Journal of Computer and System Sciences 82 (2016) 310–332

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Synchronous counting and computational algorithm design

Danny Dolev [a], Keijo Heljanko [b], Matti Järvisalo [c], Janne H. Korhonen [c], Christoph Lenzen [d], Joel Rybicki [b,*], Jukka Suomela [b], Siert Wieringa [b]

[a] The Rachel and Selim Benin School of Engineering and Computer Science, Edmond J. Safra Campus, The Hebrew University of Jerusalem, Israel
[b] Helsinki Institute for Information Technology HIIT, Department of Computer Science, Aalto University, Finland
[c] Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland
[d] Department of Algorithms and Complexity, MPI for Informatics, Saarbrücken, Germany

ARTICLE INFO

Article history:
Received 8 January 2015
Received in revised form 1 September 2015
Accepted 11 September 2015
Available online 23 October 2015

Keywords:
Distributed computing
Self-stabilisation
Byzantine fault tolerance
Synthesis
Formal methods
SAT

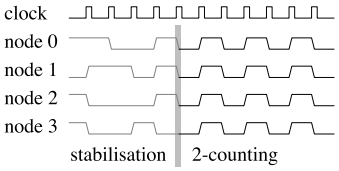
ABSTRACT

Consider a complete communication network on n nodes. In *synchronous* 2*-counting*, the nodes receive a common clock pulse and they have to agree on which pulses are "odd" and which are "even". Furthermore, the solution needs to be *self-stabilising* (reaching correct operation from any initial state) and tolerate f *Byzantine failures* (nodes that send arbitrary misinformation). Prior algorithms either require a source of random bits or a large number of states per node. In this work, we give fast state-optimal deterministic algorithms for the first non-trivial case $f = 1$. To obtain these algorithms, we develop and evaluate two different techniques for algorithm synthesis. Both are based on casting the synthesis problem as a propositional satisfiability (SAT) problem; a direct encoding is efficient for synthesising time-optimal algorithms, while an approach based on counter-example guided abstraction refinement discovers non-optimal algorithms quickly.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

Synchronous counting In the *synchronous C-counting* problem, n nodes have to count clock pulses modulo C. Starting from any initial configuration, the system has to *stabilise* so that all nodes agree on the counter value. Put otherwise, eventually all nodes have to consistently label each clock pulse with values incrementing modulo C.



In this work, we consider a fully-connected synchronous communication network of n nodes with identifiers from the set $\{0, 1, \ldots, n-1\}$. Each node is a finite state machine with s states, and after every state transition, each node *broadcasts* its current state to all other nodes—effectively, each node can see the current states of all other nodes. An algorithm specifies (1) the new state for each observed state, and (2) how to map the internal state of a node to its output.

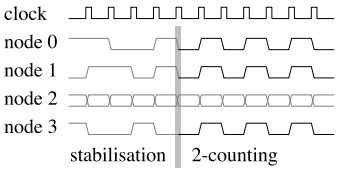
* Corresponding author.
E-mail address: joel.rybicki@aalto.fi (J. Rybicki).

http://dx.doi.org/10.1016/j.jcss.2015.09.002
0022-0000/© 2015 Elsevier Inc. All rights reserved.

Byzantine fault tolerance In a fault-free system, the C-counting problem is trivial to solve. For example, we can designate node 0 as a leader, and then all nodes (including the leader itself) can follow the leader: if the current state of the leader is c, the new state is $c+1 \bmod C$. This algorithm will stabilise in time $t=1$, and we only need $s=C$ different states.

However, we are interested in algorithms that tolerate *Byzantine failures*. Some number f of the nodes may be *faulty*. A faulty node may send arbitrary misinformation to non-faulty nodes, including *different* information to different nodes within the same round. For example, if we have nodes $0,1,2,3$ and node 2 is faulty, node 0 might observe the state vector $(0,1,1,1)$, while node 1 might observe the state vector $(0,1,0,1)$.

Our goal is to design an algorithm with the following guarantee: even if we have up to f faulty nodes, no matter what the faulty nodes do, the system will stabilise so that after t rounds all non-faulty nodes start to count clock pulses consistently modulo C. We will give a formal problem definition in Section 4.



Synchronous counting can be used as a fault-tolerant co-ordination primitive in systems where a synchronous clock signal is available, but the clock pulses have not been labelled in any manner, for example, there is no distinction between even and odd clock pulses. In general, a C-counter can be used as a fault-tolerant *round counter* that assigns explicit round numbers for each clock pulse.

State of the art Both randomised and deterministic algorithms for synchronous counting (often also referred to as *digital clock synchronisation*) have been presented in the literature (see Section 2). However, prior algorithms tend to be expensive to implement in hardware: they require a source of random bits or complicated circuitry.

In this work, we use a single parameter s, the number of states per node, to capture the complexity of an algorithm. If one resorts to randomness, it is possible to solve 2-counting with the trivially optimal number of $s=2$ states—at the cost of a slow stabilisation time (see Sections 2 and 5). However, it is not at all clear whether a small number of states suffices for *deterministic* algorithms.

Contributions We employ *computational* techniques to design deterministic 2-counting algorithms that have the smallest possible number of states. Our contributions are two-fold:

1. we present new algorithms for the synchronous counting problem,
2. we develop new computational techniques for constructing self-stabilising Byzantine fault-tolerant algorithms.

Our focus is on the first non-trivial case of $f=1$. The case of $n=1$ is trivial, and by prior work it is known that there is no algorithm for $1<n<4$. We give a detailed analysis of 2-counting for $n \geq 4$:

- there is no deterministic algorithm for $f=1$ and $n=4$ with $s=2$ states,
- there is a deterministic algorithm for $f=1$ and $n \geq 4$ with $s=3$ states,
- there is a deterministic algorithm for $f=1$ and $n \geq 6$ with $s=2$ states.

Overall, we develop more than a dozen different algorithms with different characteristics, each of which can be also generalised to a larger number of nodes. See Fig. 1 for an overview of the time–space tradeoffs that we achieve with our algorithms.

With very few states per node, our algorithms are easy to implement in hardware. For example, a straightforward implementation of our algorithm for $f=1$, $n=4$, and $s=3$ requires just 2 bits of storage per node, and a lookup table with $3^4=81$ entries. All of our computer-designed algorithms are freely available online [1] in a machine-readable format. While our algorithms are synchronous 2-counters, they can be easily composed to construct synchronous 2^b-counters for any positive integer b (see Section 3 for details).

This work can be seen as a case study of applying synthesis techniques in the area of distributed algorithms. We demonstrate that the synthesis of non-trivial self-stabilising Byzantine fault-tolerant algorithms is indeed possible with the help of modern propositional satisfiability (SAT) solvers [6,26]. We describe two complementary approaches for the synthesis of synchronous 2-counting algorithms and give an empirical comparison of their relative performance:

1. a direct encoding as SAT,
2. a SAT-based counter-example guided abstraction refinement (CEGAR) [13,14] approach.

	$t=$	3	4	5	6	7	8	9
$n=$	4			4	4	3	3	3
	5		3	3	3	3	3	3
	6	3	3	3	2	2	2	2
	7	3	3	3	2	2	2	2
	8	3	2	2	2	2	2	2
	9	3	2	2	2	2	2	2

Fig. 1. Time–space tradeoffs in our computer-designed algorithms. The figure shows s (the number of states) for each combination of n (the number of node) and t (the stabilisation time).

Both approaches make it possible to use modern SAT solvers and to benefit from the steady progress in SAT solver technology. As we will see, the former approach is typically more efficient for tightly-specified problems (e.g., synthesising both space-optimal and time-optimal algorithms), while the latter is more promising for more relaxed problems (e.g., synthesising space-optimal algorithm regardless of the stabilisation time).

Structure Section 2 covers related work and Section 3 discusses applications of synchronous 2-counters. Section 4 gives a formal definition of the problem, and Section 5 gives two examples of human-designed algorithms. Section 6 gives a graph-theoretic interpretation that is helpful in the analysis of counting algorithms. In Section 7 we show that (1) we can increase n for free, without affecting the parameters f, s, or t; this enables us to focus on small values of n, and (2) we can generalise the algorithms to a larger class of network topologies with a slight cost in stabilisation time. Section 8 presents an overview of the use of computers in algorithm design and highlights the new results for synchronous counting. Section 9 describes a direct formulation of the synthesis problem for synchronous counting algorithms as propositional satisfiability. Section 10 describes the SAT-based counter-example guided abstraction refinement synthesis technique. Finally, Section 11 overviews the results of the empirical evaluation of the two different synthesis techniques, suggesting a tradeoff between establishing the existence of any algorithm and finding optimal algorithms.

2. Related work

Randomised algorithms for synchronous counting Randomised algorithms for synchronous 2-counting are known, with different time–space tradeoffs.

The algorithm by Dolev and Welch [23] requires only $s=3$ states, but the expected stabilisation time is $2^{O(n-f)}$. On the other hand, it is possible to attain short stabilisation times using randomisation. For example, the algorithm by Ben-Or et al. [3] stabilises in expected constant time. However, it requires $\Omega(2^f)$ states and private channels (i.e., the adversary has limited information on the system's state).

Deterministic algorithms for synchronous counting The fastest known deterministic algorithm is due to Dolev and Hoch [20], with a stabilisation time of $O(f)$. However, the algorithm is not well suited for a hardware implementation. It uses as a building block several instances of algorithms that solve the Byzantine consensus problem—a non-trivial task in itself. The number of states is also large, as some storage is needed for each Byzantine consensus instance.

Consensus lower bounds for synchronous counting *Binary consensus* is a classical problem that has been studied in the context of Byzantine fault tolerance; see, e.g., the textbook by Lynch [44] for more information. In brief, the problem is defined as follows. Each node has a binary input, and all non-faulty nodes have to produce the same binary output, 0 or 1. If all inputs are equal to 0, the common output has to be 0, and if all inputs are equal to 1, the common output has to be 1; otherwise the common output can be either 0 or 1. It is easy to show that synchronous 2-counting is at least as difficult to solve as binary consensus.

Lemma 1. *If we have a 2-counting algorithm $\mathcal{A}$ that stabilises in time t, we can design an algorithm that solves binary consensus in time t, for the same parameters n and f.*

Proof. Let $\mathbf{x}(0)$ and $\mathbf{x}(1)$ be some configurations that may occur during the correct operation of $\mathcal{A}$ after it has stabilised, so that in configuration $\mathbf{x}(a)$ all nodes output a. More specifically:

- For any $a=0,1$ and $j=0,1,2,\ldots$, if we initialise the system with configuration $\mathbf{x}(a)$ and run $\mathcal{A}$ for j rounds, all non-faulty nodes output $(a+j) \bmod 2$.

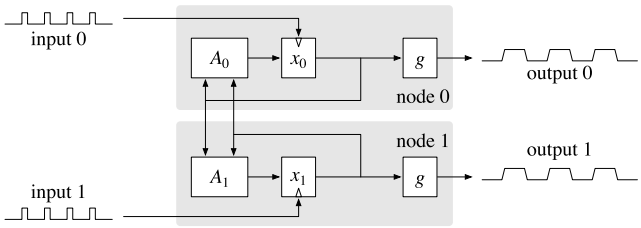


Fig. 2. A 2-counter for $n = 2$, viewed as an electronic circuit.

First assume that t is even. Each node i receives its input a for the binary consensus problem. We use the element i of $\mathbf{x}(a)$ to initialise the state of node i. Then we run $\mathcal{A}$ for t rounds. Finally, the output of algorithm $\mathcal{A}$ forms the output of the binary consensus instance. To see that the algorithm is correct, we make the following observations: (1) All non-faulty nodes produce the same output at time t, regardless of the input. (2) If all inputs had the same value a, we used $\mathbf{x}(a)$ to initialise all nodes, and hence the final output is a.

For an odd t, we can use the same approach if we complement the inputs. In summary, $\mathcal{A}$ can be used to solve binary consensus in time t. □

Now we can invoke the familiar lower bounds related to the consensus problem:

- no algorithm can tolerate $f \geq n/3$ failures [51],
- no deterministic algorithm can solve the problem in $t < f + 1$ rounds [32].

