Synthesis of Self-Stabilising and Byzantine-Resilient Distributed Systems

Roderick Bloem[1], Nicolas Braud-Santoni[1], and Swen Jacobs[2]

[1] Graz University of Technology, Austria
[2] Saarland University, Germany

Abstract. Fault-tolerant distributed algorithms play an increasingly important role in many applications, and their correct and efficient implementation is notoriously difficult. We present an automatic approach to synthesise provably correct fault-tolerant distributed algorithms from formal specifications in linear-time temporal logic. The supported system model covers synchronous reactive systems with finite local state, while the failure model includes strong self-stabilisation as well as Byzantine failures. The synthesis approach for a fixed-size network of processes is *complete* for realisable specifications, and can optimise the solution for small implementations and short stabilisation time. To solve the bounded synthesis problem with Byzantine failures more efficiently, we design an incremental, CEGIS-like loop. Finally, we define two classes of problems for which our synthesis algorithm obtains solutions that are not only correct in fixed-size networks, but in networks of arbitrary size.

1 Introduction

Distributed algorithms are hard to implement. While multi-core processors, communicating embedded devices, and distributed web services have become ubiquitous, it is very hard to correctly construct such systems because of the interplay between separate components and the possibility of uncontrollable faults.

While verification methods try to prove correctness of a system that has been implemented manually, the goal of *synthesis methods* is the automatic construction of systems that satisfy a given formal specification. The difference between these approaches is shown in Figure 1, illustrating how synthesis can relieve the designer from tedious and error-prone manual implementation and bug-fixing. Despite these benefits, formal methods that guarantee correctness *a priori*, like synthesis, have hardly found their way into distributed system design. This is in contrast to *a posteriori* methods like *verification*, which are being studied very actively [35, 45, 46].

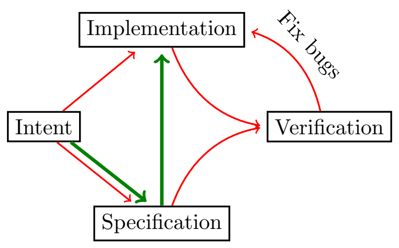

Fig. 1: Comparison of the verification ($\longrightarrow$) and synthesis ($\boldsymbol{\longrightarrow}$) workflows

One reason for this is that the underlying computational problems in synthesis are even harder than in verification. However, research on synthesis has picked up again in recent years [1, 8, 24, 36, 52], pushed forward by advances in theorem proving [26,42] and model checking algorithms [10,40] that can be used as building blocks for efficient synthesis algorithms.

In particular, also the synthesis of *concurrent and distributed systems* has received more attention lately. However, research in this area is to a large extent still restricted to basic theoretical problems [23, 24] or to simpler sub-problems, such as synthesis of synchronisation for existing programs [12, 54].

Failure resilience is critical in this setting, for two reasons: firstly, it enables use in *safety-critical* applications with weaker assumptions on the environment and the component itself. Secondly, it is needed to ensure *scalability* in practice, since in large networks it is unrealistic to assume that all components work without failure. In this work, we consider two kinds of failures: *transient failures*, as exemplified by self-stabilising systems [16, 53], where the whole system can be transported to an arbitrary state; and *permanent failures*, as exemplified by Byzantine failures [38], where some processes can deviate arbitrarily from the algorithm they should be executing.

Previous approaches for the synthesis of failure-resilient systems are either restricted in the systems that are considered [14], or in the kinds of failures that are supported [21].[1] For systems that support both self-stabilisation and Byzantine failures, the only result known to us is a problem-specific and semi-automatic approach by Dolev et al. [15], explained in the following.

Motivating Example [15]. Consider the problem of automatically constructing a distributed counter, ranging over m possible values. Processes in the system are arranged in a fully informed clique with synchronous timing, and should satisfy the following properties under self-stabilisation and Byzantine failures:

a) *agreement*: at every turn, all processes output the *same* value;
b) *increment*: the value is incremented in each step (mod m).

Since *increment* is an easy-to-implement local property, the main problem is *agreement* on the value. Dolev et al. [15] have recently shown a semi-automatic approach to obtain solutions for this problem. For a fixed number f of Byzantine nodes, they construct the distributed algorithm in two steps:

1. synthesis for a clique of sufficient size n;
2. extension to arbitrarily many processes.

The first step is based on a problem-specific encoding of the synthesis problem into a SAT problem. For a fixed number n of processes, the approach finds a solution – if one exists – by searching for implementations of increasing size and with increasing stabilisation time. The sufficient size of the clique is $n = 3 \cdot f + 1$, since this guarantees that (after stabilisation) the Byzantine nodes

[1] Related to this are also approaches for the synthesis of *robust* systems [6], essentially modelling failures in the environment of a single process.

cannot change the majority value [43]. Therefore, processes that are added to a correct system will be correct if they simply replicate the majority output of the existing processes.

The results of Dolev et al. [15] are impressive, since their solutions for the 2-counting problem extend to systems of arbitrary size, and have smaller state space and stabilisation time than any hand-crafted solution before. However, application of their approach to other problems requires significant effort for the development of a problem-specific encoding, and for proving its correctness.

In contrast, we introduce a general-purpose method for synthesis of failure-resilient systems that is fully automatic, can easily be proven correct, and is applicable to a wide range of problems. In particular, our preliminary implementation can replicate the results of Dolev et al. [15] and extend them to n-counting (with $n > 2$).

