the `values` variable to all processes in line 20. Thereafter, the `do`-loop starting in line 24 reads all received messages and computes the local-state transition in line 27 by adding the received values to the `values` set. Once the execution of the protocol is completed, lines 39 and 40 formally specify that the decision made by any process is the same as the decision of any other process, *i.e.*, agreement.

The complexity of the verification procedure is influenced by the number of permutations of n messages transmitted by each of the n processes in every round. Each incoming channel has $n!$ possible permutations of the n messages and, therefore, globally there are $n \times n!$ possibilities at the end of every round. Given that message order reduction only expands one such permutation, the resulting state-space is much smaller.

Figure 4 shows a modified version of the consensus protocol formalizing the proposed message order reduction. A token is used to impose sequential sending and reception of messages.

The Promela code shown in Figure 4 maintains the same overall structure as the full state-space version in Figure 3. The `token` variable makes processes send messages in sequential order through blocking in line 12 and subsequently passing the token in line 31. Messages are also received in sequential order through blocking in line 32 and passing the token in line 40.

Table 2 summarizes the verification results and compares full state-space analysis (code from Figure 3) with message order reduction (code from Figure 4). The left hand side of the table shows the configuration regarding the number of processes n and the number of failures f occurring during execution.

Fig. 4 Synchronous consensus with message order reduction.

```
#define n 5
#define f 1

chan channels[n] = [n] of { byte };
byte token = 1, nr_faulty = 0, decision = 0;

proctype Process(byte v; chan c) {
    byte values = 1 << (v-1);
    byte r = 1, t;
    do
    :: r >= 1 && r <= f+1 ->
       (token == _pid);       // send in order
       t = 0;
       do
       :: t < n ->
          atomic {
            if
            :: nr_faulty < f ->
               nr_faulty++;
end_faulty:
               (token == _pid);
               token = token % n + 1;
               goto end_faulty
            :: true -> skip
            fi
          }
          channels[t] ! values;
          t++
       :: else -> break
       od;
       token = token % n + 1; // pass the token
       (token == _pid);       // receive in order
       do
       :: nempty(c) ->
          c ? t;
          values = values | t
       :: empty(c) -> break
       od;
       r++;
       token = token % n + 1  // pass the token
    :: r > f+1 -> break
    od;
    t = 0;
    do
    :: values & (1 << t) -> break
    :: else -> t++
    od;
    v = t+1;
    decision = v;
    assert(decision == v)
}
```

The results presented in Table 2 were obtained using exactly the same hardware and software configuration as the results in the preceding section. The columns on the left hand side show the number of states, the memory used and the verification time using the reduction techniques. The columns on the right hand side show the same results without the reduction. The rightmost column presents the ratio between the number of states required using the reduction and the default state-space analysis.

Table 2 Verification results for the synchronous consensus protocol.

	Message order reduction			Full state-space			
n, f	States	Memory	Time	States	Memory	Time	Ratio
2,1	1636	0.1 MB	∼0 s	5135	0.4 MB	∼0 s	31.9%
2,2	3192	0.3 MB	∼0 s	9091	0.7 MB	∼0 s	35.1%
3,1	59426	5.7 MB	∼0 s	5.1×10^6	485.8 MB	2 s	1.2%
3,2	4.0×10^5	37.7 MB	∼0 s	8.9×10^6	847.5 MB	4 s	4.5%
4,1	2.3×10^6	252.4 MB	1 s	—	—	—	—
4,2	3.1×10^7	3.4 GB	19 s	—	—	—	—
5,1	1.1×10^8	13.2 GB	91 s	—	—	—	—

Two main observations can be made from the results presented in Table 2. First, a system with $n = 3$ has a state-space that is between one and two orders of magnitude smaller by using message order reduction (both for $f = 1$ and $f = 2$). This is in line with the theoretical observation that 1 permutation of messages is expanded rather than $3 \times 3! = 18$, resulting in the ratio of $1 \div 18 \approx 5.6\%$. For $n = 2$ processes the ratio is also close to the ratios obtained in practice using Spin.

The second observation is that it is possible to verify systems with up to $n = 5$ processes using message order reduction, whereas $n = 3$ was the maximum achievable with full state-space verification, limited by 16GB of memory (with $n = 4$ the verification process used up all memory and started the swapping mechanism). Therefore, for the synchronous consensus protocol, the proposed message order reduction allows one to increase from $n = 3$ to $n = 5$ processes in present workstations.

7 Proofs of reduction techniques

The reasoning behind the proposed reduction techniques is based upon observations of the kinds of behaviour that can be abstracted from the models, while retaining sufficient detail for the verification process. In this section we construct proofs of the reduction techniques.