Pulse synchronisation Both 2-counting and *pulse synchronisation* [3,16,19,23] have a superficially similar goal: produce well-separated, (approximately) synchronised clock pulses in a distributed system in a fault-tolerant manner. However, there are also many differences: in pulse synchronisation the task is to construct a clock pulse without any external reference, while in 2-counting we are given a reference clock and we only need to label each clock pulse as "even" or "odd", or put otherwise, construct a clock that ticks at a slower rate. In general, once pulse synchronisation has been solved, a C-counting algorithm can be used to generate explicit round numbers in a fault-tolerant manner. Also the models of computation for the two problems differ—for pulse synchronisation, a relevant model is an asynchronous network with some bounds on propagation delays and clock drifts. For further discussion on this topic, see a recent survey by Dolev et al. [19].

In summary, a 2-counting algorithm does not solve the pulse synchronisation problem, and a pulse synchronisation algorithm does not solve the 2-counting problem. However, if one is designing a distributed system that needs to produce synchronised clock ticks in a fault-tolerant manner, either of the approaches may be applicable.

Computational algorithm design The computational element of our work can be interpreted as a form of *algorithm synthesis*. In synthesis, the task is to algorithmically find an algorithm or a protocol that satisfies a given specification. The idea of synthesising circuits was proposed by e.g. Church [11] already in the 1960s and there exists a vast body of work related to synthesis.

Classic work on model checking [12,45] consider algorithms for synthesis of both shared-memory and message-passing protocols by solving the satisfiability of certain temporal logic formulas. Unfortunately, synthesis of distributed systems is often intractable both in theory and practice—distributed synthesis problems are often either of high complexity or undecidable [30,50,52]. However, despite the hardness of synthesis—or because of it—several techniques have been proposed to make synthesis tractable [29,31,37].

In contrast to applying general synthesis techniques, that is, algorithms for synthesising a general class of problems, combinatorial search algorithms have also been applied to solve specific synthesis problems. For example, SAT solvers have been used for, e.g., circuit synthesis [7,34,35,41,42], synthesis from safety specifications [8], controller synthesis [47], program sketching [54], synthesising sorting networks [10,15,48], and synthesising local graph algorithms [36,53].

3. Applications

Counters as frequency dividers We can visualise a C-counter as an electronic circuit that consists of n components (nodes); see Fig. 2. Each node i has a register x_i that stores its current state—one of the values $0, 1, \ldots, s-1$. There is a logical circuit g that maps the current state to the output, and another logical circuit A_i that maps the current states of all nodes to the new state of node i. At each rising edge of the clock pulse, register x_i is updated.

If the clock pulses are synchronised, regardless of the initial states of the registers, after t clock pulses the system has stabilised so that the outputs are synchronised and they are incremented (modulo C) at each clock pulse.

In particular, if we have an algorithm for 2-counting, it can be used as a *frequency divider*: given synchronous clock pulses at rate 1, it produces synchronous clock pulses at rate 1/2.

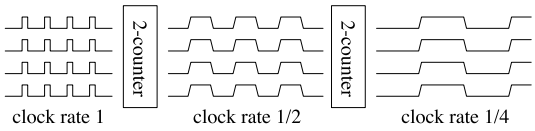


Fig. 3. Composition of 2-counters.

From 2-counters to C-counters Given a 2-counting algorithm, it is also possible to devise C-counters for larger values of $C > 2$. For example, we can compose b layers of 2-counters to build a clock that counts modulo 2^b; see Fig. 3. In a synchronous system, a composition of self-stabilising algorithms is self-stabilising [22]. For the purposes of the analysis, we can wait until layer $i-1$ stabilises, use this as the initial state of layer i, and then argue that the nodes on layer i receive a synchronous clock pulse and hence they will eventually stabilise. In a similar fashion, it is possible to compose two 2^b-counters to attain 2^{2b}-counters, and so on [3].

Moreover, recent work [43] shows how to devise a C-counter for any $C > 1$ by first constructing a suitable $O(f)$-counter. The $O(f)$-counter is used to provide round numbers for a modified consensus protocol. Using the consensus protocol, it is possible to attain a C-counter for any $C > 1$. For the case $f = 1$, the required $O(f)$-counter can be constructed by composing only constantly many 2-counters. Thus, starting from just 2-counters, it is possible to construct C-counters for any $C > 1$.

Counters in mutual exclusion With a C-counter we can implement *mutual exclusion* and *time division multiple access* in a fairly straightforward manner. If we have $C = n$ nodes and one shared resource (e.g., a transmission medium), we can let node i to access the resource when its own counter has value i. Care is needed with the actions of faulty nodes, though—for further information on achieving *fault-tolerant* mutual exclusion, see, e.g., Moscibroda and Oshman [49]. Again 2-counting is of particular interest, as it may be leveraged by more complex mutual exclusion algorithms.

4. Problem formulation

We will now formalise the C-counting problem and the synthesis problem, and introduce the definitions that we will use in this work. Throughout this work, we will follow the convention that nodes, states, and time steps are indexed from 0. We use the notation $[k] = \{0, 1, \ldots, k-1\}$.

Intuitively, the model of computing is as follows. The system consists of a fully-connected message-passing network of n nodes where all nodes have unique identifiers from the set $[n]$. All nodes first *broadcast* their state to all other nodes in the network along the communication links. Moreover, the communication links are labelled so that nodes know from which node a message originated. Thus, after broadcasting, each node receives a vector of messages which the node uses to decide on a new state.

Simplifications As our focus is primarily on 2-counters, we will now fix $C = 2$; the definitions are straightforward to generalise.

In prior work, algorithms have made use of a function that maps the internal state x_i of a node to its output $g(x_i)$. However, in this work we synthesise algorithms that do not need any such mapping: for our positive results, an identity mapping is sufficient, and for the negative result, we study the case of $s = 2$ which never benefits from a mapping. Hence we will now give a formalisation that omits the output mapping.

Algorithms Fix the following parameters:

- $n =$ the number of nodes,
- $f =$ the maximum number of faulty nodes,
- $s =$ the number of internal states.

An algorithm $\mathbf{A}$ specifies a *state transition* function $A_i : [s]^n \to [s]$ for each node $i \in [n]$. Here $[s]^n$ is the set of *observed configurations* of the system.

Projections Let $F \subseteq [n]$, $|F| \leq f$ be the set of *faulty* nodes. We define the *projection* π_F as follows: for any observed configuration $\mathbf{u} \in [s]^n$, let $\pi_F(\mathbf{u})$ be a vector $\mathbf{x}$ such that $x_i = *$ if $i \in F$ and $x_i = u_i$ otherwise. For example,

$$\pi_{\{2,4\}}((0,1,0,1,1)) = (0,1,*,1,*).$$

This gives us the set $V_F = \pi_F([s]^n)$ of *actual configurations*. Two actual configurations are particularly important:

$$\mathbf{0}_F = \pi_F((0,0,\ldots,0)) \quad \text{and} \quad \mathbf{1}_F = \pi_F((1,1,\ldots,1)).$$

Note that since non-faulty nodes do not know the set F, they cannot uniquely determine the actual configuration from any observed configuration.

1. If more than $(n+f)/2$ entries in $\mathbf{u}$ are 0:
   - Switch to state 1.
2. Otherwise, if more than $(n+f)/2$ entries in $\mathbf{u}$ are 1:
   - Switch to state 0.
3. Otherwise:
   - Flip the coin to get a random bit $b \in \{0,1\}$.
   - Switch to state b.

Fig. 4. A randomised 2-counting algorithm. All nodes follow the same algorithm.

Executions Let $\mathbf{x}, \mathbf{y} \in V_F$. We say that configuration $\mathbf{y}$ is *reachable* from $\mathbf{x}$ if for each non-faulty node $i \notin F$ there exists some observed configuration $\mathbf{u}_i \in [s]^n$ satisfying $\pi_F(\mathbf{u}_i) = \mathbf{x}$ and $A_i(\mathbf{u}_i) = y_i$. Intuitively, the faulty nodes can feed such misinformation to node i that it chooses to switch to state y_i. We emphasise that $\mathbf{u}_i$ may be different for each i; the misinformation need not be consistent.

An *execution* of an algorithm $\mathbf{A}$ for given set of faulty nodes F is an infinite sequence of actual configurations $X = (\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \ldots)$ such that $\mathbf{x}^{r+1}$ is reachable from $\mathbf{x}^r$ for all r.

Stabilisation For an execution $X = (\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \ldots)$, define its t-tail

$$X[t] = (\mathbf{x}^t, \mathbf{x}^{t+1}, \mathbf{x}^{t+2}, \ldots).$$

We say that X *stabilises in time* t if one of the following holds:

$$X[t] = (\mathbf{0}_F, \mathbf{1}_F, \mathbf{0}_F, \ldots) \quad \text{or} \quad X[t] = (\mathbf{1}_F, \mathbf{0}_F, \mathbf{1}_F, \ldots).$$

Synchronous counters We say that an algorithm $\mathbf{A}$ *stabilises in time* t if for any set of faulty nodes F with $|F| \leq f$, all executions of $\mathbf{A}$ stabilise in time t. An algorithm $\mathbf{A}$ *solves synchronous 2-counting* if $\mathbf{A}$ stabilises in time t for some finite t; we refer to such algorithms as *2-counting algorithms*.

The synthesis problem Now that we have formally defined what a 2-counting algorithm is, we can give the definition for the synthesis problem of counting algorithms. First, the decision version of the problem is the *realisability problem*. Given an instance (n, f, s, t), the task is to decide whether there exists a 2-counting algorithm for a network with n nodes satisfying the following properties:

1. the algorithm tolerates f failures,
2. each node uses at most s states,
3. the algorithm stabilises in at most t steps.

If such an algorithm exists, we say that the instance (n, f, s, t) is *realisable*. The *synthesis problem* is to output an algorithm $\mathbf{A}$ if the instance is realisable or state that no algorithm exists.

5. Human-designed algorithms

Before moving on to computer-designed algorithms using SAT-based techniques, in this section we illustrate a few human-designed algorithms. First, we show that randomisation helps when it comes to designing small-state (but slow) algorithms. This is followed by a deterministic algorithm that solves the counting problem in the general case with a large number of internal states.

Randomised algorithms We extend our model to randomised algorithms by equipping each node with a *private coin*. Now in a single synchronous round, every node can flip its coin to access one random bit. Thus, node i can decide on its new state using the random bit $b \in \{0,1\}$ and the observed configuration $\mathbf{u} \in [s]^n$. Here we call bit 1 *heads*. In contrast to the randomised algorithm by Dolev and Welch [23], the following algorithm only uses two states.

Let $n \geq 4$, $f < n/3$, and $s = 2$. We can solve the 2-counting problem with the algorithm of Fig. 4.

Lemma 2. *Let p be the probability that out of $n - f - 1$ fair coin flips, more than $(n+f)/2 - 1$ flips have the same value. Then the randomised algorithm solves synchronous 2-counting in $1/p + 1$ rounds in expectation.*

Proof. Observe that no two distinct non-faulty nodes apply rules 1 and 2 during the same round: if a node i sees the value 0 more than $(n+f)/2$ times, then any node j must see value 0 at least $(n-f)/2$ times, and thus, j sees the value 1 fewer than $(n+f)/2$ times. Moreover, if more than $(n+f)/2$ non-faulty nodes have the same output, then the system will stabilise in the next round as all non-faulty nodes switch to the same state.

Next we argue that with probability at least p, more than $(n+f)/2$ non-faulty nodes have the same state. We have three cases. In the first case, at least one non-faulty node applies rule 1. Then in the worst case all other nodes flip their coins, so the system stabilises with probability at least p. The second case, where at least one non-faulty node applies rule 2, is symmetrical. Finally, the third case consists of all nodes flipping their coins simultaneously. In this case, fix the output of a single non-faulty node and repeat the analysis of the previous two cases.

The number of rounds before we stabilise follows a geometric distribution, so in expectation, we get a successful streak of coin flips in $1/p$ rounds and stabilise during the next round. □

Theorem 1. *For all $n \geq 4$ and $f \leq n/3$, the expected stabilisation time of the randomised algorithm is bounded by*

$$\min\{2^{2f+2}+1, 2^{O(f^2/n)}\}.$$

Proof. We bound the probability p in Lemma 2 from which the expected stabilisation time follows.