Contributions. In this paper, we propose a novel approach for the automatic synthesis of Byzantine-tolerant self-stabilising systems, in the form of distributed labelled transition systems. Our synthesis method takes as input a description of the network of processes and a specification in linear-time temporal logic, as well as a bound on the number of Byzantine processes in the network. It encodes the existence of a solution into a problem in satisfiability modulo theories (SMT), and synthesises correct implementations for all processes, if they exist.

We show that our method is correct and complete, and will terminate if a bound on the size of process implementations is given.

The first-order problems that result from our encoding critically need quantification over finite, but possibly large, sets. We provide a dedicated approach to solve those problems incrementally. On a prototype implementation of the approach, we show that this makes our examples tractable.

Finally, we give new results for extending our synthesis method from networks of fixed size to families of networks of unbounded size, based on the notion of *cutoffs* and the Parameterised Synthesis approach [30]. In particular, we define *colourless* specifications (or tasks) for non-terminating systems in cliques and similar network topologies, as well as a class of *local* specifications for networks with a fixed degree. For colourless specifications, we obtain cutoffs that depend on the number of Byzantine nodes, while for local specifications we obtain cutoffs that depend on the stabilisation time.

Structure. We introduce our system model and class of specifications in Section 2, and the basic synthesis approach in Section 3. We present the incremental approach for solving our synthesis problem in Section 4, the extension to parametric networks in Section 5, and experimental results in Section 6.

2 System and Failure Model, Specifications

We consider distributed systems that are defined by a fixed network of finite-state processes, in a *synchronous* composition: in every global step of the system, each process observes the outputs (possibly the complete state) of neighbouring components, and makes a transition. Our composition models atomic *snapshot*, the classical communication model for self-stabilising systems [13]. Furthermore, synchronous timing (possibly as an abstraction of the system behaviour) is a standard assumption when reasoning about consensus problems, as these problems are undecidable in asynchronous networks in the presence of faults [25, 39, 47]. To support asynchronous systems, one option is to use an abstraction to an effectively synchronous system, like for example in the model based on "communication rounds" by Dragoi et al. [18].

In the following, we formalise these notions for the case of fixed-size networks. We will consider networks of parametric size in Section 5.

2.1 System Model

Labelled Transition Systems. For given finite sets Σ of inputs and Υ of labels – or outputs – a *Υ-labelled Σ-transition system* (or short: a (Υ, Σ)-LTS) $\mathcal{T}$ is a tuple (T, T_0, τ, o) of a set T of states, a set $T_0 \subseteq T$ of initial states, a *transition function* $\tau : T \times \Sigma \rightarrow T$ and a *labelling (or output) function* $o : T \rightarrow \Upsilon$. $\mathcal{T}$ is called *finite* if T is finite.

We consider $\Upsilon = 2^O$ and $\Sigma = 2^I$, representing valuations of a set of Boolean output variables O (controlled by the system) and a set of Boolean input variables I (not controlled by the system).

Communication Graphs, Symmetry Constraints. A *communication graph* C is a tuple $(V, X, \mathcal{I}, \mathcal{O})$, where V is a finite set of nodes, X is a set of system variables, and $\mathcal{I} : V \rightarrow \mathcal{P}(X)$, $\mathcal{O} : V \rightarrow \mathcal{P}(X)$ assign sets of input and output variables to the nodes, with $\mathcal{O}(v) \cap \mathcal{O}(v') = \emptyset$ for all $v \neq v' \in V$. For a given v, we call $(\mathcal{I}(v), \mathcal{O}(v))$ the *interface* of v, and $(|\mathcal{I}(v)|, |\mathcal{O}(v)|)$ the *type* of the interface of v. If $\mathcal{I}(v) \cap \mathcal{O}(v') \neq \emptyset$, i.e., an output of v' is an input of v, then we say that v and v' are *neighbours* in C. Variables that are not assigned (by $\mathcal{O}$) as output variables to any of the nodes are *global input variables*, controlled by the environment. Denote this set of variables as $\mathcal{O}(env)$.

The communication graph may come with a *symmetry constraint*, given as a partitioning $V_1 \dot\cup \ldots \dot\cup V_m = V$ of the set of nodes. We assume that for every element V_i of the partition, nodes $v, v' \in V_i$ have the same type of interface, and that interfaces of all nodes have a fixed order that identifies corresponding in- and outputs of v and v'. The intended semantics is that nodes in the same element of the partition should have the same implementation modulo this correspondence.

Distributed Systems. An *implementation* of a node $v \in V$ in a communication graph C is a $(2^{\mathcal{O}(v)}, 2^{\mathcal{I}(v)})$-LTS. A *distributed system* is defined by a communication graph C and a finite family $(\mathcal{L}_v)_{v \in V}$ of implementations.