7.1 Partition symmetry reduction

Consider A to be a round-based algorithm. Without loss of generality, we assume that the state of each process is a positive integer of arbitrary size. The state of a failed process is 0, which we assume to be its output as well. No other process has state 0. Each round, a process sends its state in a message to the peers. Transition from one round to the next in A is, therefore, a function $\mathbf{f} : \mathbb{N}_0^n \to \mathbb{N}_0^n$, for n processes. Process i uses the global state exchanged in the round, to produce its own state, which is the i^{th} component of $\mathbf{f}$'s output.

We restrict our demonstration to algorithms A characterized by a "permutation-resistant" function $\mathbf{f}$, such that for any permutation P, $P \cdot \mathbf{f} \cdot P^{-1} = \mathbf{f}$, where "$\cdot$" means "after". As an example, if $\mathbf{f}([1, 2, 1]) = [3, 4, 3]$, a permutation

that switches the last two elements in the input will do the same to the output: $\mathbf{f}([1,1,2]) = [3,3,4]$; that is, we apply in the output the inverse permutation we applied to the input. This happens, for example, when the output of $\mathbf{f}$ is the most common value in the messages. It does not happen if the function selects as outputs the first element of the input.

We can now build the similar notion of "permutation-resistant assertion". In a "permutation-resistant assertion", if assertion Q is true for the state vector $\mathbf{v}$, it is also true after any permutation P, such that $\mathbf{w} = P(\mathbf{v})$. Based on these definitions, in Theorem 1, we demonstrate the validity of the partition symmetry reduction for a round-based algorithm A characterized by a permutation-resistant function.

Theorem 1 *A permutation-resistant assertion on A, characterized by a permutation-resistant function, is always true under partition symmetry reduction (C_1) if and only if it is also always true without partition symmetry reduction (C_2).*

Proof The theorem states the equivalence between two conditions, such that $C_1 \Leftrightarrow C_2$. $C_2 \Rightarrow C_1$ is trivial to prove. Since the space of cases without partition symmetry contains the space with partition symmetry reduction, if all cases are true in the larger space, they are also true in the contained space.

To see that $C_1 \Rightarrow C_2$, we reason as follows: having two successive rounds where we apply P on the inputs and P^{-1} on the outputs for both rounds is the same as applying only P on the inputs of the first round and P^{-1} on the outputs the second round. I.e., $\mathbf{f} \cdot \mathbf{f} = P \cdot \mathbf{f} \cdot P^{-1} \cdot P \cdot \mathbf{f} \cdot P^{-1} = P \cdot \mathbf{f} \cdot \mathbf{f} \cdot P^{-1}$. By induction, we can apply this reasoning to r rounds. This corresponds to converting the inputs on the first round and, upon getting the final result, converting it back using the inverse permutation. Given the permutation-resistant assertion, the theorem follows.

7.2 Message order reduction

Assume that B is a round-based algorithm, where each process determines its state transition based on the *set* of messages it received so far. This is a subset of the round-based algorithms of Algorithm 1, where the order of the received messages is not relevant; only the messages (and the ones that the process misses) are relevant. This is true whenever the operation performed on the messages is commutative and associative, a common case in reality often explored by *reduce()* operations, such as in the Message Passing Interface (Hughes and Hughes, 2003) or the MapReduce (Dean and Sanjay Ghemawat, 2004) programming model. In Theorem 2, we demonstrate the validity of message order reduction:

Theorem 2 *Assertions on B are always true under message order reduction (C_1) if and only if they are also always true without message reduction (C_2).*

Proof Similarly to the previous case, the theorem states the equivalence $C_1 \Leftrightarrow C_2$. $C_2 \Rightarrow C_1$ is trivial to prove. Since the space of cases without message

reduction contains the space with message reduction, if all cases are true in the larger space, they are also true in the contained space.

For $C_1 \Rightarrow C_2$, assume that C_1 holds, but C_2 does not. This would mean that all cases with order reduction are true, while, at least one case without order reduction is not true. Since, for the same set of messages, both cases take the same decision, then, the case without order reduction must find a specific set of messages, which does not exist with message reduction, where the assertion is false. But this is a contradiction, because the case with order reduction should find all possible sets of messages.

8 Reusable verification framework for round-based algorithms

We now turn to specifying a generic framework for verifying round-based algorithms. The goal is to provide a reusable template comprised of two parts: a generic part that models the round-based computation model of distributed systems; and a specific part that models the behavior of each concrete protocol. If one wishes to model and verify a new algorithm, only the specific part requires effort.