For the first bound, it suffices to analyse the event where the first $2f+1$ non-faulty nodes and at least half of the remaining non-faulty nodes all flip heads at the same round, as $2f+1-(n-f-2f-1)/2 > (n+f)/2$. Now observe that the probability of $2f+1$ coin flips all being heads is 2^{-2f-1} and the probability that at least half of out of N coin flips are heads is at least $1/2$. Combining these observations gives us the first bound: the probability of the analysed event is at least 2^{-2f-2} and the number of trials for the first success follows a geometric distribution, and thus, the expected number of trials is at most 2^{2f+2}.

For the second bound, if $f = \Theta(n)$ then the second bound trivially follows from the first. Suppose $f = o(n)$. We use the fact [28,46] that for any $t \in [N/8]$

$$\Pr[X \geq N/2 + t] \geq \frac{1}{15}\exp(-16t^2/N),$$

where X is the number of heads in N coin flips. Setting $N = n-f-1$ and $t = \lfloor (n+f)/2 \rfloor + 1 - N/2$ gives us the desired bound. □

Deterministic algorithms We can leverage existing deterministic algorithms for binary consensus to come up with synchronous counting algorithms. However, this leads to a large number of states per node.

For example, this theorem follows from the results by Dolev and Hoch [20]:

Theorem 2. *Let* **A** *be a deterministic algorithm that solves binary consensus in R rounds for n nodes and f faults. Then there exists a deterministic algorithm* **B** *that solves synchronous C-counting in time $t \in O(R+C)$ for n nodes and f faults.*

Now we can use any consensus algorithm, such as the phase king algorithm [4], to get a synchronous counter. The phase king achieves optimal resilience and has $O(f)$ stabilisation time and uses $O(\log f)$ state bits (for keeping track of the current round number) per node. However, the resulting synchronous counter relies on executing $O(f)$ consensus instances in parallel, which yields a very large state space. We get the following corollary:

Corollary 1. *For all $n \geq 4$, $f < n/3$ and $C \geq 2$, there is a deterministic C-counting algorithm that stabilises in $t \in O(C+f)$ rounds and uses $s \in 2^{O(\log C + f \log f)}$ states.*

This approach is not very attractive, for example, from the perspective of hardware implementations. For further discussion on human-designed algorithms, see a recent survey [19] on the topic. We will now turn our attention to efficient, deterministic, computer-designed algorithms.

6. Projection graphs

Before discussing how to *find* an algorithm (or prove that an algorithm does not exist), let us first explain how we can *verify* that a given algorithm is correct. Here the concept of a *projection graph* is helpful—see Fig. 10 in the appendix for an example.

Fix the parameters s, n, and f, and consider a candidate algorithm **A** that is supposed to solve the 2-counting problem. For each set $F \subseteq [n]$ of faulty nodes, construct the directed graph $G_F(\mathbf{A}) = (V_F, R_F(\mathbf{A}))$ as follows.

1. The set of nodes V_F is the set of actual configurations.
2. There is an edge $(\mathbf{u}, \mathbf{v}) \in R_F(\mathbf{A})$ if configuration $\mathbf{v} \in V_F$ is reachable from configuration $\mathbf{u} \in V_F$. In general, this may produce self-loops.

Note that the outdegree of each node in $G_F(\mathbf{A})$ is at least 1. Directed walks in $G_F(\mathbf{A})$ correspond to possible executions of algorithm **A**, for this set F of faulty nodes. To verify the correctness of algorithm **A**, it is sufficient to analyse the projection graphs G_F. The following lemmas are straightforward consequences of the definitions.

Lemma 3. *Algorithm* **A** *stabilises in some time* t *iff for every* F*, graph* $G_F(\mathbf{A})$ *contains exactly one directed cycle,* $\mathbf{0}_F \mapsto \mathbf{1}_F \mapsto \mathbf{0}_F$.

Lemma 4. *Algorithm* **A** *stabilises in time* t *iff the following holds for all* F*:*

1. *In* $G_F(\mathbf{A})$*, the only successor of* $\mathbf{0}_F$ *is* $\mathbf{1}_F$ *and vice versa.*
2. *In* $G_F(\mathbf{A})$*, every directed walk of length* t *reaches node* $\mathbf{0}_F$ *or* $\mathbf{1}_F$.

Lemma 5. *Let* **A** *be an algorithm. Consider any four configurations* $\mathbf{x}, \mathbf{u}, \mathbf{v}, \mathbf{w} \in V_F$ *with the following properties:* $(\mathbf{x}, \mathbf{u}) \in R_F(\mathbf{A})$*,* $(\mathbf{x}, \mathbf{v}) \in R_F(\mathbf{A})$*, and* $w_i \in \{u_i, v_i\}$ *for each* $i \notin F$*. Then* $(\mathbf{x}, \mathbf{w}) \in R_F(\mathbf{A})$.

7. Increasing the number of nodes

It is not obvious how to use computational techniques to design an algorithm that solves the 2-counting problem for a fixed $f = 1$ but arbitrary $n \geq 4$. However, as we will show next, we can generalise any algorithm so that it solves the same problem for a larger number of nodes, without any penalty in time or space complexity. Therefore it is sufficient to design an algorithm for the special case of $f = 1$ and $n = 4$. From the perspective of parametrised verification and synthesis, the following lemma can be regarded as a *cut-off* result [27,37].

Lemma 6. *Fix* $n \geq 4$*,* $f < n/2$*,* $s \geq 2$*, and* $t \geq 1$*. Assume that* **A** *is an algorithm that solves the* 2*-counting problem for* n *nodes, out of which at most* f *are faulty, with stabilisation time* t *and with* s *states per node. Then we can design an algorithm* **B** *that solves the* 2*-counting problem for* $n + 1$ *nodes, out of which at most* f *are faulty, with stabilisation time* t *and with* s *states per node.*

Proof. The claim would be straightforward if we permitted the stabilisation time of $t + 1$. However, some care is needed to avoid the loss of one round.

We take the following approach. Let p be a projection that removes the last element from a vector, for example, $p((a, b, c)) = (a, b)$. In algorithm **B**, nodes $i \in [n]$ simply follow algorithm **A**, ignoring node n:

$$B_i(\mathbf{u}_i) = A_i(p(\mathbf{u}_i)).$$

Node n tries to predict the majority of nodes $0, 1, \ldots, n-1$, i.e., what most of them are going to output after this round:

- Assume that node n observes a configuration $\mathbf{u}_n$. For each $i \in [n]$, define $h_i = A_i(p(\mathbf{u}_n))$. If a majority of the values h_i is 1, then the new state of node n is also 1; otherwise it is 0.

To prove that the algorithm is correct, fix a set $F \subseteq [n+1]$ of faulty nodes, with $|F| \leq f$. Clearly, all nodes in $[n] \setminus F$ will start counting correctly at the latest in round t. Hence any execution of **B** with $n \in F$ trivially stabilises within t rounds; so we focus on the case of $F \subseteq [n]$, and merely need to show that also node n counts correctly.

Fix an execution $X = (\mathbf{x}^0, \mathbf{x}^1, \ldots)$ of **A**, and a time step $r \geq t$. Consider the state vector $\mathbf{x}^{r-1}$. By assumption, **A** stabilises in time t. Hence the successors of $\mathbf{x}^{r-1}$ in the projection graph must be in $\{\mathbf{0}_F, \mathbf{1}_F\}$.

The key observation is that only one of the configurations $\mathbf{0}_F$ and $\mathbf{1}_F$ can be the successor of $\mathbf{x}^{r-1}$. Otherwise Lemma 5 would allow us to construct another state that is a successor of $\mathbf{x}^{r-1}$, contradicting the assumption that **A** stabilises.

We conclude that for all rounds $r \geq t$ and all nodes $i \in [n] \setminus F$, the value h_i is independent of the states communicated by nodes in F. Since the values h_i are identical and $n - f > f$, node n attains the same state as other correct nodes in rounds $r \geq t$. □

Other network topologies Recall that our basic definitions only consider algorithms that operate in fully-connected networks, that is, the topology of the communication network is a complete graph. Next we show that it is relatively straightforward to generalise our small-state algorithms to other network topologies as well—albeit with a slight increase in the stabilisation time. The idea is to have a small core of nodes to initially solve synchronous counting, and from thereon, propagate the solution throughout the network. This approach was originally introduced by Braud-Santoni et al. [9]. We now show how this idea can be applied in a large class of graphs.

Consider the following families of graphs $\mathcal{G}(k, m, d)$ for integers $k, m, d > 0$. Let $G = (V, E)$ be a graph. We say $G \in \mathcal{G}(k, m, d)$ if there exists a partition $V_0, \ldots, V_d$ of the nodes V such that

1. V_0 is a k-clique.
2. Each node $i \in V_a$ has at least m neighbours in $V_0 \cup \cdots \cup V_{a-1}$.

Put otherwise, we can characterise $\mathcal{G}(k, m, d)$ using the following game (which is reminiscent of threshold models in the context of influence spreading in social networks). Initially, colour all vertices of graph G white. We pick a clique of k nodes and colour all the nodes black. Now any node with at least m black neighbours switches its own colour black. If after d iterations all nodes are coloured black, then $G \in \mathcal{G}(k, m, d)$. See Fig. 5 for examples.

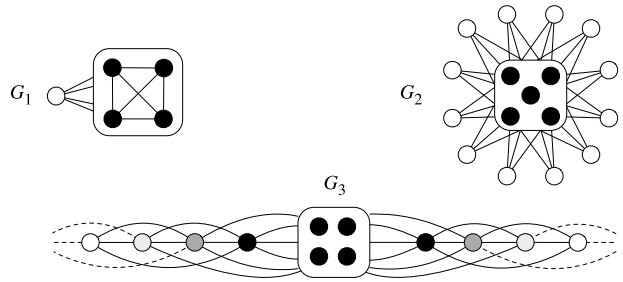


Fig. 5. Examples of generalised network topologies. Nodes encompassed within a rectangle form a clique from which the stabilisation propagates throughout the network. Here, $G_1 \in \mathcal{G}(4,3,1)$ and $G_2 \in \mathcal{G}(5,3,1)$. The partially illustrated graph $G_3 \in \mathcal{G}(4,3,k)$ is a cycle where there are additional edges to all neighbours within distance 3.

Lemma 7. *Assume* **A** *is an algorithm that solves synchronous 2-counting in a complete network of n nodes, out of which at most f are faulty, with stabilisation time t and with s states per node. Then for any $G \in \mathcal{G}(n, 2f+1, d)$, we can design an algorithm* **B** *that solves the synchronous 2-counting in G using s states per node. Moreover,* **B** *tolerates f failures and stabilises in time $t + d - 1$.*

Proof. Let $G \in \mathcal{G}(n, 2f+1, d)$ be our network topology. Fix a partition $V_0, \dots, V_d$ where $V_0 = \{1, \dots, n\}$ is a n-clique. We construct an algorithm **B** using the following rules:

1. If $i \in V_0 = K$, then i outputs $A_i(x_1, \dots, x_n)$.
2. If $i \in V_a$ for some $a > 0$, then node i follows the majority of neighbours in $V_0 \cup \dots \cup V_{a-1}$. If the majority has output y, then output $1 - y$. Otherwise output the current state.

We argue that at time step $t + r$, all nodes in $V_0 \cup \dots \cup V_{r+1}$ have stabilised. The case of $r = 0$ follows from Lemma 6. Suppose the claim holds for some r' and consider node $i \in V_{r'+2}$. By the induction assumption and definition of G, i has a set $P \subseteq V_0 \cup \dots \cup V_{r'+1}$ of at least $2f + 1$ neighbours.

Now node i sees a majority of more than $f + 1$ nodes in P having the same output y. Thus node i outputs $1 - y$ and is in agreement with non-faulty nodes in P in the next round. Since there are $d + 1$ sets in the partition of V, the algorithm stabilises in $t + d - 1$ steps. □

It is known that consensus cannot be solved in networks with vertex-connectivity less than $2f + 1$ [18], and by Lemma 1, this result carries over to synchronous 2-counting.