Let $C = (V, X, \mathcal{I}, \mathcal{O})$ be a communication graph with $V = \{v_1, \ldots, v_n\}$, and for every $v_i \in V$ let $\mathcal{L}_i = (L_i, L_{0,i}, \tau_i, o_i)$ be an implementation of v_i in C. The *composition* of $(\mathcal{L}_v)_{v \in V}$ in C is the $(2^X, 2^{\mathcal{O}(env)})$-LTS $\mathcal{G} = (G, G_0, \tau, o)$ with:

- $G = L_1 \times \ldots \times L_n$,
- $G_0 = L_{0,v_1} \times \ldots \times L_{0,v_n}$,
- $o(l_1, \ldots, l_n) = o_1(l_1) \cup \ldots \cup o_n(l_n)$, and

– $\tau((l_1, \ldots, l_n), e) = ((\tau_1(l_1, \sigma_1), \ldots, \tau_n(t_n, \sigma_n))$, where $\sigma = o(l_1, \ldots, l_n) \cup e$ and σ_i is the restriction of σ to variables in $\mathcal{I}(v_i)$.

Note that this is essentially the same formalism as in Finkbeiner and Schewe's seminal paper [24], and in the following we re-use part of their work on encoding the synthesis problem for such systems into SMT.

2.2 Failure Model

We consider two kinds of failures: *transient failures* that are limited in *time*, but may affect the whole system, and *permanent failures* that are limited in their *locations*, i.e., only affect a subset of the processes. We model these failures as *self-stabilisation* and *Byzantine failures*, respectively. The conjunction of both kinds of failures is called *Byzantine tolerant self-stabilisation* [17].

Self-Stabilisation. Self-stabilisation is the strongest model for *transient failures*, introduced by Dijkstra [13, 16, 53]; it assumes that the system as a whole fails – once – and is put in an arbitrary state. When the failure is over, processes resume their execution from this state. In transition systems, it is thus easily modelled by making all global states of the system initial.

Since an arbitrary state of the system will in general not satisfy strict safety requirements, in self-stabilisation one usually requires that a specification will *eventually* be satisfied, i.e., after a (either fixed or unknown) *stabilisation time*.

Byzantine Failures. Byzantine failure is a model of *permanent failure* where some processes do not execute the protocol, but are under the control of a *Byzantine adversary*. Our assumptions on the adversary are:

– *non-adaptiveness*: the adversary picks the set of faulty nodes before the algorithm is run;
– *full information*: the adversary can read the global state of the system;
– *computational power*: the adversary has *unbounded* computational power.

In our setting, the *non-adaptiveness* does not remove any power from the adversary [11].[2] Therefore, it is equivalent to the *strong Byzantine adversary*, which subsumes most models of permanent failure. We will consider systems with a fixed upper bound f on the number of Byzantine failures.

2.3 Formal Specifications

We consider formal specifications in linear-time temporal logic (LTL), where the atomic propositions are the system variables. A formula that uses only the input and output variables of a tuple $\overline{v} = (v_1, \ldots, v_k)$ of nodes will sometimes be written $\varphi(\overline{v})$. We assume that the body of our specification is of the form

$$\forall \overline{v} \in V^k : \quad \varphi(\overline{v}),$$

for some $k \leq |V|$.

[2] Essentially, this is because our model is not probabilistic, and because the protocol must work for any choice of Byzantine nodes and any behaviour they can exhibit, which includes all possible behaviours of an adaptive adversary.

Example 1. Consider a fully connected network of a set of nodes V. Suppose every process $v \in V$ has a binary output variable c_v. In the 2-counting problem from Section 1, every node v has an output c_v, and the formal specification φ is the conjunction

$$\begin{aligned} &\forall v \in V. \quad \mathsf{G}\,(c_v = 0 \leftrightarrow \mathsf{X}\,c_v = 1) \\ \wedge\, &\forall v_1, v_2 \in V. \quad \mathsf{G}\,(c_{v_1} = c_{v_2})\,, \end{aligned}$$

stating that (for every node) the binary output should be flipped in every step, and (for all pairs of nodes) the output of two nodes should always be the same.

Fault-Tolerant Specifications. Since we consider systems that exhibit both self-stabilisation and Byzantine failures, we need to consider a special type of specifications:

- *self-stabilisation* implies that specifications φ with non-trivial safety requirements (like in Example 1) in general cannot be satisfied without explicitly allowing a stabilisation time. Therefore, we consider specifications φ that are either of the form $\mathsf{F}\,\psi$ (if we allow an unspecified stabilisation time), or of the form $\mathsf{X}^t\,\psi$ (if we require that the stabilisation time is bounded by t steps).
- *Byzantine failures* imply that the respective nodes can behave arbitrarily, and properties of the specification can not be expected to hold for them. Therefore, we require that for every choice of the Byzantine nodes, the specification holds only for tuples of *correct* nodes, i.e., where none of the nodes is Byzantine. Formally, this means that instead of the original specification $\forall\, \overline{v} \in V^k : \varphi(\overline{v})$ we consider the specification

$$\forall\, \overline{b} \in V^f, \overline{v} \in V^k : \left[\left(\bigwedge_{1 \le i \le k, 1 \le j \le f} v_i \neq b_j\right) \rightarrow \varphi(\overline{v})\right]. \qquad (1)$$

Example 2. Recall the second part of the 2-counting specification:

$$\forall v_1, v_2 \in V. \quad \mathsf{G}\,(c_{v_1} = c_{v_2})\,.$$

For systems with one Byzantine node b in V, this property is modified to:

$$\forall b \in V.\ \forall v_1, v_2 \in V. \quad [(v_1 \neq b \wedge v_2 \neq b) \ \rightarrow\ \mathsf{G}\,(c_{v_1} = c_{v_2})]\,.$$

3 Bounded Synthesis of Resilient Systems

Synthesising distributed systems is in general undecidable [23, 44, 48]—with or without failures—and only becomes decidable by bounding the size of the implementation. The *bounded synthesis* problem consists in constructing an implementation that satisfies a given temporal logic specification and a bound on the number of states.