Figure 5 provides the proposed generic model of round-based algorithms, for the Spin model checker. A process follows the elementary structure shown in Algorithm 1, whereby at each round a new message is computed, sent to all processes, a wait is performed to receive incoming messages and a state-transition function computes the state for the next round. This is modeled in lines 32–39 in Figure 5.

The generic model shown in Figure 5 declares one `broadcast` channel for each processes, used to send messages. A `token` variable is declared to synchronize processes while sending and receiving messages. For a process to begin a round, it blocks until the token reaches its *pid*, as written in line 5, to wait for its turn. It sends its `msg` variable to all processes by calling the inline function in lines 17–23. This function sends the message to the broadcast channel just after ensuring that the broadcast channel is empty. Then, the process passes the token to the next process, in line 13, and waits for the appropriate turn in line 14. The round ends in line 9 simply by passing the turn and the `do`-loop begins again for the subsequent round.

The code in Figure 5 provides the generic part of round-based algorithms. This is the reusable system model. Regarding the specific part of an algorithm, one should write the code for inline functions `compute_message` (called in line 34) and `state_transition` (called in line 37). The first inline function reads the current local state of the process and updates the `msg` variable; and the second function reads all received messages and updates the local state.

Figure 6 provides the algorithm-specific functions for the one third rule protocol as specified in (Charron-Bost and Schiper, 2009). Computing the message to be sent requires a single instruction, given that the message always contains the value of `x`, which is the value currently proposed by the process. The state-transition function is executed at the end of each round, by each

Fig. 5 Reusable verification framework for round-based algorithms.

```
1  chan broadcast[NPROC] = [1] of {message}
2  byte token;
3
4  inline begin_round() {
5    (token == _pid)
6  }
7
8  inline end_round() {
9    token--
10 }
11
12 inline wait_to_receive() {
13   token--;
14   (token == _pid)
15 }
16
17 inline send_to_all(m) {
18   if
19   :: nempty(broadcast[_pid-1]) -> broadcast[_pid-1] ? _
20   :: empty(broadcast[_pid-1]) -> skip
21   fi;
22   broadcast[_pid-1] ! m
23 }
24
25 inline receive(m, id) {
26   broadcast[id] ? <m>
27 }
28
29 proctype Process() {
30   byte i, j, k, l;
31   message msg;
32   do
33   :: begin_round();
34      compute_message(msg);
35      send_to_all(msg);
36      wait_to_receive();
37      state_transition();
38      end_round()
39   od
40 }
```

process. The local values are updated to include all received values during the current round. This is performed in lines 8–16. If the process receives more than two thirds of the messages, then variable `l` will hold the smallest most frequent value and `x` will be updated to that value (line 29). The final `if` statement asserts that all processes make the same decision using a ghost variable.

Algorithm-specific structures and state information are shown in Figure 7. The global message format is declared in lines 3–5 and contains a single value representing the proposal of a process. Lines 7–13 define the local state of each process and line 15 declares one such local state for each of the n processes. Line 16 provides a shorthand notation for each process to access the local state, which is not only practical but also avoids common errors such as a process accessing other processes' values.

Figure 8 shows the `init` process which runs the n processes and calls a system-wide initialization hook named `system_init` and, at every round the `system_every_round` inline function.

Fig. 6 Specific inline functions modeling the one third rule algorithm.

```
inline compute_message(m) {
  m.value = local.x
}

inline state_transition() {
  d_step {
    local.rmcount = 0;
    for(i : 0..(NPROC-1)) {
      if
      :: local.rm[i] ->
         receive(msg, i);
         local.values[i] = msg.value;
         local.rmcount++
      :: else -> skip
      fi
    }
    if
    :: local.rmcount > (2*NPROC/3) ->
       l = 0;
       for(i : 1..(NPROC)) {
         k = 0;
         for(j : 0..(NPROC-1)) {
           if
           :: local.values[j] == i -> k++
           :: else -> skip
           fi
         }
         if
         :: k > l -> local.x = i; l = k
         :: else -> skip
         fi
       }
    :: else -> skip
    fi;
    if
    :: l > (2*NPROC/3) && local.decision == 0 ->
       local.decision = local.x;
       ghost = local.decision;
       assert(local.decision == ghost)
    :: else -> skip
    fi
  }
}
```

Fig. 7 Specific variables and structures modeling the one third rule algorithm.

```
#define NPROC 4

typedef message {
  byte value
}

typedef p_state {
  bool rm[NPROC];
  byte x;
  byte decision;
  byte values[NPROC];
  byte rmcount
}

p_state state[NPROC];
#define local state[_pid-1]
```