Beyond synchronous counting We note that the previous lemmas hold for a larger class of problems as well: if it suffices that a node v simply follows a majority of its neighbours, the generalisation techniques can be applied. These problems include, for example, binary consensus and set agreement [9].

8. Computer-designed algorithms

In principle, we could now attempt to use a computer to tackle our original problem. By the discussion of Section 7, it suffices to discover an algorithm with the smallest possible s for the special case of $n = 4$ and $f = 1$. We could try increasing values of $s = 2, 3, \dots$. Once we have fixed n, f, and s, the problem becomes finite: an algorithm is a lookup table with $\ell = ns^n$ entries, and hence there are s^ℓ candidate algorithms to explore. For each candidate algorithm, we could use the projection graph approach of Section 6 to quickly reject any invalid algorithm.

Unfortunately, the search space grows very rapidly and super-exponentially in the parameters n, s, and f. As we will see, there is no algorithm with $n = 4$ and $s = 2$. For $n = 4$ and $s = 3$, we have approximately 10^{154} candidates. We use three complementary approaches to tackle the task.

1. Reduce (encode) the problem directly to propositional satisfiability and apply SAT solvers.
2. Instead of directly encoding the problem as SAT, apply a SAT-based iterative counter-example guided abstraction refinement approach, in hope of better coping with the inherent combinatorial explosion.
3. Narrow down the search space by also considering restricted classes of algorithms.

The first approach is discussed in Section 9 and the second approach in Section 10. We will now describe the third approach, restricting the class of algorithms.

Table 1
Summary of computer-designed algorithms. The number of nodes n is the smallest network on which the algorithm works and t is the worst-case stabilisation time.

class	nodes (n)	states (s)	stabilisation time (t)
cyclic	4	3	7
	5	3	6
	6	3	3
	7	2	8
	8	2	4
general	4	4	5
	5	3	4
	6	2	6

Cyclic algorithms We will consider two classes of algorithms—general algorithms (without any restrictions) and *cyclic* algorithms. We say that algorithm **A** is cyclic if

$$A_i((x_i, x_{i+1}, \ldots, x_{n-1}, x_0, x_1, \ldots, x_{i-1})) = A_0((x_0, x_1, \ldots, x_{n-1}))$$

for all i and all **x**. That is, a cyclic algorithm is invariant under cyclic renaming of the nodes.

There is no a priori reason to expect that the most efficient algorithms are cyclic. However, cyclic algorithms have many attractive features: for example, in a hardware implementation of a cyclic algorithm we only need to take n copies of identical modules. Furthermore, the search space is considerably smaller: we only need to define transition function A_0. For $n = 4$ and $s = 3$, we have approximately 10^{38} candidate algorithms.

Cyclic algorithms are also much easier to verify. The projection graphs $G_F(\mathbf{A})$ are isomorphic for all $|F| = 1$ and hence it is sufficient to check one of them.

Results We now present our main results on the new computer-generated algorithms and refer the discussion on how the results were obtained to Sections 9 and 10.

The positive results are reported in Table 1. The key findings are a cyclic algorithm for $s = 3$, $n = 4$, and $f = 1$, and a non-cyclic algorithm for $s = 2$, $n = 6$, and $f = 1$. The table also gives examples of space-time tradeoffs: we can often obtain faster stabilisation if we use a larger number of states.

For the sake of comparison, we note that the *fastest* deterministic algorithm from prior work [20] stabilises in time $t = 13$ for $f = 1$ and it requires a large state space. Our algorithms achieve the stabilisation time of $t = 5$ for $s = 4$ and $t = 7$ for $s = 3$.

Machine-readable versions of all positive results, together with a Python script that can be used to verify the correctness of the algorithms, are freely available online [1]. Selected examples of the algorithms are also given in Appendix A. We also provide a compact, computer-checkable proof that shows that there is no algorithm for $s = 2$, $n = 4$, and $f = 1$, together with a verification program [1].

9. Synthesis via directly encoding to SAT

In this section, we describe how to directly encode the synthesis problem into SAT. At a high level, we take the following approach:

1. Fix the parameters s, n, f, t, and the algorithm family (cyclic or general).
2. Construct a propositional formula φ that is satisfiable iff an algorithm **A** for the given parameters exists.
3. Use SAT solvers to find a satisfying assignment **a** of φ.
4. Translate **a** to an algorithm **A**.

In essence, the formula φ encodes the conditions given in Lemma 4 and the SAT solver (implicitly) searches through all algorithms **A**:

1. Guess an algorithm **A** and construct the projection graph $G_F(\mathbf{A})$.
2. Verify that there are no self-loops in G_F.
3. Verify that the only successor of $\mathbf{0}_F$ is $\mathbf{1}_F$ and vice versa.
4. For each $d = 1, 2, \ldots, t$, find the subset $B_F(d) \subseteq V_F$ of configurations with the following property: for each $\mathbf{x} \in B_F(d)$ there is a directed walk of length d in G_F that starts from **x** and does not traverse $\mathbf{0}_F$ or $\mathbf{1}_F$. We say that $\mathbf{x} \in B_F(d)$ is a *d-bad* configuration.
5. Verify that the set $B_F(t)$ is empty.

For cyclic algorithms, we identify equivalent transitions and add corresponding equivalence constraints into the formula.

In the following, we describe the encoding by giving constraints for a single set $F \subseteq [n]$ of faulty nodes. The final formula is then the conjunction of these constraints over every possible choice of faulty nodes F.

Variables Fix $F \subseteq [n]$ and let $\mathbf{u} \in [s]^n$, $\mathbf{x}, \mathbf{y} \in V_F$, $i \in [n]$, $d \in [t]$, and $c \in [s]$. We will use the following variables in the encoding:

- $a(\mathbf{u}, i, c)$ is true if $A_i(\mathbf{u}) = c$,
- $h(\mathbf{x}, i, c)$ is true if the adversary can force node i to switch to state c from configuration $\mathbf{x}$,
- $e(\mathbf{x}, \mathbf{y})$ is true if there exists an edge $(\mathbf{x}, \mathbf{y}) \in R_F$,
- $b(\mathbf{x}, d)$ is true if the configuration $\mathbf{x} \in B_F(d)$.

Transition functions The a-variables describe the algorithm, that is, the transition function A_i for each node i. Since we want each A_i to be a well-defined function, we enforce the following constraints for all $\mathbf{u} \in [s]^n$, $i \in [n]$:

$$\bigvee_{c \in [s]} a(\mathbf{u}, i, c) \tag{1}$$

and, for all $c \in [s]$,

$$a(\mathbf{u}, i, c) \to \Big(\bigwedge_{c' \in [s] \setminus c} \neg a(\mathbf{u}, i, c') \Big). \tag{2}$$

Observe that if the constraints given in (2) are omitted, then A_i may be a relation: a node may have several possible state transitions from a given observed state. Although one could always post-process each A_i into a function, allowing transition relations instead of function will only help the adversary.

Projections Let $\mathbf{x}, \mathbf{y} \in V_F$ be configurations. Recall from Section 4 the definition of reachability. If the actual configuration is $\mathbf{x}$, then the adversary can choose any observed configuration from the set

$$U(\mathbf{x}) = \{\mathbf{u} \in [s]^n : \pi_F(\mathbf{u}) = \mathbf{x}\}$$

for each non-faulty node. For all $\mathbf{u} \in U(\mathbf{x})$, we have

$$a(\mathbf{u}, i, c) \to h(\mathbf{x}, i, c), \tag{3}$$

declaring that the adversary can force node i to switch to state c from configuration $\mathbf{x}$. Now, the h-variables imply edges in the projection graph G_F:

$$\bigwedge_{i \in [n] \setminus F} h(\mathbf{x}, i, y_i) \to e(\mathbf{x}, \mathbf{y}). \tag{4}$$

Ensuring counting behaviour The goal of the algorithm is to eventually stabilise and start oscillating only between the two actual configurations $\mathbf{0}_F$ and $\mathbf{1}_F$. To enforce this, we have the clauses

$$e(\mathbf{0}_F, \mathbf{1}_F) \text{ and } e(\mathbf{1}_F, \mathbf{0}_F) \tag{5}$$

together with

$$\neg e(\mathbf{0}_F, \mathbf{x}) \text{ and } \neg e(\mathbf{1}_F, \mathbf{x}) \tag{6}$$

for all $\mathbf{x} \in V_F \setminus \{\mathbf{0}_F, \mathbf{1}_F\}$.

Forbidding non-stabilising walks First, we forbid self-loops in the projection graphs with the unit clause

$$\neg e(\mathbf{x}, \mathbf{x}) \tag{7}$$

for every $\mathbf{x} \in V_F$. To ensure that all configurations but $\mathbf{0}_F$ and $\mathbf{1}_F$ belong to the set $B_F(0)$, we have the clauses

$$\neg b(\mathbf{0}_F, 0) \text{ and } \neg b(\mathbf{1}_F, 0), \tag{8}$$

and, for each $\mathbf{x} \in V_F \setminus \{\mathbf{0}_F, \mathbf{1}_F\}$, the clause

$$b(\mathbf{x}, 0). \tag{9}$$

If a configuration $\mathbf{x}$ can reach a d-bad configuration $\mathbf{y} \in B_F(d)$, then $\mathbf{x}$ must be $(d+1)$-bad. This is captured by the clause

$$\big(e(\mathbf{x}, \mathbf{y}) \wedge b(\mathbf{y}, d)\big) \to b(\mathbf{x}, d+1) \tag{10}$$

for each $\mathbf{x}, \mathbf{y} \in V_F$. Finally, in order for the algorithm to eventually stabilise in the time limit t, we require that there are no t-bad configurations:

$$\neg b(\mathbf{x}, t). \tag{11}$$

Extension: non-uniform stabilisation time It is straightforward to generalise the approach to non-uniform stabilisation time as follows, for some $t_0 < t$:

- if $|F| = 0$, the algorithm stabilises in time t_0,
- if $|F| = 1$, the algorithm stabilises in time t.

This means that in executions where there are no Byzantine failures, we require the synthesised algorithm **A** to stabilise faster. Put otherwise, we constrain the projection graph $G_\emptyset(\mathbf{A})$ so that all directed walks of length $t_0 < t$ reach state **0** or **1**. This is done by simply adding the previously described constraints for all $F \subseteq [n]$, but the stabilisation time bound used for the case $|F| = 0$ is t_0 instead of t. These additional constraints can potentially help with the synthesis, by making the search space smaller, and it also helps with the quality of the algorithms.

Many of our algorithms are synthesised with this kind of encoding, with $t_0 = 2$ or $t_0 = 3$. Hence they not only work correctly in the presence of a Byzantine failure, but they also stabilise very quickly if all nodes are non-faulty. See the online supplement [1] for details.

10. SAT-based counter-example guided search

We now describe an alternative approach for synthesising synchronous counting algorithms: a counter-example guided search algorithm. The structure of our algorithm is similar to counter-example guided abstraction refinement techniques for model checking [13,14] which have previously been successfully applied in various other computationally hard problem domains [2,17,24,29,33,38–40,55]. We repeatedly (1) try to construct an algorithm, (2) check whether the algorithm is correct, and (3) if not, then refine the encoding.

On a high-level, the search algorithm tries to guess a synchronous counting algorithm **A** and then uses a SAT solver to find a *counter-example*, an execution that does not stabilise, for **A**. If one is found, then the counter-example is used to include additional constraints to prune the search space, that is, to rule out at least the found counter-example from the implicit set of remaining algorithm candidates. Otherwise, **A** must be a correct algorithm.

10.1. Encoding

For this approach, we use a symbolic encoding reminiscent of SAT-encodings for bounded model checking [5]. As we want the SAT solver to verify that no counter-examples exist, we use an encoding where the SAT solver finds (i) a set F of faulty nodes and (ii) a bad execution under F for the counting algorithm.

Variables Unlike previously, here we use a bit-wise encoding for the states. Each node has $B = \log(s)$ bits that represent its state. Here an observed configuration $\mathbf{u}$ is represented as a bit string of length nB; each node has B bits to encode its state in $[s]$. If s is not a power of two, then we add extra constraints that only allow s states to be used.