Finkbeiner and Schewe [24] gave an algorithm for bounded synthesis based on an encoding into *satisfiability modulo theories (SMT)*. Inspired by their encoding, we describe in the following an algorithm for the bounded synthesis of distributed systems with Byzantine-tolerant self-stabilisation. The high-level structure of the approach is depicted in Figure 2.

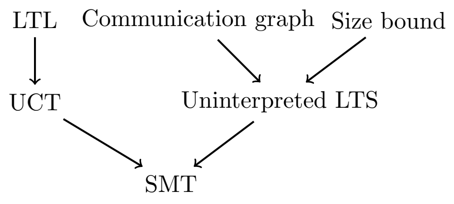

Fig. 2: The *bounded synthesis* approach

Input: specification and communication graph. The input to our synthesis problem is a communication graph $C = (V, X, \mathcal{I}, \mathcal{O})$, possibly with a symmetry constraint $V_1 \dot\cup \ldots \dot\cup V_m = V$, and a (global) temporal specification φ over atomic propositions in X. In the following, let $O = \bigcup_{v \in V} \mathcal{O}(v)$ be the set of global output variables (controlled by the system), and $I = X \setminus O$ the set of global input variables (controlled by the environment) of C.

In the following we encode the existence of local implementations of the nodes in V such that the composition of these implementations in C satisfies φ.

Conversion of the Specification to an Automata. Using the approach of Kupferman and Vardi [37], the specification φ is translated into a *universal co-Büchi tree automaton (UCT)* $\mathcal{A}_\varphi = (Q, q_o, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ the initial state, $\delta : Q \times 2^O \to \mathcal{P}(Q \times 2^I)$ a transition relation, and $F \subseteq Q$ a set of rejecting states. A given UCT $\mathcal{A}$ *accepts* an $(2^O, 2^I)$-LTS $\mathcal{T}$ if no run in the parallel execution of $\mathcal{A}$ and $\mathcal{T}$ visits a rejecting state infinitely often. The UCT $\mathcal{A}_\varphi$ is constructed such that it accepts an LTS $\mathcal{T}$ if and only if $\mathcal{T} \models \varphi$.

As an optimisation, we use a Safra-less conversion to generalised Rabin automata[50] rather than converting to a co-Büchi automata, as the automatically-generated Rabin automata are smaller. However, co-Büchi and Rabin automata are known to be equally expressive.

Uninterpreted LTS Based on Size Bound and Communication Graph. Let s be a local size bound for implementations of nodes in C. Then, for each node $v_i \in \{v_1, \ldots, v_n\} = V$, we want to find a $(2^{\mathcal{O}(v_i)}, 2^{\mathcal{I}(v_i)})$-LTS $\mathcal{L}_i = (L_i, L_{0,i}, \tau_i, o_i)$, with:

- a set of (local) states L_i with $|\, L_i \,| = s$;
- a set of initial states $L_{0,i} \subseteq L_i$;
- a transition function $\tau_i : L_i \times 2^{\mathcal{I}(v_i)} \to L_i$;
- a labelling function $o : L_i \to 2^{\mathcal{O}(v_i)}$.

L_i can be considered a fixed set of elements, while the transition and labelling functions are to be synthesised. That is, in our SMT encoding they are considered as *uninterpreted* functions. If $v_i, v_j \in V_k$ for some V_k in the symmetry constraint, then we introduce just *one* uninterpreted function symbol that is used for both τ_i and τ_j, and similarly for o_i and o_j. This enforces the constraint that both nodes will have the same implementation. The choice of L_{0_i} is explained below.

Encoding of UCT and LTS into SMT Problem. To encode the synthesis problem, we follow the approach of Finkbeiner and Schewe [24]. Let $\mathcal{G} = (G, G_0, \tau, o)$ be the composition of the local implementations $\mathcal{L}_i$ in C. Then, we define an (uninterpreted) *annotation function* $\lambda : Q \times G \to \mathbb{Q} \cup \{\bot\}$ that maps states in the product automaton $\mathcal{A}_\varphi \times \mathcal{G}$ to either $\bot$ or a rational number. To ensure that $\mathcal{G}$ is accepted by $\mathcal{A}_\varphi$, we introduce constraints on λ such that λ tracks whether states in the product automaton $\mathcal{A}_\varphi \times \mathcal{G}$ are reachable, and bounds the number of visits to rejecting states in runs of $\mathcal{A}_\varphi \times \mathcal{G}$. In particular, we require that

$$\forall g_0 \in G_0 : \; \lambda(q_0, g_0) \neq \bot \tag{2}$$

$$\begin{aligned}\forall q, q' \in Q, g \in G, \sigma \in \Sigma : \; & \lambda(q,g) \neq \bot \wedge (q', \sigma) \in \delta(q, o(g)) \wedge q' \notin F \\ & \to \lambda(q', \tau(g,\sigma)) \geq \lambda(q,g)\end{aligned} \tag{3}$$

$$\begin{aligned}\forall q, q' \in Q, g \in G, \sigma \in \Sigma : \; & \lambda(q,g) \neq \bot \wedge (q', \sigma) \in \delta(q, o(g)) \wedge q' \in F \\ & \to \lambda(q', \tau(g,\sigma)) > \lambda(q,g)\end{aligned} \tag{4}$$

The conjunction (2) $\wedge$ (3) $\wedge$ (4), in the following denoted as SMT_φ, encodes the existence of an implementation that satisfies φ in a system *without failures*.