Fig. 8 Initialization of processes and round-by-round synchronization.

```
init {
  byte i, j, a = ALPHA;
  bool synchronous = false;
  system_init();
  atomic {
    for(i : 1..(NPROC)) {
      run Process()
    }
  }
  do
  :: (token == 0);
     system_every_round();
     token = NPROC;
     (token == 0);
     token = NPROC
  od
}
```

Fig. 9 Specific functions to initialize processes and update received-message sets.

```
inline system_init() {
  j = 1;
  for(i : 0..(NPROC-1)) {
    state[i].x = j;
    if
    :: j++
    :: skip
    fi
  }
}

#define ALPHA 1

inline system_every_round() {
  if
  :: a == 0 -> synchronous = true
  :: else -> a--
  fi;
  for(i : 0..(NPROC-1)) {
    for(j : 0..(NPROC-1)) {
      if
      :: synchronous || i == j -> state[i].rm[j] = true
      :: else ->
         if
         :: state[i].rm[j] = true
         :: state[i].rm[j] = false
         fi
      fi
    }
  }
}
```

The `system_every_round` inline function is responsible for updating the `rm` sets. It chooses whether process i receives the message from process j during the round that is about to start. Therefore, this inline function is called just before every new round to set the *received messages* set in a non-deterministic way.

It is noteworthy that the `system_every_round` inline function has the potential to generate an immense number of states. For a system with n processes, there are n^2 distinct message deliveries, of which $n \times (n-1)$ might

Table 3 Framework results for the one third rule algorithm.

	Partition symmetry reduction			Framework			
n	States	Memory	Time	States	Memory	Time	Ratio
2	686	0.3 MB	∼0 s	665	0.3 MB	∼0 s	103.2%
3	22499	1.4 MB	∼0 s	29971	1.8 MB	∼0 s	75.1%
4	3.3×10^6	192.7 MB	2 s	5.3×10^6	308.7 MB	3 s	62.3%

be unreliable (depending on the specific communication system being modeled). Therefore, at each round, there are $2^{n\times(n-1)}$ possible combinations for a fully asynchronous system in which all messages might be lost. Consequently, although we mitigate the state-space explosion problem using the proposed reduction techniques, the complexity of the verification process is still exponential in the number of processes.

8.1 Experimental results

Tables 3 and 4 summarize the verification results of using the framework to verify the one third rule algorithm. The framework's code implicitly includes message order reduction, as can be seen in Figure 5, where processes wait on the `token` variable to begin every new round and to receive all messages.

Furthermore, we have collected results of partition symmetry reduction by using the code in Figure 2 at the system initialization code in Figure 9. The code of the `system_init` inline was replaced by the code in Figure 2, with a single modification to substitute `run Process(v)` with `state[i].x = v` to initialize the process values.

Table 3 shows the verification results, obtained using exactly the same hardware and software configuration as the results in the preceding sections. The right hand side of the table provides the framework's results, including message order reduction. The left hand side of the table provides the results of enabling partition symmetry reduction, in addition to message order reduction.

One observation that can be made from these results is that for $n = 4$ the state-space is relatively large. In fact, for $n = 5$ the model checker exhausted the available memory without completing the verification. Another observation is that partition symmetry reduction has a modest improvement to the results. As discussed in the preceding section, communication failures generate an exponential number of states. Hence, the main source of complexity in this model is the combinations of failures.

To further examine the issue of communication failures, Table 4 provides the results of verifying the one third rule protocol with a bounded number of failures. It is feasible to verify systems with $n = 8$ processes and one may observe that partition symmetry reduction has a greater impact for systems composed by a large number of processes. These results, along with the results presented in the preceding sections, substantiate the practical use of the framework and the reductions.

Table 4 Framework results for the one third rule algorithm, with bounded failures.

	Partition symmetry reduction			Framework			
n, f	States	Memory	Time	States	Memory	Time	Ratio
4,1	14541	1.1 MB	∼0 s	22352	1.5 MB	∼0 s	65.1%
4,2	75918	4.6 MB	∼0 s	119621	7.1 MB	∼0 s	63.5%
4,12	3.4×10^6	195.1 MB	2 s	5.4×10^6	312.1 MB	3 s	63.0%
5,1	43208	2.9 MB	∼0 s	96842	6.1 MB	∼0 s	44.6%
5,2	3.8×10^5	23.1 MB	∼0 s	8.6×10^5	56.0 MB	∼0 s	44.2%
5,5	3.4×10^7	2.4 GB	39 s	7.8×10^7	5.5 GB	78 s	43.6%
6,1	1.3×10^5	9.7 MB	∼0 s	3.7×10^5	26.8 MB	∼0 s	35.1%
6,2	1.7×10^6	120.6 MB	∼0 s	4.9×10^6	363.3 MB	3 s	34.7%
6,3	1.6×10^7	1.1 GB	12 s	4.5×10^7	3.4 GB	48 s	35.6%
7,1	3.1×10^5	23.9 MB	∼0 s	1.3×10^6	101.8 MB	1 s	23.8%
7,2	5.7×10^6	427.1 MB	4 s	2.4×10^7	1.9 GB	22 s	23.8%
7,3	7.3×10^7	5.6 GB	83 s	—	—	—	—
8,1	7.4×10^5	60.0 MB	1 s	4.2×10^6	354.1 MB	3 s	17.6%
8,2	1.8×10^7	1.4 GB	17 s	1.1×10^8	8.9 GB	130 s	16.4%