We now list the variables used in the encoding and their semantics:

- $p(i)$ is true if node i is faulty. In other words, $p(i) = 1$ implies $i \in F$.
- $a(\mathbf{u}, i, b)$ represents the bth bit of the next state of node i when it observes the configuration $\mathbf{u} \in \{0,1\}^{nB}$.
- $u(i, j, b, k)$ is the bth bit of node i as observed by node j at time step k.
- $z(k)$ and $o(k)$ are true if all non-faulty nodes are in state 0 or 1, respectively, at time step k.
- $z(i, k)$ and $o(i, k)$ are true if i is faulty *or* it is in state 0 or 1, respectively, at time step k.

We will also use the short-hand $g(i, b, k) = u(i, i, b, k)$ to represent the bth bit of node i at time step k. Next we define each part of the encoding as a separate formula.

Choosing the set of faulty nodes We now define the subformula ψ_{faulty}. We want the solver to be able to guess a set F of faulty nodes under which a counter-example exists. To achieve this, we add constraints that force *exactly* f of the $p(i)$ variables to be true.

In the following let $k \in [f]$, $i \in [n]$ and $j \in [n] \setminus \{0\}$. We will introduce the following variables:

- $p_=(k, i)$ is true if the kth faulty node is i.
- $p_\leq(k, i)$ is true if the kth faulty node is at most i.

To enforce the semantics of these variables, we use the following clauses:

$$p_{=}(k,i) \rightarrow p_{\leq}(k,i), \tag{12}$$

$$p_{\leq}(k,j) \rightarrow (p_{=}(k,j) \vee p_{\leq}(k,j-1)), \tag{13}$$

$$p_{\leq}(k,j-1) \rightarrow \big(p_{\leq}(k,j) \wedge \neg p_{=}(k,j)\big), \tag{14}$$

$$\neg p_{=}(k,0) \rightarrow \neg p_{\leq}(k,0). \tag{15}$$

To ensure that exactly f faulty nodes will be chosen, we use the following clauses: we enforce that at least one node is designated as the kth faulty node with

$$p_{\leq}(k,n-1), \tag{16}$$

and we enforce that there is a strict ordering among the nodes with

$$p_{=}(h-1,i) \rightarrow \neg p_{\leq}(h,i) \tag{17}$$

for all $h \in [f] \setminus \{0\}$. Finally, we establish the correspondence to $p(i)$ variables by enforcing

$$\big(p_{=}(k,i) \rightarrow p(i)\big) \wedge \big(p(i) \rightarrow \bigvee_{k' \in [f]} p_{=}(k',i)\big). \tag{18}$$

Trivial transitions Next, we give clauses that fix the trivial transitions for synchronous counting. The conjunction of these clauses is denoted as ψ_{trivial}.

Let $\mathbf{0}$ and $\mathbf{1}$ correspond to the observed configuration where all nodes are in state 0 or state 1, respectively. The state $0 \in [s]$ is encoded by a bit-string with all zeros, whereas $1 \in [s]$ is encoded as the 0th bit set to one and all other bits zero. Now, for all $i \in [n]$ and $b \in [B] \setminus \{0\}$, we enforce

$$a(\mathbf{0},i,0) \quad \text{and} \quad \neg a(\mathbf{0},i,b), \tag{19}$$

declaring that after observing configuration $\mathbf{0}$, node i must change its state to $1 \in [s]$. Conversely, from configuration $\mathbf{1}$ we need to transition to state 0. Thus, for all $b \in [B]$ we have

$$\neg a(\mathbf{1},i,b). \tag{20}$$

Representing state transitions Let $k \in [t]$. We now define the subformula $\psi_{k,\text{state}}$ encoding the systems behaviour at time step k.

If node i is non-faulty, then the state of node i is observed correctly by all other nodes. This is enforced with

$$\neg p(i) \rightarrow \big(u(i,j,b,k) \leftrightarrow g(i,b,k)\big) \tag{21}$$

for all $i,j \in [n]$ and $b \in [B]$.

For every observable configuration $\mathbf{w} \in [s]^n$, we introduce an auxiliary variable $d(\mathbf{w},i,k)$ representing that node i observes $\mathbf{w}$ at timepoint k. Let $w(i,b)$ denote the bth bit of the binary representation of the state of node i in the observed configuration $\mathbf{w}$.

To enforce the semantics of $d(\mathbf{w},i,k)$, for every observable configuration $\mathbf{w} \in [s]^n$ and every $j \in [n]$ the following constraint is needed:

$$\neg d(\mathbf{w},j,k) \rightarrow \left(\left(\bigvee_{i\in[n],\, b\in[B]:\ w(i,b)=0} u(i,j,b,k)\right) \vee \left(\bigvee_{i\in[n],\, b\in[B]:\ w(i,b)=1} \neg u(i,j,b,k)\right)\right). \tag{22}$$

The intuition behind (22) is that, if $d(\mathbf{w},j,k)$ is false, then there must be at least one bit in the bit representation of the state observed by node j at timepoint k that is unequal to the bit representation of $\mathbf{w}$.

Finally, the state transitions of the system are enforced by the clauses

$$d(\mathbf{w},i,k-1) \rightarrow \big(g(i,b,k) \leftrightarrow a(\mathbf{w},i,b)\big), \tag{23}$$

where $k > 0$, $\mathbf{w} \in [s]^n$, $i \in [n]$ and $b \in [B]$. Equation 23 enforces that if at the previous timepoint we observed state $\mathbf{w}$, then the state of node i equals the successor state of $\mathbf{w}$ as specified by the transition relation of node i.

Indicators for stabilisation Finally, we define the behaviour of the z- and o-variables; the conjunction of these clauses is the subformula $\psi_{k,\text{indicator}}$. Recall that at timepoint k, the variable $z(k)$ is true iff the actual configuration is $\mathbf{0}_F$, and respectively $o(k)$ is true iff the actual configuration is $\mathbf{1}_F$. The equivalence is given by clauses which enforce for all $i \in [n], k \in [t]$:

$$z(k) \to z(i,k) \quad \text{and} \quad o(k) \to o(i,k), \tag{24}$$

together with

$$\neg z(k) \to \bigvee_{j\in[n]} \neg z(j,k) \quad \text{and} \quad \neg o(k) \to \bigvee_{j\in[n]} \neg o(j,k). \tag{25}$$

It remains to describe the clauses that force the semantics of $z(i,k)$ and $o(i,k)$ variables. First, if a node i is faulty then both $z(i,k)$ and $o(i,k)$ are forced to true:

$$p(i) \to \big(z(i,k) \wedge o(i,k)\big). \tag{26}$$

For the z-variables, we enforce for all $b \in [B]$ the clauses

$$z(i,k) \to \big(p(i) \vee \neg g(i,b,k)\big) \tag{27}$$

and the disjunction

$$\neg z(i,k) \to \bigvee_{b\in[B]} g(i,b,k), \tag{28}$$

declaring that $z(i,k)$ is true iff i is faulty or in state $0 \in [s]$. Similarly for the o-variables, as state $1 \in [s]$ was encoded as the bit string $10\ldots0$, we declare the following clauses to constrain the o-variables:

$$o(i,k) \to \Big(p(i) \vee \big(g(i,0,k) \wedge \bigwedge_{b\in[B]\setminus\{0\}} \neg g(i,b,k)\big)\Big) \tag{29}$$

together with the disjunction

$$\neg o(i,k) \to \big(\neg g(i,0,k) \vee \bigvee_{b\in[B]\setminus\{0\}} g(i,b,k)\big). \tag{30}$$

Combining the subformulas The counter-example guided search algorithm incrementally builds a propositional formula to use for both verification and synthesis. In the algorithm description, we will refer to the following formulas:

$$\psi_{\text{base}} = \psi_{\text{faulty}} \wedge \psi_{\text{trivial}}, \tag{31}$$

which gives the basis of the encoding, and, for each $k \geq 0$,

$$\tau_k = \psi_{k,\text{state}} \wedge \psi_{k,\text{indicator}}, \tag{32}$$

which encodes the unrolling of time.

10.2. Basic search algorithm

Our search algorithm will iteratively construct a sequence $\Psi_0, \Psi_1, \ldots$ of formulas until a stabilising 2-counting algorithm is found. Given a satisfiable formula Ψ_i, a satisfying assignment ρ defines the following:

- $\mathbf{A}(\rho)$: an algorithm defining the n transition functions $A_1, \ldots, A_n$,
- $F(\rho) \subseteq [n]$: a set of f faulty nodes,
- $X(\rho) = (\mathbf{x}^0, \ldots, \mathbf{x}^k)$: an execution of $\mathbf{A}$ under the set $F(\rho)$ of faulty nodes,
- $U(\rho) = \{\mathbf{u}_{ij} : i \in [n] \setminus F(\rho), j \in [k]\}$: the configurations observed by non-faulty nodes.

That is, the algorithm $\mathbf{A}(\rho)$ is determined by the $a(\cdot)$ variables assigned true in ρ, the set $F(\rho)$ by the $p(\cdot)$ variables, and so on.

If an assignment ρ exists, then either $\mathbf{A}(\rho)$ is a correct algorithm or $X(\rho)$ gives an execution that violates the specification of synchronous 2-counting. In the latter case, the search algorithm inspects $X(\rho)$ and adds constraints that forbid any other solutions ρ' such that $\mathbf{A}(\rho) = \mathbf{A}(\rho')$. Of course, a naïve approach is to add constraints that explicitly exclude algorithm $\mathbf{A}$. However, inspecting the transition functions carefully allows for more frugal constraints that forbid several algorithms, that is, a tighter abstraction refinement.

1. Let $\Psi \leftarrow \psi_{\text{base}} \wedge \tau_0 \wedge \tau_1$.
2. While $\exists \rho$ such that $\rho \models \Psi \wedge \psi_{\text{illegal}}$:
   - Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\rho, 1)$.
3. Let $\Psi \leftarrow \Psi \wedge \tau_2 \wedge \cdots \wedge \tau_t$.
4. While $\exists \rho$ such that $\rho \models \Psi$:
   (a) If $\exists \sigma$ such that $\sigma \models \Psi \wedge \Gamma(\rho) \wedge \neg z(t) \wedge \neg o(t)$:
      - Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\sigma, t)$.
   (b) Otherwise:
      - Stop and output "$\mathbf{A}(\rho)$ is a correct algorithm".
5. Stop and output "no algorithm exists".

Fig. 6. Basic search algorithm. Steps 2, 4, and 4a resort to a SAT solver to find a satisfying assignment of a given formula.

The *basic search algorithm* is given in Fig. 6. Step 1 defines the initial formula that acts as a basis for the incremental search. In Step 2, the search algorithm first removes all algorithm candidates that do not correctly oscillate between the $\mathbf{0}_F$ and $\mathbf{1}_F$ states even in the special case when the system starts from either state. The formula ψ_{illegal} is defined as $(z(0) \wedge \neg o(1)) \vee (o(0) \wedge \neg z(1))$, and the formulas $\psi_{\text{forbid}}(\cdot, \cdot)$ are constraints that remove bad algorithms from the search space—we will describe these in detail in Section 10.3.

Step 4 asks the SAT solver to guess an algorithm candidate $\mathbf{A}(\sigma)$. In Step 4a, the SAT solver is used to find a counter-example to $\mathbf{A}(\sigma)$ to see whether it stabilises. If a counter-example is found, then we use the counter-example to add more constraints to prune the search space. Here, the formula $\Gamma(\rho)$ encodes $\mathbf{A}(\rho)$ as a conjunction of literals consisting of variables $a(\mathbf{u}, i, b)$. Step 4b is reached if no counter-example is found, meaning that $\mathbf{A}$ is a correct algorithm for synchronous counting.

Finally, if we reach Step 5, we know that Ψ is unsatisfiable, and hence, there does not exist any correct algorithms for the given parameters.

Remark. Note that there exist several possible trade-offs between having a simple search algorithm and speeding up synthesis by introducing problem-specific knowledge into the algorithm and encoding. For example, Step 2 essentially learns Lemma 4.1 which we could also directly encode into the base formulas. In Step 4, we can introduce $z(0)$ as a conjunct into the formula to make the search for $\mathbf{A}(\sigma)$ intuitively easier, and so on. However for clarity of exposition, we will focus on more general algorithmic ideas instead of problem-specific tunings.