Encoding Self-Stabilisation. We encode self-stabilisation by considering all states of the system as initial states, i.e., $L_{0_i} = L_v$. Thus, a solution to our synthesis problem has to ensure that the specification φ is satisfied for *all* runs that begin in *any* of the states of the composed system. This corresponds directly to the definition of self-stabilisation introduced by Dijkstra [13].

Encoding Byzantine Failures. Byzantine nodes can behave arbitrarily, and the Byzantine adversary has information about the global state of the system [38]. Thus, the behaviour of Byzantine failures can be modelled by allowing them to give arbitrary outputs at any time, essentially re-assigning their outputs to outputs of the adversarial environment.

To encode this, we modify the constraints above such that Byzantine processes are observed with arbitrary output. In particular, if node v_i is Byzantine, then in (3) and (4) we replace each occurrence of a system variable $x \in \mathcal{O}(v_i)$ with a fresh variable x_b, and add a quantifier $\forall x_b \in \mathbb{B}$. For a given formula ψ and $\overline{x}(v) = \mathcal{O}(v)$, this substitution is denoted as $\forall \overline{x_b}(v) \in \mathbb{B}.\ \psi[\overline{x}(v) \mapsto \overline{x_b}(v)]$.

Example 3. Consider that the first component v_1 is Byzantine, let $\overline{x} = \mathcal{O}(v_1)$, $g = (l_1, \ldots, l_n)$ and $\sigma = (\sigma_1, \ldots, \sigma_n)$. Then, constraint (3) is modified to:

$$\begin{aligned}&\forall \overline{x_b} \in \mathbb{B}, q, q' \in Q, g \in G, \sigma \in \Sigma : \\ &\qquad \lambda(q,g) \neq \bot \wedge (q', \sigma[\overline{x} \mapsto \overline{x_b}]) \in \delta(q, o(g)) \wedge q' \notin F \\ &\qquad \to \lambda(q', (\tau_1(l_1, \sigma_1[\overline{x} \mapsto \overline{x_b}]), \ldots, \tau_n(l_n, \sigma_n[\overline{x} \mapsto \overline{x_b}])) \geq \lambda(q,g).\end{aligned}$$

Finally, since the Byzantine adversary can choose a set of at most f Byzantine nodes, we have to quantify over all possible choices of the adversary. Since the satisfaction of the specification can depend on the choice of processes, we have an important change to the encoding: our quantification has to reflect that the correctness argument, and therefore the witness function λ, can depend on this choice, while the transition and labelling functions τ, o must not depend on this choice. This results in the following encoding:

$$\begin{aligned} &\exists \tau, o.\ \ \forall \{b_1, \ldots, b_f\} \subseteq V.\ \ \exists \lambda. \\ &\qquad\qquad \forall \overline{x_b}(b_1, \ldots, b_f) \in \mathbb{B}.\ \ \mathsf{SMT}_\varphi[\overline{x}(b_1, \ldots, b_f) \mapsto \overline{x_b}(b_1, \ldots, b_f)]. \end{aligned} \tag{5}$$

Note that with this encoding, all neighbours of a Byzantine process observe the *same* outputs. It is straightforward to have different observed outputs for different neighbours in our encoding, at the cost of introducing one quantified variable for the observation of each neighbour.[3]

Furthermore, note that all quantification is over finite sets, so we can eliminate all quantifiers by Skolemising λ such that it is a function that depends on the choice of $\{b_1, \ldots, b_f\}$, and explicitly instantiating the universal quantifiers.

Correctness. For uninterpreted functions τ, o, λ, satisfiability of our encoding is equivalent to the existence of an LTS that satisfies the specification φ. Moreover, satisfying valuations of τ and o give us a solution to the synthesis problem, and the valuation of λ witnesses correctness of that solution.

Theorem 1 (Correctness for fixed bound). *The constraint system* (5) *is satisfiable if and only if the specification is finite-state realisable in a self-stabilising system with f Byzantine nodes in the given communication graph. A satisfying assignment of τ and o represents a solution to the synthesis problem.*

Up to the encoding of failures, our encoding is equivalent to that of Finkbeiner and Schewe, and correctness follows from Theorem 5 of [24]. Correctness of the encoding of self-stabilisation is straightforward, and correctness of the encoding of Byzantine failures follows from our elaborations above.

Increasing the bound. By iterating bounded synthesis for increasing bounds, we obtain a semi-decision procedure for the synthesis problem.

Corollary 1 (General correctness and completeness). *A semi-procedure that iterates bounded synthesis of resilient systems for increasing bounds will eventually find a finite-state implementation of φ if it exists.*

Practical Applicability. Our encoding includes a large number of quantifiers, both universal and existential. Since we consider finite domains, they could all be explicitly instantiated, but experiments show that their full instantiation results in a combinatorial blowup that quickly makes the SMT formula intractable. For non-trivial examples, existing SMT solvers (such as Z3 and CVC3) were unable to solve the resulting problem instances.