9 Translation to logic-based symbolic model checking

The Spin model checker performs an exhaustive search over all possible execution paths of a given Promela program. An alternative to this approach is to use a logic-based symbolic model checker (Clarke et al., 2001), which does not necessarily explore all the states, but instead uses the logical structure of the formulation to verify if an error can occur in that model.

Over the last decades, SMT and SAT solvers have improved significantly Fichte et al. (2020) and can provide a better performance than exhaustive search. Here, we present an approach for converting Promela models to SMT encodings, so these techniques can be applied to the same problem.

We present a translation from the proposed template for round-based model checking, introduced in Section 8, to an SMT encoding, which can be verified using a SAT-solver. This translation methodology can be automated through a compiler. We show such methodology using the example introduced in Figure 3: the wait-free n-process consensus protocol (Herlihy, 1991), formalized in Promela.

For the sake of readability, Figure 10 shows the translation of the Promela model in Python, using the Z3 bindings (de Moura and Bjørner, 2008). We also support the generation of SMT-LIB2, a common language for SMT solvers.

The key to generating an SMT problem equivalent to the Promela model lies in modeling variables (state) and instructions (state transitions). Variables that are write-only (like v) are converted into an array of symbolic bytes, one for each process. Variables that are set from a `select` constructs have each local symbolic variable limited by their bounds (line 15 limits the possible values of v). This technique is popular in parallelizing compilers (Bondhugula et al., 2008), with the polyhedral model being the most precise (Singh et al., 2017).

Variables that change value during the execution of each process are translated to a matrix of $N*T$ symbolic bytes. N stands for the number of processes, representing the local state, and T (or `max_time`) represents the maximum time

Fig. 10 Wait-free consensus n-process consensus protocol, encoded in Z3.

```
1  n = 3
2  instructions = 3
3  max_time = instructions * n
4
5  s = Solver()
6
7  r = byte("r")
8  ghost = byte("ghost")
9
10 vs = constant_local_byte("v", n)
11 decisions = p_byte("decision", n)
12 firsts = p_byte("first", n)
13
14 # select( v : 1 .. n)
15 restrict_all(s, vs, lambda v: And(v>0, v<=n))
16 restrict_all(s, decisions, lambda d: And(d>0, d<=n))
17 restrict_all(s, firsts, lambda f: And(f>0, f<=n))
18
19 # Body
20 proc =  bitvect_in_time("proc", max_time)
21 ins = bitvect_in_time("proc", max_time)
22
23 restrict_all(s, proc_domain, lambda p: And(p >= 0, p < n))
24 restrict_all(s, ins_domain, lambda p: And(i >= 0, i < instructions))
25
26 for i in range(max_time):
27     for j in range(i):
28         restrict(s, Implies(proc[i] == proc[j], ins[i] > ins[j]))
29
30 for t, cp, ci in zip(range(max_time), proc, ins):
31     """Instruction #0: Compare and Swap"""
32     for p, v, first in zip(range(n), vs, firsts[t]):
33         # If(old r == 0, r = vs, r = old r)
34         restrict(s, If(And(cp == p, ci == 0, r[t] == 0),
35                  r[t+1] == v,
36                  r[t+1] == r[t] ))
37         # new first = old r
38         restrict(s, If(And(cp == p, ci == 0),
39                  firsts[t+1][p] == r[t],
40                  firsts[t+1][p] == first ))
41     """Instruction #1: Decision depends on first and input/vs"""
42     for p, decision, first, v in zip(range(n), decisions[t], firsts[t], vs):
43         restrict(s, If(And(p == cp, ci == 1),
44               decisions[t+1][p] == If(first == 0, v, first),
45               decisions[t+1][p] == decision))
46     """Instruction #2: Ghost can have previous value, or decision value"""
47     for p, decision in enumerate(decisions[t]):
48         restrict(s, If(And(p == cp, ci == 2),
49             ghost[t+1] == decision,
50             ghost[t+1] == ghost[t]))
51 # assertions
52 restrict_not(s,  decisions[max_time],  lambda d: ghost[max_time] == d ))
53 sat_result = s.check()
```

instant, obtained from the serialization of global instructions in each process. As an example, the `decision` variable is converted into a matrix of 3 by 9 bytes (line 11).