10.3. Refinement through counter-examples

Once the SAT solver finds a counter-example, we need to forbid algorithms that exhibit the incorrect behaviour. Intuitively, we add constraints that force the change of *some* transitions that caused the bad execution.

Formally, we construct $\psi_{\text{forbid}}(\sigma, k)$ as follows. Let $\mathbf{x}^0, \ldots, \mathbf{x}^k$ be the execution $X(\sigma)$ and let $\mathbf{u}_{ij}$ be the configuration observed by node $i \notin F(\sigma)$ at timepoint $j < k$. The literals responsible for the transitions are divided into two sets, P^+ and P^-, as follows:

$$(i, j, b) \in P^+ \quad \text{iff} \quad \sigma[a(\mathbf{u}_{ij}, i, b)] = 1$$
$$(i, j, b) \in P^- \quad \text{iff} \quad \sigma[a(\mathbf{u}_{ij}, i, b)] = 0.$$

Above, $\sigma[x] \in \{0, 1, \perp\}$ denotes the value (false, true, unassigned) of variable x in assignment σ. Now the constraint is

$$\psi_{\text{forbid}}(\sigma, k) = \bigvee_{(i,j,b) \in P^+} a(\mathbf{u}_{ij}, i, b) \vee \bigvee_{(i,j,b) \in P^-} \neg a(\mathbf{u}_{ij}, i, b). \tag{33}$$

Note that the case $P^+ = P^- = \emptyset$ must be a contradiction, and hence the formula is always non-empty.

10.4. Improvement: finding short loops

The constraint can be strengthened when $X(\sigma)$ contains a loop $\mathbf{x}^0, \ldots, \mathbf{x}^h$ for some $h < k$, by then only considering timepoints $j \leq h$ when constructing the sets P^+ and P^-. Then, instead of stating that some transition must be changed in the entire length-k execution, we state that it suffices to change something for only $h < k$ of the steps. This results in a shorter disjunction in the constraint.

To this end, we modify Step 4 in the basic search algorithm as shown in Fig. 7. We introduce a new variable $\ell(k)$ which is true iff $\mathbf{x}^0 = \mathbf{x}^k$. We first find the smallest $k < t$ for $\mathbf{A}(\rho)$ such that a bad execution consisting of a length-k loop exists. If no such loop exists, we proceed as before. Otherwise, we use the counter-example consisting of a loop to refine the current abstraction.

4. While $\exists \rho$ such that $\rho \models \Psi$:
   (a) If $\exists k \leq t$ and $\exists \sigma$ such that $\sigma \models \Psi \wedge \Gamma(\rho) \wedge \ell(k)$:
      • Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\sigma, k^*)$, where k^* is the smallest such k.
   (b) Otherwise, if $\exists \sigma$ such that $\sigma \models \Psi \wedge \Gamma(\rho) \wedge \neg z(t) \wedge \neg o(t)$:
      • Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\sigma, t)$.
   (c) Otherwise:
      • Stop and output "$\mathbf{A}(\rho)$ is a correct algorithm".

Fig. 7. Finding short loops: modifications to Step 4 of Fig. 6.

1. Let $\Psi \leftarrow \psi_{\text{base}} \wedge \tau_0 \wedge \tau_1$.
2. While $\exists \rho$ such that $\rho \models \Psi \wedge \psi_{\text{illegal}}$:
   • Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\rho, 1)$.
3. Let $k \leftarrow 1$.
4. While $\exists \rho$ such that $\rho \models \Psi \wedge z(0)$:
   (a) Let $i \leftarrow \min\{j \leq k : \exists \sigma_j \text{ such that } \sigma_j \models \Psi \wedge \Gamma(\rho) \wedge \ell(j)\} \cup \{\infty\}$.
   (b) If $i \leq k$:
      • Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\sigma_i, i)$.
   (c) Otherwise, if $\exists \pi$ such that $\pi \models \Psi \wedge \Gamma(\rho) \wedge \neg z(k) \wedge \neg o(k)$:
      • If $k < t$:
         ◦ Let $k \leftarrow k + 1$ and $\Psi \leftarrow \Psi \wedge \tau_k$,
         ◦ Resume from Step 4a.
      • Otherwise:
         ◦ Let $\Psi \leftarrow \Psi \wedge \psi_{\text{forbid}}(\pi, k)$.
   (d) Otherwise:
      • Output "$\mathbf{A}(\rho)$ is a correct algorithm that stabilises in k steps",
      • Let $k \leftarrow k - 1$ and $t \leftarrow k$,
      • Resume from Step 4b.
5. Stop and output: "no algorithm exists that stabilises in time t".

Fig. 8. Overshooting algorithm.

10.5. Improvement: overshooting and unrolling on demand

Usually we are interested in knowing whether there exist *any* stabilising counting algorithm for given parameter values s, n, and f. For this task, we modify the search algorithm so that it can first quickly find some algorithm, possibly with a very long stabilisation time, and then gradually further tightening the stabilisation-time requirement.

The *overshooting algorithm* is given in Fig. 8. It unrolls the encoding on demand. By setting $t = \infty$, the algorithm tries to find *any* algorithm that stabilises. Of course, as the state space is finite, there is also a finite upper bound on t that can be used here.

The algorithm works as follows. Step 4a searches for the smallest i such that a i-loop counter-example exists for $\mathbf{A}(\rho)$. In Step 4b, if we have already unrolled the execution to at least i steps, then we add new constraints. Otherwise, Step 4c attempts to find a counter-example π of length k. If $k < t$, then we unroll the encoding for one additional time step, as it may be that our current time bound k is too small for a stabilising algorithm to exist. Otherwise, we prune the search space using the counter-example π.

11. Empirical results

So far we have introduced two different approaches for constructing synchronous counting algorithms. Now the obvious question remains: *which one is better?* To answer this, we empirically compared the direct encoding given in Section 9 against the counter-example guided algorithm described in Section 10. In particular, our goal was to find out which method is more useful in practice when one wants to synthesise new algorithms.

Solvers For solving instances via the direct propositional encoding, we used two freely available state-of-the-art complete SAT solvers: MiniSAT [26] (version 2.2.0 with simplifications) and lingeling (version `ayv`) [6]. The input formula was encoded in the standard DIMACS CNF file format. As both solvers allow a wide range of different parameters to fine-tune the solver search routines, we settled on running both solver using their respective default parameters.

Our implementation of the counter-example guided search, dubbed as symsync, builds on top of the incremental interface of the MiniSAT solver [25]. We used the overshooting variant of the counter-example guided search. Thus, the solver relaxes the time bound when it does not find a correct algorithm matching the target stabilisation time, but after finding some stabilising algorithm, the solver will then gradually tighten the time bound.

Experiment setup Recall that an instance of the synthesis problem consists of the class of algorithms (general or cyclic) and four parameters: number of nodes n, faulty nodes f, states s, and the stabilisation time t. We chose a set of problem

Table 2
Problem instances used in the empirical experiments. For all realisable instances, we also run the experiments for relaxed instances with stabilisation time $t+1$, $2t$, and the maximum possible stabilisation time. The last column gives the $\log_{10}$ of the number of algorithm candidates.

class	n	s	t	realisable?	$\log_{10}$ of #candidates
cyclic	4	3	6	no	38
	7	2	3	no	38
	8	2	3	no	77
	4	3	7	yes	38
	5	3	6	yes	115
	6	3	3	yes	347
	7	2	8	yes	38
	8	2	4	yes	77
general	4	3	7	yes	154
	5	2	79	no	48
	5	3	4	yes	579
	6	2	6	yes	115
	7	2	8	yes	269

instances consisting of both realisable (an algorithm exists) and unrealisable (no algorithm exists) instances, as listed in Table 2. We attempted to choose instances of various difficulty, but still solvable within a four hour limit on CPU time; we note that some of the algorithms presented in Table 1 of Section 8 required considerably longer time to synthesise.

For each problem instance, we ran $N = 100$ copies of each of the three solvers, initialising every process with a different random seed. We recorded the running time, the maximum memory footprint, and other statistics for each process. When using the direct encoding, we did not include the time required to generate the instance. The experiments were run on a computing cluster with Intel Xeon X5650 2.67-GHz processors. Each process was single-threaded and the memory limit was set to 8 GB.

For each realisable problem instance listed in Table 2, we also ran the same experiment setup as above for relaxed instances by increasing the stabilisation time bound in three ways: increasing the stabilisation bound by one, doubling the bound, and finally using the maximal bound of $t = s^{n-f} - 2$ time steps. Intuitively, suboptimal algorithms with a longer stabilisation time should be more common, and hence, perhaps easier to find. However, this also increases the size of the search space and the size of the SAT instances.

Results The synthesis times for realisable instance are summarised in Table 3 and Fig. 9. For each solver, the table gives the median together with first and ninth decile of synthesis times (in seconds). The time to generate the propositional formula for direct encoding instances is not included in the running times of MiniSAT and lingeling solvers, but is for symsync solver, as it iteratively generates its internal encoding within the CEGAR loop during execution.

The immediate observation is that neither direct encoding or the CEGAR approach consistently outperform the other. However, it is easy to see some patterns. First, the direct encoding works well for finding *optimal* or nearly-optimal algorithms, but finding *some* algorithm is much faster with symsync. On the other hand, symsync rarely manages to find optimal algorithms within the time limit of four hours or the memory limit of eight gigabytes.

Typically, when the solvers failed to find a solution, this was due to hitting the time limit. The only notable exceptions to this were the instances for general algorithms with $n = 5$ and $s = 3$, where each symsync instance ran out of memory in each case, and the cyclic instances with $n = 6$ and $s = 3$, where most of the failures were caused by running out of memory. Neither MiniSAT nor lingeling ran out of memory in these experiments.

The second pattern is that in many cases symsync gives solutions to instances with $s = 2$ states at least an order of magnitude faster than the direct encoding approach. For general algorithms with $n \in \{6, 7\}$, the direct encoding approach does not even produce results in the given time limit.

Indeed the observed behaviour is expected. The symsync solver refines the abstraction and relaxes the time bound if a fast algorithm is not found steadily increasing the size of the encoding. Usually, some algorithm will be encountered, and from thereon, the solver will simply proceed by adding new constraints until an algorithm with the desired time bound is found. On the other hand, trying to find *some* algorithm using the direct encoding amounts to simply increasing the time bound to a large enough value right from the start—this greatly increases the size of the propositional formula making the search slower.

When comparing the two different SAT solvers used in the direct encoding approach, rather unsurprisingly, the actively developed lingeling solver outperforms MiniSAT. We suspect that lingeling greatly benefits from its inprocessing capabilities, which are not present in the other solvers.

Table 3
Summary of *realisable* problem instances. The solver columns indicate the first, fifth (median), and ninth decile of running times in seconds. Columns marked with ⋆ indicate that a solution was found by some but less than 10% of the processes. For the first decile we have highlighted the running time of the fastest solver. Here $f = 1$ for all cases.