[3] Also, note that *fail-stop failures* can be seen as a special case of Byzantine failures, and can be modelled in a similar way: instead of giving arbitrary outputs, the chosen nodes at some point move into a special *stop* state, from which they cannot recover.

Abstracting from the universal quantifiers inside the innermost existential quantifier (as these are treated rather efficiently by existing methods like incremental instantiation [29, 41] or in some cases simply full instantiation) and the concrete meaning of the function symbols, our synthesis problem is of the form

$$\exists x.\, \forall y.\, \exists z.\, Q(x, y, z)$$

where $Q(x, y, z)$ is an SMT formula and x, y, z are from finite domains. In the following, we introduce a new, incremental algorithm that performs well for instances of this problem produced by our encoding.

4 Incremental Synthesis Algorithm

In this section, we introduce an algorithm that allows us to solve synthesis problems more efficiently than a direct application of an SMT solver on the full encoding of the previous section. To this end, we extend the approach of *Counter-Example-Guided Inductive Synthesis* (CEGIS) [52][51, Chapter 4] to handle finite model extraction for first-order formulae with two quantifier alternations ($\exists\forall\exists$). Like CEGIS, our algorithm is only guaranteed to terminate when the universal quantification is over a finite domain.

4.1 Previous Work

Solar-Lezama et al. introduced CEGIS [51, 52] in the context of template-based synthesis, but it is a general method for solving first-order problems of the form $\exists x.\, \forall y.\, Q(x, y)$. It is complete and terminating if y belongs to a finite domain Y. CEGIS performs *model extraction*, which is crucial when doing synthesis.

In the following, we will use x, y, z as first-order variables and $\hat{x}, \hat{y}, \hat{z}$ as concrete values for these variables. CEGIS proceeds by building a database of *counterexamples* $\hat{y}_i$ for any *candidate* $\hat{x}$ that it has encountered. In the worst case, CEGIS performs $O(|Y|)$ SMT queries until it reaches a conclusion; it is especially efficient if every $\hat{y}$ eliminates a large portion of the possible values for x.

The CEGIS algorithm is shown in Figure 3. Formula $\phi(x)$ acts as the database of counterexamples. The algorithm uses two incremental SMT solvers. In Line 3, it extracts candidates for x that work for all counterexamples in the database. In Line 5 it uses a new variable y_{n_1} to extract a new counterexample that rejects at least the last candidate $\hat{x}$.

4.2 Extension to First-Order Model Extraction

The encoding of our synthesis problem is of the form

$$\exists x.\, \forall y.\, \exists z.\, Q(x, y, z). \tag{6}$$

In the specific case of our encoding, described earlier, x was called τ and ranges over uninterpreted functions over finite domains; y was called B and

```
Data: A first-order formula $\exists x.\, \forall y.\, Q(x,y)$
Result: FALSE or a value $\hat{x}$ such that $\forall y.\, Q(\hat{x},y)$ holds
1  Initialise $\phi(x)$ to $\top$ and $n = 0$;
2  while true do
3      if $\phi(x)$ is satisfiable then
4          Extract a concrete value $\hat{x}$ for $x$ from the model;
5          if $\neg Q(\hat{x}, y_{n+1})$ is satisfiable then
6              Extract a concrete value $\hat{y}_{n+1}$ for $y_{n+1}$ from the model;
7              $\phi := \phi \wedge Q(x, \hat{y}_{n+1})$;
8              $n{+}{+}$;
9          else
10             return $\hat{x}$;
11         end
12     else
13         return FALSE;
14     end
15 end
```

Fig. 3: Original CEGIS algorithm, solving $\exists x.\, \forall y.\, Q(x,y)$

ranges over tuples of process identifiers from a finite domain and z was called λ and ranges over the rationals.

While we still keep a set of counterexamples $\hat{y}_i$, candidate generation is now a little more intricate: we look for one $\hat{x}$ and a $\hat{z}_i$ for every $\hat{y}_i$ in the database.

The algorithm is shown in Figure 4. Here, $y_1 \dots y_n$ is (still) the database of counterexamples. The candidate extraction is again in Line 3, but is more intricate: it now extracts a candidate $\hat{x}$ for x as well as a candidate $\hat{z}_i$ for each counterexample $\hat{y}_i$. In Line 5 we then look for a new counterexample $\hat{y}_{n+1}$ that shows that the formula is false for $\hat{x}$ and any of the $\hat{z}_i$.

Note that, again, we can use two incremental SMT solvers. In the outer satisfiability call (Line 3), we only add conjunctive constraints to ϕ. In the inner satisfiability call (Line 5), we add conjunctive constraints to $\bigwedge_{i=1}^{n} \neg Q(\hat{x}, y_{n+1}, \hat{z}_i)$ as long as $\hat{x}$ does not change, and we reset the formula if $\hat{x}$ does change.[4]

Correctness Argument. Let us assume the algorithm returns $\hat{x}$. At the point where it returns, it has concrete values $\hat{z}_1 \dots \hat{z}_n$ such that there is no y that falsifies $Q(\hat{x}, y, \hat{z}_i)$ for all i. This means that for any y, there is a $\hat{z}_i$ such that $Q(\hat{x}, y, \hat{z}_i)$ is satisfied: we indeed exhibited a valid model for the formula.