Variables that are shared across processes do not require duplication per-process nor per-instant and are represented as a single byte (like `r` or `ghost`).

Encoding the execution of programs corresponds to defining the relationship between matrices of mutable variables (like `decision`). Lines 31-50 encode,

Table 5 Execution time and total memory comparison of Explicit versus Symbolic Model Checking for Herlihy's wait-free consensus protocol, with and without Partition Symmetry Reduction.

	Spin with PSR		Spin without PSR		Z3 with PSR		Z3 without PSR	
n	Memory	Time (s)	Memory	Time (s)	Memory	Time (s)	Memory	Time (s)
2	129 MB	∼0	129 MB	∼0	15 MB	∼0	15 MB	∼ 0
3	129 MB	∼0	129 MB	∼0	20 MB	∼0	20 MB	∼ 0
4	129 MB	∼0	147 MB	∼0	59 MB	13±5	59 MB	13±5
5	132 MB	∼0	1.4 GB	78	829 MB	1187±173	829 MB	1244±309
6	158 MB	1	—	—	—	—	—	—
7	373 MB	12	—	—	—	—	—	—
8	2.5 GB	119	—	—	—	—	—	—
9	16.3 GB	493	—	—	—	—	—	—

using the If SMT primitive, the execution flow in each process. Notice that atomic blocks correspond to a single restriction (Instruction #0 is one example). If a variable v has its value increased by one, the corresponding translation would be `v[t+1] = v[t] + 1`.

The final step is to translate the assertions from Promela to restrictions in SMT. Because we are trying to find if there are values that violate this assertions, we negate the assertion within the SMT. The SMT solver guarantees that there are no values that break the assertion, or gives one such example.

Partition symmetry and message order reductions can be applied in SMT encodings. As an example, we apply partition symmetry reduction to the SMT encoding of this problem by generating all the possible partitions, and restricting the values of variables in the different processes to be equal or different, according to the current partition. All possible partitions are then combined using disjunction and added to the current problem.

Table 5 shows the comparison of Spin vs Z3 on execution time and total memory, unlike in the previous evaluation where we reported the memory used to represent the states. Because Z3 does not have a comparable measure, we report total memory consumption of the process. Because SMT-solving using Z3 is non-deterministic (in the order of strategies applied, not on the result), we executed each instance of the problem 30 times and report the mean and standard deviation. For this experiment, we used a 2.93GHz Intel Xeon X5670 with 24GB of 1066MHz RAM, using Spin 6.5.0 and Z3 4.4.1.

One important conclusion to take is that SMT-solvers have a much more unpredictable behavior than exhaustive model checking. Executing Spin several times yields the same number of states and same peak-memory, with little deviations on execution time. SMT-solvers, on the other hand, can be much faster or slower, depending on the initial random seed. Performance predictability is a preferred factor in developer tools.

Another conclusion can be taken from comparing Spin vs Z3 without the reduction on $n = 4$. Spin is significantly faster than the SMT solver, but requires almost double the memory. This might indicate that Z3 might be a viable choice in memory-constrained scenarios where time is a non-critical factor.

Partition Symmetry Reduction in SMT approaches does not significantly improve performance, especially when compared to the high variability of execution time. The actual benefits depend on the problem at hand and can have more impact when the search-strategy of the SMT-solver is fixed, instead of randomly selected. It is also important to notice that Partition Symmetry Reduction allows the Spin model checker to evaluate an higher number of processes, compared to both exhaustive and symbolic searches.

10 Conclusion

Model checking and other formal techniques have proven extremely useful for verifying distributed algorithms. However, these techniques have two key challenges. In many cases the verification process is computationally intensive and is affected by the state-space explosion problem. Furthermore, there is great manual effort in formalizing the smallest sufficient model to verify the correctness of a given system.

To mitigate the state-space explosion problem, we propose two reduction techniques that avoid expanding equivalent states by exploiting symmetries in computations. Partition symmetry reduction deals with initial nondeterministic values selected by processes and reduces the possible permutations to the possible partitions of processes, thereby assigning a partition to each process instead of an explicit initial value. This reduction is valid for algorithms in which the set of initial values is unordered. The theoretical reduction achieved is significant and is substantiated by the practical results obtained using Spin.