Instance				Running time (seconds)								
				MiniSAT			lingeling			symsync		
class	n	s	t	10%	50%	90%	10%	50%	90%	10%	50%	90%
cyclic	4	3	7	1	1	1	1	1	1	1	2	6
			8	1	1	3	1	1	1	1	1	5
			14	1	1	1	1	1	1	1	1	4
			25	1	1	2	1	1	1	1	1	4
	5	3	6	2373	–	–	**803**	2715	–	–	–	–
			7	1477	13305	–	**44**	632	711	⋆	–	–
			12	25	436	3009	12	16	91	**5**	31	1014
			79	66	672	4180	114	167	441	**3**	18	42
	6	3	3	79	3634	–	**16**	22	70	–	–	–
			4	⋆	–	–	**178**	272	3734	–	–	–
			6	2053	–	–	**251**	2451	4344	⋆	–	–
			241	6930	–	–	1981	2735	–	**41**	505	–
	7	2	8	**34**	604	4177	65	–	–	⋆	–	–
			9	32	560	2356	**21**	26	101	5233	–	–
			16	16	102	661	18	72	79	**2**	20	84
			62	41	442	1921	60	185	267	**2**	5	35
	8	2	4	**7**	101	440	19	67	81	–	–	–
			5	**15**	119	797	28	56	83	–	–	–
			8	62	558	3000	**50**	56	216	622	7304	–
			126	850	4117	–	967	3945	7993	**9**	21	145
general	4	3	7	10 859	–	–	**4246**	–	–	–	–	–
			8	2639	–	–	**497**	–	–	–	–	–
			14	**2884**	–	–	3211	–	–	–	–	–
			25	**2600**	–	–	13 639	–	–	–	–	–
	5	3	4	⋆	–	–	⋆	–	–	–	–	–
			5	⋆	–	–	⋆	–	–	–	–	–
			8	⋆	–	–	⋆	–	–	–	–	–
			79	–	–	–	–	–	–	–	–	–
	6	2	6	–	–	–	–	–	–	1167	–	–
			7	–	–	–	–	–	–	541	–	–
			12	⋆	–	–	⋆	–	–	69	1782	–
			30	⋆	–	–	⋆	–	–	46	382	2069
	7	2	8	–	–	–	–	–	–	528	–	–
			9	–	–	–	–	–	–	354	8990	–
			16	–	–	–	–	–	–	111	946	–
			62	–	–	–	–	–	–	75	415	–

The results for unrealisable instances are listed in Table 4. For unrealisable instances, it is relatively clear that the direct encoding outperforms the counter-example guided approach, although symsync is able to prove the non-existence of a two-state algorithm for $n = 5$ nodes in time comparable to the direct encoding approach.

12. Conclusions

In this work, we have used computational techniques to study the synchronous counting problem. At first sight the problem is not well-suited for computational algorithm design—we need to reason about stabilisation from any given starting

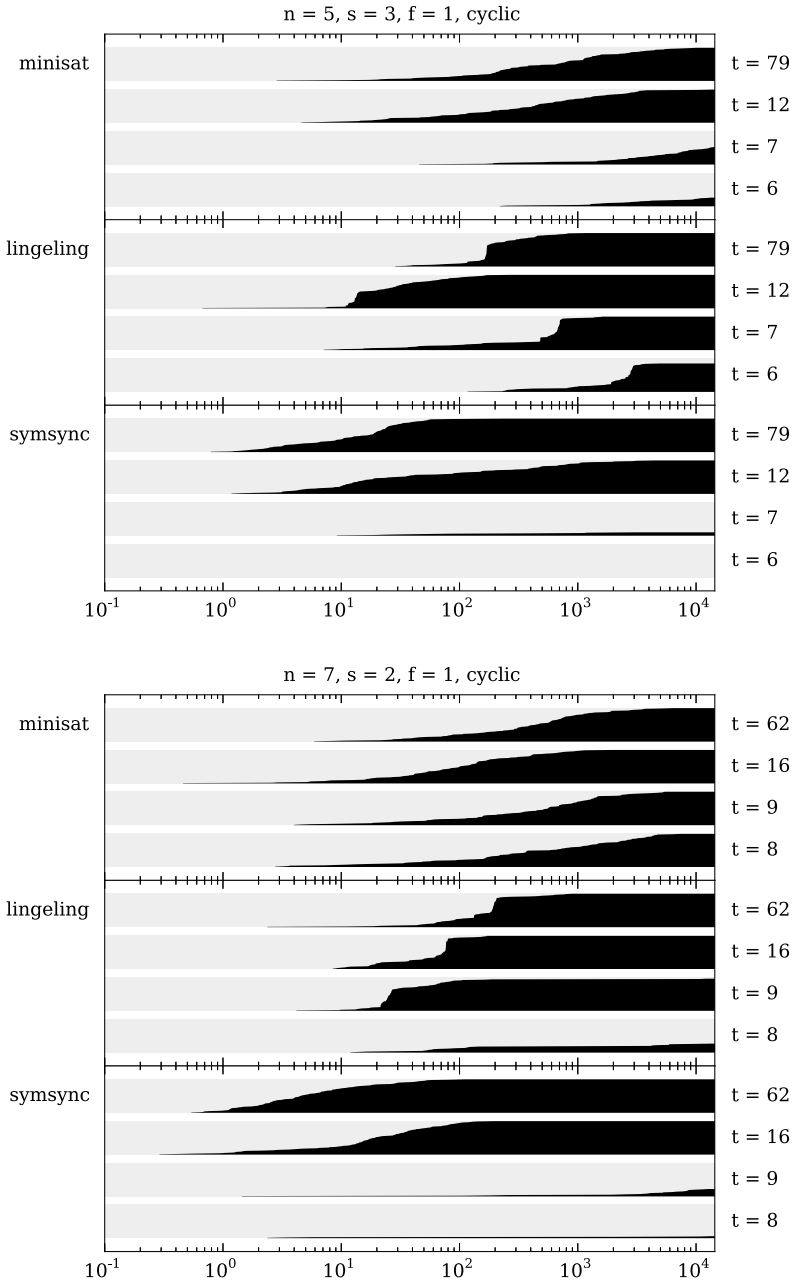


Fig. 9. Example of synthesis times. The x axis is the logarithm of time in seconds and the y-axis is the fraction of processes that had solved the problem instance.

Table 4
Summary of *unrealisable* problem instances.

Instance				Running time (seconds)								
				MiniSAT			lingeling			symsync		
class	n	s	t	10%	50%	90%	10%	50%	90%	10%	50%	90%
cyclic	4	3	6	2	3	3	4	6	6	–	–	–
	7	2	7	–	–	–	–	–	–	–	–	–
	8	2	3	9405	13 809	–	999	1364	1612	–	–	–
general	5	2	79	1148	1502	2016	1563	2353	2927	2482	2780	3421

Table 5
Cyclic algorithm for $s = 3$, $n = 4$, $f = 1$, and $t = 7$.

	00	01	02	10	11	12	20	21	22
00	1	1	1	1	0	1	1	1	1
01	1	1	1	2	2	0	1	1	1
02	1	1	1	1	0	1	1	1	1
10	1	0	1	1	0	0	1	0	1
11	0	0	0	0	0	0	0	0	0
12	1	0	1	0	0	0	0	0	0
20	1	1	1	1	1	0	1	1	1
21	1	1	1	1	0	0	1	0	0
22	1	1	1	1	0	0	1	0	1

configuration, for any adversarial behaviour, in a system with arbitrarily many nodes. Nevertheless, we have demonstrated that computational techniques can be used in this context to discover novel algorithms.

Our algorithms outperform the best human-designed algorithms: they are deterministic, small ($2 \leq s \leq 3$), fast ($3 \leq t \leq 8$), and easy to implement in hardware or in software—a small lookup table suffices. In summary, our work leaves very little room for improvement in the case of $f = 1$. The general case of $f > 1$ has been considered in subsequent work [43], which shows how the algorithms designed in this work can be used as subroutines to construct efficient algorithms that tolerate a larger number of failures.

We presented two complementary approaches for algorithm synthesis: the direct SAT encoding from Section 9 and the SAT-based CEGAR approach from Section 10. In our experiments, the direct encoding was typically the fastest method for finding *optimal* algorithms, while the CEGAR approach quickly discovered *some* algorithms.

Even though our computer-generated algorithms are constructed with a fairly complicated toolchain, the end results are compact, machine readable, and easy to verify with a straightforward script. All results and the verification tools are freely available online [1].

Acknowledgments

This work is an extended and revised version of a preliminary conference report [21]. We thank Josef Widder and Igor Konnov for helpful suggestions, and Nicolas Braud-Santoni, Aristides Gionis, Tomi Janhunen, Jussi Rintanen and Ulrich Schmid for discussions.

DD: Danny Dolev is Incumbent of the Berthold Badler Chair in Computer Science. This research project was supported in part by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11), by grant 3/9778 of the Israeli Ministry of Science and Technology.

KH and SW: Work supported by Academy of Finland under grants 139402 and 277522.

MJ: Work supported by Academy of Finland under grants 251170 COIN Centre of Excellence in Computational Inference Research, 276412, and 284591.

JHK, JR, JS: This work was supported in part by the Helsinki Doctoral Programme in Computer Science – Advanced Computing and Intelligent Systems, by the Academy of Finland (grants 132380 and 252018), and by the Research Funds of the University of Helsinki. Part of this work was done while JR and JS were affiliated with the University of Helsinki.

CL: This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-AF-0937274, CNS-1035199, 0939370-CCF and CCF-1217506, the AFOSR under Award number FA9550-13-1-0042, and the German Research Foundation (DFG, reference number Le 3107/1-1).

Computer resources were provided by the Aalto University School of Science "Science-IT" project, and by the Department of Computer Science at the University of Helsinki.

Appendix A. Algorithm listings

In this appendix, we give two examples of our algorithms—machine-readable versions of all algorithms, verification code, and some illustrations are available online [1].

Table 5 gives a cyclic algorithm for $n = 4$. The rows are labelled with (x_0, x_1), the columns are labelled with (x_2, x_3), and the values indicate $A_0((x_0, x_1, x_2, x_3))$, that is, the new state of the first node in the observed configuration $\mathbf{x}$. The projection graph (Section 6) for this algorithm is given in Fig. 10.

Table 6 shows a non-cyclic algorithm for $n = 6$. Again, the rows are labelled with the first half (x_0, x_1, x_2) of the observed state $\mathbf{x}$ and the columns are labelled with the second half (x_3, x_4, x_5) of the observed state $\mathbf{x}$. The values show the new state for each node: $A_0(\mathbf{x}), A_1(\mathbf{x}), \ldots, A_5(\mathbf{x})$.

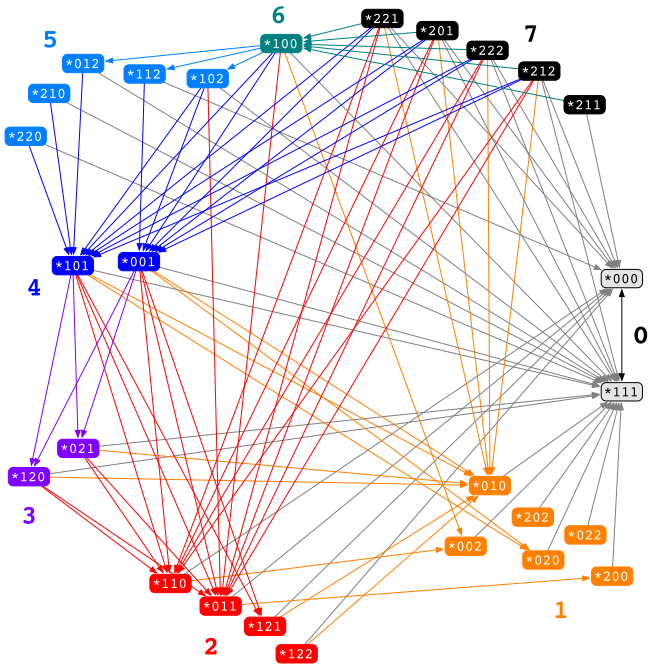


Fig. 10. The projection graph $G_F(\mathbf{A})$ for the algorithm **A** given in Table 5, assuming that the faulty nodes are $F = \{0\}$. The actual configurations have been clustered according to the length of the longest path that avoids the good states $\mathbf{0}_F$ and $\mathbf{1}_F$. Based on the projection graph, it is straightforward to verify that for any initial state and for any adversarial activities the algorithm will stabilise in $t = 7$ steps.

Table 6
Algorithm for $s = 2$, $n = 6$, $f = 1$, and $t = 6$.

	000	001	010	011	100	101	110	111
000	111111	111111	111111	111111	111111	111111	111111	011000
001	111111	111111	111111	111011	111011	111011	010001	010000
010	111111	111111	111111	101001	111111	101001	011111	001000
011	111111	111011	101001	100000	100001	100000	000001	000000
100	111111	111111	111111	110110	111111	110110	011111	000000
101	111111	111111	110110	110110	110110	110110	010000	000000
110	011111	110110	011111	000000	011111	000000	011111	001000
111	010110	010110	000000	000000	000010	000000	000001	000000

References

[1] Supplementary online material, https://github.com/suomela/counting (primary), https://bitbucket.org/suomela/counting (backup).