Conversely, let us assume the algorithm returns **FALSE**: this means that the formula $\phi = \bigwedge_{i=1}^{n} Q(x, \hat{y}_i, z_i)$ is not satisfiable. Assuming our original formula

[4] In fact, in our prototype implementation we use a heuristic that avoids throwing away the formula by re-assigning the value of $\hat{x}$ in the formula whenever the outer SMT call returns a new value. Then, we do not throw away the formula at all, but risk that it grows unnecessarily large. In our experiments, this has shown favourable effects.

```
Data: A first-order formula $\exists x.\, \forall y.\, \exists z.\, Q(x, y, z)$
Result: FALSE or a value $\hat{x}$ such that $\forall y. \exists z.\, Q(\hat{x}, y, z)$ holds
Initialise $\phi(x, z_1, \ldots, z_n)$ to $\top$ and $n = 0$;
while true do
    if $\phi(x, z_1, \ldots, z_n)$ is satisfiable then
        Extract concrete values $\hat{x}, \hat{z}_1, \ldots, \hat{z}_n$ for $x, z_1, \ldots, z_n$ from the model;
        if $\bigwedge_{i=1}^{n} \neg Q(\hat{x}, y_{n+1}, \hat{z}_i)$ is satisfiable then
            Extract a concrete value $\hat{y}_{n+1}$ for $y_{n+1}$ from the model;
            $\phi := \phi \wedge Q(x, \hat{y}_{n+1}, z_{n+1})$;
            $n{+}{+}$;
        else
            return $\hat{x}$;
        end
    else
        return FALSE;
    end
end
```

Fig. 4: Proposed algorithm, solving $\exists x.\, \forall y.\, \exists z.\, Q(x, y, z)$

was satisfiable, and given a model $x, z(\cdot)$ that satisfies it, then $x, z_i = z(\hat{y}_i)$ would be a model for ϕ: hence, our original formula is UNSAT.

Finally, termination of the algorithm follows from the fact that our domains are finite, which implies that every formula only has finitely many satisfying assignments, and every call to the inner SMT solver strengthens the formula ϕ such that at least one satisfying assignment is removed.

4.3 Related Work

Our work is close to Janota et al. [31] which extends CEGIS to decide QBF formulas with arbitrary quantifier alternation. Janota et al. propose a recursive algorithm which uses a number of nested SMT calls linear in the number of quantifier alternations, whereas we need only two. Moreover, since candidate values are changed by subsequent SMT calls more often, the algorithm cannot efficiently use incremental solving.

Another modification of CEGIS that is close to ours is that of Koksal et al. [34]. At a high level (i.e., the level we chose for our description in this section), their approach is very similar to ours. The differences between the algorithms are in the encoding of synthesis problems, as well as in the specialised verification and synthesis algorithms that are part of the description in Koksal et al. [34]. We chose a higher level of description for the CEGIS algorithm in order to increase its re-usability in different settings.

Finally, another approach for the synthesis of reactive systems that uses incremental refinement of candidate models is *lazy synthesis* [22]. The difference to our approach is that lazy synthesis is not based on CEGIS and a direct encoding of correctness into SAT or SMT, but instead uses LTL model checking and an encoding of error traces into SMT to obtain and refine candidate models. Lazy synthesis does not consider systems with Byzantine failures, but could probably be extended to our setting by extending the LTL model checking to all possible choices of Byzantine nodes and all possible actions taken by the Byzantine adversary. Whether this would be efficient is an open question.

5 Extension to Networks of Unbounded Size

The synthesis method we have introduced thus far is restricted to systems with a fixed number of components. However, correctness in networks of arbitrary size is needed for scalability, as it is unfeasible to synthesise a new solution whenever new processes are introduced into the system. In this section, we show how to obtain process implementations that are correct in systems of arbitrary size, based on the idea of *Parameterised Synthesis* [30]: by combining a general correctness argument for a specific class of systems and specifications, we can synthesise systems that will be correct in networks of unbounded size by synthesising a solution that (i) satisfies the specification and (ii) belongs to the class of systems for which the correctness argument holds.

Parameterised Systems. Let $\mathcal{C}$ be the set of all communication graphs. Then a parameterised communication graph is a function $\Gamma : \mathbb{N} \rightarrow \mathcal{C}$, where we assume that every $\Gamma(i)$ comes with a symmetry constraint that separates the nodes into a finite number of implementation classes (with identical interface types). A parameterised communication graph Γ is *of order* k if, for all $n \in \mathbb{N}$, the number of implementation classes in $\Gamma(n)$ is less or equal to k. Then, an *implementation* of a parameterised communication graph Γ of order k is a set of implementations $\{\mathcal{T}_1, \dots, \mathcal{T}_k\}$ for its nodes, one for each implementation class.

Parameterised specifications. In specifications of parameterised systems, the atomic propositions are the system variables, indexed by fixed component identifiers or identifier variables. An identifier variable i can be quantified *globally* in the form $\forall i.\varphi$, or *locally* in the form $\forall i : \mathit{neighbour}(x).\varphi$. In every given instance of the parameterised communication graph, this quantification is simply interpreted as a finite conjunction over all possible values for i.