Message order reduction deals with the possible interleaving of message transmissions in distributed systems. Distributed algorithms that work in communication-closed rounds apply a state-transition function, at the end of each round, that consists of operations that are commutative. As such, regardless of the order in which incoming messages are read, the local state at the end of a round is the same, exclusively determined by the failures. Therefore, the proposed reduction only examines one permutation of message transmissions. The reduction achieved by this technique is significant and the practical results obtained with Spin are close to the theoretical values.

The two proposed reductions are limited with respect to the algorithms in which validity is guaranteed. This also occurs with many other reduction techniques and, notably, it is the case of the well known partial order reduction. We advocate using the principle of design for verification, *i.e.*, given that it is reasonably practical to design protocols that fulfill the restrictions of the reductions, one may guarantee at design time that a protocol allows the two reductions to be applied. Namely, the protocol should be based upon unordered sets of values and the state-transition function should consist of commutative operations. This would allow both proposed reductions to be applied and result in a significantly smaller state space by orders of magnitude.

To deal with the challenge of formalizing distributed algorithms while aiming for a small but sufficient model, this paper provides a verification template written for the Spin model checker. The template provides designers with a reusable model that implements round-based communication among distributed processes. A designer of a new algorithm only specifies the local state-transition function and the message-computation function for each round. The remainder of the model can be reused. This verification framework includes also the implementation of the proposed reduction techniques, facilitating the formal verification of distributed algorithms. While this framework targets the Promela language, the proposed methods are general and can be applied to other model checking techniques, such as SMT-backed symbolic model checkers.

Acknowledgements This work is funded by national funds through the FCT - Foundation for Science and Technology, I.P., within the scope of project CISUC - UID/CEC/00326/2020 and by European Social Fund, through the Regional Operational Program Centro 2020, project LASIGE - UIDB/00408/2020, the AESOP project (P2020-31/SI/2017, No. 040004) and through the CMU-Portugal project CAMELOT (POCI-01-0247-FEDER-045915).

References

Aminof B, Rubin S, Stoilkovska I, Widder J, Zuleger F (2018) Parameterized model checking of synchronous distributed algorithms by abstraction. In: International Conference on Verification, Model Checking, and Abstract Interpretation, Springer, pp 1–24

Ben-Or M (1983) Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, NY, USA, PODC '83, p 27–30, DOI 10.1145/800221.806707, URL https://doi.org/10.1145/800221.806707

Bóna M (2002) A walk through combinatorics: an introduction to enumeration and graph theory. World Scientific

Bondhugula U, Hartono A, Ramanujam J, Sadayappan P (2008) A practical automatic polyhedral parallelizer and locality optimizer. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pp 101–113, DOI 10.1145/1375581.1375595, URL https://doi.org/10.1145/1375581.1375595

Bosnacki Dragan DD, Holenderski L (2002) Symmetric spin. International Journal on Software Tools for Technology Transfer 4:92–106, DOI 10.1007/s100090200074, URL http://dx.doi.org/10.1007/s100090200074

Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ (1992) Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2):142–170

Chaouch-Saad M, Charron-Bost B, Merz S (2009) A reduction theorem for the verification of round-based distributed algorithms. In: Bournez O, Potapov I

(eds) Reachability Problems, Lecture Notes in Computer Science, vol 5797, Springer Berlin Heidelberg, pp 93–106, DOI 10.1007/978-3-642-04420-5-10

Charron-Bost B, Schiper A (2009) The heard-of model: computing in distributed systems with benign faults. Distributed Computing 22:49–71, DOI 10.1007/s00446-009-0084-6, URL http://dx.doi.org/10.1007/s00446-009-0084-6

Clarke E, McMillan K, Campos S, Hartonas-Garmhausen V (1996) Symbolic model checking. In: Alur R, Henzinger T (eds) Computer Aided Verification, Lecture Notes in Computer Science, vol 1102, Springer Berlin Heidelberg, pp 419–422, DOI 10.1007/3-540-61474-5-93, URL http://dx.doi.org/10.1007/3-540-61474-5-93

Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Emerson E, Sistla A (eds) Computer Aided Verification, Lecture Notes in Computer Science, vol 1855, Springer Berlin Heidelberg, pp 154–169, DOI 10.1007/10722167-15, URL http://dx.doi.org/10.1007/10722167-15

Clarke EM, Emerson EA, Sistla AP (1986) Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263, DOI 10.1145/5397.5399, URL http://doi.acm.org/10.1145/5397.5399

Clarke EM, Grumberg O, Long DE (1994) Model checking and abstraction. ACM Trans Program Lang Syst 16(5):1512–1542, DOI 10.1145/186025.186051, URL http://doi.acm.org/10.1145/186025.186051