[2] Clark W. Barrett, David L. Dill, Aaron Stump, Checking satisfiability of first-order formulas by incremental translation to SAT, in: Proc. 14th International Conference on Computer Aided Verification, CAV 2002, in: Lecture Notes in Computer Science, vol. 2404, Springer, 2002, pp. 236–249.

[3] Michael Ben-Or, Danny Dolev, Ezra N. Hoch, Fast self-stabilizing Byzantine tolerant digital clock synchronization, in: Proc. 27th Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, ACM Press, 2008, pp. 385–394.

[4] Piotr Berman, Juan A. Garay, Kenneth J. Perry, Towards optimal distributed consensus, in: Proc. 30th Annual Symposium on Foundations of Computer Science, FOCS 1989, IEEE, 1989, pp. 410–415.

[5] Armin Biere, Bounded model checking, in: Armin Biere, Marjin Heule, Hans van Maaren, Toby Walsh (Eds.), Handbook of Satisfiability, IOS Press, Amsterdam, 2009, pp. 457–481, chapter 14.

[6] Armin Biere, Yet another local search solver and lingeling and friends entering the SAT competition 2014, in: Proc. SAT Competition 2014: Solver and Benchmark Descriptions, in: Department of Computer Science Series of Publications B, vol. B-2014-2, University of Helsinki, 2014, pp. 43–44, http://hdl.handle.net/10138/135571.

[7] Roderick Bloem, Uwe Egly, Patrick Klampfl, Robert Könighofer, Florian Lonsing, SAT-based methods for circuit synthesis, arXiv:1408.2333, August 2014.

[8] Roderick Bloem, Robert Könighofer, Martina Seidl, SAT-based synthesis methods for safety specs, in: Proc. 15h International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2014, in: Lecture Notes in Computer Science, vol. 8318, Springer, 2014, pp. 1–20.

[9] Nicolas Braud-Santoni, Roderick Bloem, Swen Jacobs, Synthesising resilient distributed systems, http://forsyte.at/download/frida14/braud-santoni-frida14.pdf, 2014.

[10] Daniel Bundala, Jakub Závodný, Optimal sorting networks, in: Proc. 8th International Conference on Language and Automata Theory and Applications, LATA 2014, in: Lecture Notes in Computer Science, vol. 8370, Springer, 2014, pp. 236–247.

[11] Alonzo Church, Logic, arithmetic, and automata, in: Proc. of the International Congress of Mathematicians, 1962, pp. 23–35.

[12] Edmund M. Clarke, E. Allen Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Proc. 3rd Workshop on Logic of Programs, LOP 1981, in: Lecture Notes in Computer Science, vol. 131, Springer, 1982, pp. 52–71.
[13] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (5) (2003) 752–794, http://dx.doi.org/10.1145/876638.876643.
[14] Edmund M. Clarke, Anubhav Gupta, Ofer Strichman, SAT-based counterexample-guided abstraction refinement, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 23 (7) (2004) 1113–1123, http://dx.doi.org/10.1109/TCAD.2004.829807.
[15] Michael Codish, Luís Cruz-Filipe, Michael Frank, Peter Schneider-Kamp, Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten), arXiv:1405.5754, May 2014.
[16] Ariel Daliot, Danny Dolev, Hanna Parnas, Self-stabilizing pulse synchronization inspired by biological pacemaker networks, in: Proc. 6th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2003, in: Lecture Notes in Computer Science, vol. 2704, Springer, 2003, pp. 32–48.
[17] Leonardo de Moura, Harald Rueß, Maria Sorea, Lazy theorem proving for bounded model checking over infinite domains, in: Proc. 18th International Conference on Automated Deduction, CADE-18, in: Lecture Notes in Computer Science, vol. 2392, Springer, 2002, pp. 438–455.
[18] Danny Dolev, The Byzantine generals strike again, J. Algorithms 3 (1) (1982) 14–30.
[19] Danny Dolev, Matthias Függer, Christoph Lenzen, Ulrich Schmid, Andreas Steininger, Fault-tolerant distributed systems in hardware, Bull. EATCS 116 (June 2015), http://bulletin.eatcs.org/index.php/beatcs/issue/view/18.
[20] Danny Dolev, Ezra N. Hoch, On self-stabilizing synchronous actions despite Byzantine attacks, in: Proc. 21st International Symposium on Distributed Computing, DISC 2007, in: Lecture Notes in Computer Science, vol. 4731, Springer, 2007, pp. 193–207.
[21] Danny Dolev, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, Synchronous counting and computational algorithm design, in: Proc. 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS 2013, in: Lecture Notes in Computer Science, vol. 8255, Springer, 2013, pp. 237–250, arXiv:1304.5719.
[22] Shlomi Dolev, Self-Stabilization, The MIT Press, Cambridge, MA, 2000.
[23] Shlomi Dolev, Jennifer L. Welch, Self-stabilizing clock synchronization in the presence of Byzantine faults, J. ACM 51 (5) (2004) 780–799, http://dx.doi.org/10.1145/1017460.1017463.
[24] Wolfgang Dvořák, Matti Järvisalo, Johannes Peter Wallner, Stefan Woltran, Complexity-sensitive decision procedures for abstract argumentation, Artificial Intelligence 206 (2014) 53–78, http://dx.doi.org/10.1016/j.artint.2013.10.001.
[25] Niklas Eén, Niklas Sörensson, Temporal induction by incremental SAT solving, Electron. Notes Theor. Comput. Sci. 89 (4) (2003) 543–560, http://dx.doi.org/10.1016/S1571-0661(05)82542-3.
[26] Niklas Eén, Niklas Sörensson, An extensible SAT-solver, in: Proc. 6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003, in: Lecture Notes in Computer Science, vol. 2919, Springer, 2004, pp. 502–518.
[27] E. Allen Emerson, Kedar S. Namjoshi, Reasoning about rings, in: Proc. 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, ACM, 1995, pp. 85–94.
[28] William Feller, Generalization of a probability limit theorem of Cramér, Trans. Amer. Math. Soc. 54 (3) (1943) 361–372.
[29] Bernd Finkbeiner, Swen Jacobs, Lazy synthesis, in: Proc. 13th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2012, in: Lecture Notes in Computer Science, vol. 7148, Springer, 2012, pp. 219–234.
[30] Bernd Finkbeiner, Sven Schewe, Uniform distributed synthesis, in: Proc. 20th Annual IEEE Symposium on Logic in Computer Science, LICS 2005, IEEE, 2005, pp. 321–330.
[31] Bernd Finkbeiner, Sven Schewe, Bounded synthesis, Int. J. Softw. Tools Technol. Transf. 15 (5–6) (2012) 519–539, http://dx.doi.org/10.1007/s10009-012-0228-z.
[32] Michael J. Fischer, Nancy A. Lynch, A lower bound for the time to assure interactive consistency, Inform. Process. Lett. 14 (4) (1982) 183–186, http://dx.doi.org/10.1016/0020-0190(82)90033-3.
[33] Cormac Flanagan, Rajeev Joshi, Xinming Ou, James B. Saxe, Theorem proving using lazy proof explication, in: Proc. 15th International Conferoence on Computer Aided Verification, CAV 2003, in: Lecture Notes in Computer Science, vol. 2725, Springer, 2003, pp. 355–367.
[34] Carsten Fuhs, Peter Schneider-Kamp, Synthesizing shortest linear straight-line programs over GF(2) using SAT, in: Proc. 13th International Conference on Theory and Applications of Satisfiability Testing, SAT 2010, in: Lecture Notes in Computer Science, vol. 6175, Springer, 2010, pp. 71–84.
[35] Daniel Große, Robert Wille, Gerhard W. Dueck, Rolf Drechsler, Exact multiple-control Toffoli network synthesis with SAT techniques, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 28 (5) (2009) 703–715, http://dx.doi.org/10.1109/TCAD.2009.2017215.
[36] Juho Hirvonen, Joel Rybicki, Stefan Schmid, Jukka Suomela, Large cuts with local algorithms on triangle-free graphs, arXiv:1402.2543, February 2014.
[37] Swen Jacobs, Roderick Bloem, Parameterized synthesis, in: Proc. 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2012, in: Lecture Notes in Computer Science, vol. 7214, 2012, pp. 362–376.
[38] Mikoláš Janota, Radu Grigore, Joao Marques-Silva, Counterexample guided abstraction refinement algorithm for propositional circumscription, in: Proc. 12th European Conference on Logics in Artificial Intelligence, JELIA 2010, in: Lecture Notes in Computer Science, vol. 6341, Springer, 2010, pp. 195–207.
[39] Mikoláš Janota, William Klieber, Joao Marques-Silva, Edmund Clarke, Solving QBF with counterexample guided refinement, in: Proc. 15h International Conference on Theory and Applications of Satisfiability Testing, SAT 2012, in: Lecture Notes in Computer Science, vol. 7317, Springer, 2012, pp. 114–128.
[40] Mikoláš Janota, Joao Marques-Silva, Abstraction-based algorithm for 2QBF, in: Proc. 14th International Conference on Theory and Applications of Satisfiability Testing, SAT 2011, in: Lecture Notes in Computer Science, vol. 6695, Springer, 2011, pp. 230–244.
[41] Matti Järvisalo, Petteri Kaski, Mikko Koivisto, Janne H. Korhonen, Finding efficient circuits for ensemble computation, in: Proc. 15th International Conference on Theory and Applications of Satisfiability Testing, SAT 2012, in: Lecture Notes in Computer Science., vol. 7317, Springer, 2012.
[42] Arist Kojevnikov, Alexander S. Kulikov, Grigory Yaroslavtsev, Finding efficient circuits using SAT-solvers, in: Proc. 12th International Conference on Theory and Applications of Satisfiability Testing, SAT 2009, in: Lecture Notes in Computer Science, vol. 5584, Springer, 2009, pp. 32–44.
[43] Christoph Lenzen, Joel Rybicki, Jukka Suomela, Towards optimal synchronous counting, in: Proc. 34th Annual ACM Symposium on Principles of Distributed Computing, PODC 2015, ACM Press, 2015, pp. 441–450.
[44] Nancy A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, San Francisco, 1996.
[45] Zohar Manna, Pierre Wolper, Synthesis of communicating processes from temporal logic specifications, ACM Trans. Program. Lang. Syst. 6 (1) (1984) 68–93.
[46] Jiří Matoušek, Jan Vondrák, The probabilistic method: lecture notes, http://kam.mff.cuni.cz/~matousek/prob-ln.ps.gz, March 2008.
[47] Andreas Morgenstern, Manuel Gesell, Klaus Schneider, Solving games using incremental induction, in: Proc. 10th International Conference on Integrated Formal Methods, IFM 2013, in: Lecture Notes in Computer Science, vol. 7940, Springer, 2013, pp. 177–191.
[48] Andreas Morgenstern, Klaus Schneider, Synthesis of parallel sorting networks using SAT solvers, in: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2014, OFFIS-Institut Für Informatik, 2011, pp. 71–80.
[49] Thomas Moscibroda, Rotem Oshman, Resilience of mutual exclusion algorithms to transient memory faults, in: Proc. 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, ACM Press, 2011, pp. 69–78.
[50] Moni Naor, Larry Stockmeyer, What can be computed locally?, SIAM J. Comput. 24 (6) (1995) 1259–1277, http://dx.doi.org/10.1137/S0097539793254571.

[51] Marshall C. Pease, Robert E. Shostak, Leslie Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234, http://dx.doi.org/10.1145/322186.322188.
[52] Amir Pnueli, Roni Rosner, Distributed reactive systems are hard to synthesize, in: Proc. 31st Annual Symposium on Foundations of Computer Science, FOCS 1990, vol. 2, 1990, pp. 746–757.
[53] Joel Rybicki, Exact bounds for distributed graph colouring, Master's thesis, Department of Computer Science, University of Helsinki, May 2011, http://urn.fi/URN:NBN:fi-fe201106091715.
[54] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, Sanjit Seshia, Combinatorial sketching for finite programs, in: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, ACM, 2006, pp. 404–415.
[55] Christoph M. Wintersteiger, Youssef Hamadi, Leonardo de Moura, Efficiently solving quantified bit-vector formulas, Form. Methods Syst. Des. 42 (1) (2012) 3–23, http://dx.doi.org/10.1007/s10703-012-0156-2.