Clarke EM, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1):7–34, DOI 10.1023/A:1011276507260, URL https://doi.org/10.1023/A:1011276507260

Clarke EM, Henzinger TA, Veith H, Bloem R (eds) (2018) Handbook of Model Checking. Springer

Cristian F, Fetzer C (1999) The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems 10(6):642–657

Dean J, Sanjay Ghemawat I Google (2004) Mapreduce: Simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI '04), Usenix

Eisner C, Peled D (2002) Comparing symbolic and explicit model checking of a software system. In: Model Checking Software, Lecture Notes in Computer Science, vol 2318, Springer Berlin Heidelberg, pp 230–239, DOI 10.1007/3-540-46017-9-18, URL http://dx.doi.org/10.1007/3-540-46017-9-18

Elrad T, Francez N (1982) Decomposition of distributed programs into communication-closed layers. Science of Computer Programming 2(3):155 – 173, DOI 10.1016/0167-6423(83)90013-8, URL http://www.sciencedirect.com/science/article/pii/0167642383900138

Emerson E, Sistla A (1996) Symmetry and model checking. Formal Methods in System Design 9:105–131, DOI 10.1007/BF00625970, URL http://dx.doi.org/10.1007/BF00625970

Erdős P (1942) On an elementary proof of some asymptotic formulas in the theory of partitions. Annals of Mathematics pp 437–450

Fichte JK, Hecher M, Szeider S (2020) A time leap challenge for sat-solving. In: Simonis H (ed) Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings, Springer, Lecture Notes in Computer Science, vol 12333, pp 267–285, DOI 10.1007/978-3-030-58475-7_16, URL https://doi.org/10.1007/978-3-030-58475-7_16

Gafni E (1998) Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In: Coan BA, Afek Y (eds) Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998, ACM, pp 143–152, URL http://dl.acm.org/citation.cfm?id=277697

García-Pérez Á, Gotsman A, Meshman Y, Sergey I (2018) Paxos consensus, deconstructed and abstracted. In: European Symposium on Programming, Springer, Cham, pp 912–939

Hardy GH, Ramanujan S (1918) Asymptotic formulaæ in combinatory analysis. Proceedings of the London Mathematical Society 2(1):75–115

Herlihy MP (1991) Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13:124–149

Holzmann GJ (2003) The SPIN Model Checker: primer and reference manual. Addison-Wesley

Hughes C, Hughes T (2003) Parallel and Distributed Programming Using C++, 1st edn. Addison-Wesley, The address

Lynch N (1996) Distributed Algorithms. Morgan Kaufmann, San Francisco, CS, URL http://theory.lcs.mit.edu/tds/distalgs.html

Marić O, Sprenger C, Basin D (2017) Cutoff bounds for consensus algorithms. In: International Conference on Computer Aided Verification, Springer, pp 217–237

Minsky M (1961) Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. Annals of Mathematics 74:437

de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, pp 337–340, DOI 10.1007/978-3-540-78800-3_24, URL https://doi.org/10.1007/978-3-540-78800-3_24

Peled D (1994) Combining partial order reductions with on-the-fly model-checking. In: Dill D (ed) Computer Aided Verification, Lecture Notes in Computer Science, vol 818, Springer Berlin Heidelberg, pp 377–390, DOI 10.1007/3-540-58179-0-69, URL http://dx.doi.org/10.1007/3-540-58179-0-69

Raynal M (2018) Consensus and interactive consistency in synchronous systems prone to process crash failures. In: Fault-Tolerant Message-Passing Distributed Systems, Springer, pp 173–187

Santoro N, Widmayer P (2005) Majority and unanimity in synchronous networks with ubiquitous dynamic faults. In: Pelc A, Raynal M (eds) Structural Information and Communication Complexity, 12th International Col-

loquium, SIROCCO 2005, Mont Saint-Michel, France, May 24-26, 2005, Proceedings, Springer, Lecture Notes in Computer Science, vol 3499, pp 262–276

Singh G, Püschel M, Vechev MT (2017) Fast polyhedra abstract domain. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, pp 46–59, URL http://dl.acm.org/citation.cfm?id=3009885

Srikanth TK, Toueg S (1987) Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. Distrib Comput 2(2):80–94, DOI 10.1007/BF01667080, URL https://doi.org/10.1007/BF01667080

Tsuchiya T, Schiper A (2008) Using bounded model checking to verify consensus algorithms. In: Taubenfeld G (ed) Distributed Computing, Lecture Notes in Computer Science, vol 5218, Springer Berlin Heidelberg, pp 466–480, DOI 10.1007/978-3-540-87779-0-32, URL http://dx.doi.org/10.1007/978-3-540-87779-0-32