Cutoffs for Parameterised Synthesis. A central notion of parameterised synthesis is the *cutoff*: an upper bound c on the number of nodes in a network that need to be considered, such that components that are correct in the network of size c are also correct in any network of a bigger size. Formally, $c \in \mathbb{N}$ is a cutoff for a set of specifications Φ and a class of systems S if, for every $\varphi \in \Phi$ and every $(\Gamma, \{\mathcal{T}_1, \dots, \mathcal{T}_k\}) \in S$ (where Γ is of order k), it holds that

$$\forall n > c : \; (\{\mathcal{T}_1, \dots, \mathcal{T}_k\}, \Gamma(c) \models \varphi \;\Leftrightarrow\; \{\mathcal{T}_1, \dots, \mathcal{T}_k\}, \Gamma(n) \models \varphi)\,.$$

Thus, a cutoff enables parameterised synthesis if and only if we can guarantee that our solution belongs to the system class S. In principle, this idea can directly be applied to failure-resilient systems, but existing cutoff results [2,3,7,19,20,27] usually do not take into account fault-tolerance.

Colourless specifications. In distributed systems, there is a classical notion of (weakly) *colourless tasks* for terminating executions of a system. This includes many important properties of finite runs, such as consensus and k-set agreement. We extend this notion to infinite runs.

For a given global state $g = (l_1, \ldots, l_n)$ of a system $\mathcal{G}$, a *variant* of g is a state g' that can be obtained from g by changing the local state l_i of one process i to another local state $l'_i \in L_i$, such that $o_i(l'_i) = o_j(l_j)$ for some $j \neq i$, or by a sequence of such changes.

Then, define a specification φ to be *colourless* if for every execution trace $o(g_0)o(g_1)\ldots o(g_n)\ldots$ that satisfies φ, and any variant g'_n of g_n, the partial trace $o(g_0)o(g_1)\ldots o(g_{n-1})o(g'_n)$ can be extended to a trace that satisfies φ.

An example of a colourless specification is the m-counting specification from our motivating example. Note that colourlessness is a semantic property of a specification, and we do not supply a syntactic fragment of LTL that guarantees colourlessness.

Cutoffs for Colourless Specifications. We show how to extend an n-process system $\mathcal{G}$ satisfying a colourless specification φ into an $(n+k)$-process system, satisfying the same specification. We assume that the processes in $\mathcal{G}$ are fully connected (i.e., in a clique) and that state labels are unique, i.e., the output of a process is sufficient to conclude its current internal state. Based on these assumptions, we show how to synthesise a system that can be considered a larger clique, where the additional processes can have a different implementation.

The additional processes will have an implementation $\mathcal{L}'$ that is different from that of the original processes. $\mathcal{L}'$ reads the current input from the environment and the outputs of processes in the original clique, uses this information (which by assumption lets us conclude their current internal state) as well as knowledge about the original implementation $\mathcal{L}$ to anticipate their next transition, and moves to a state that has the same output as the majority of the next-states in the clique.

To ensure that this will result in a correct system even under up to f Byzantine nodes, we need to enforce an *f-majority property* in the original system: in every round, the output chosen by the largest number of correct nodes is picked by f more nodes than the second largest one.[5] Then, even if the computation of $\mathcal{L}'$ described above includes Byzantine nodes, its output will be equal to that of the majority of original implementations $\mathcal{L}$, and therefore the colourless specification will still be satisfied.

If we are synthesising the original system, the *f-majority property* can be directly encoded as an additional cardinality constraint over the outputs. This constraint preserves satisfiability of the synthesis constraints, even for a given state space.

To see this, assume that a given original system satisfies a colourless specification, but does not have the *f-majority property*. Then we can transform it into a system $\mathcal{G}'$ which simulates $\mathcal{G}$. At each step, processes in $\mathcal{G}'$ simulate $\mathcal{G}$ for one

[5] This is an extension of the argument for proving the exact number of Byzantine failures that can be survived while solving consensus problems [38].

step, and check whether f-majority is achieved. If it is not, then we can (partially) determinise the given system to obtain *f-majority*: for instance, nodes can be grouped by output value, and state and output value of some nodes can be replaced with ones from the largest group. The modified system still produces valid runs thanks to the specification being *colourless*.

Theorem 2. *If φ is a colourless specification, C is a fully informed clique and $(\mathcal{L}_v)_{v \in V}$ a set of implementations such that their composition $\mathcal{G}$ in C has the f-majority property and $\mathcal{G} \models \varphi$, then any extension of $\mathcal{G}$ with additional processes $\mathcal{L}'$ as described above will satisfy φ.*

Cutoffs for local specifications in regular networks. We can also obtain cutoffs for the setting that satisfies the following:

- the networks has a constant-degree – also called regular – where all nodes have the same interface and implementation,
- *local* specifications: specifications of the form $\forall i.\ \mathsf{X}^t\,\mathsf{G}\,\phi(i)$ (where $\phi(i)$ is a Boolean formula over the current state of processes in a maximal distance of r to a process i),
- a fixed number of Byzantine nodes f in a distance of r around any node, and
- a fixed stabilisation time t.

Theorem 3. *Let C be a constant-degree network with a given interface for all nodes, and such that all nodes have a maximal distance of $r + t$ from a central node v. If an implementation $\mathcal{L}$ satisfies a local specification $\varphi(v)$ in C, then $\mathcal{L}$ satisfies $\forall i.\ \varphi(i)$ in any C' with the same degree, the same interface, and a radius greater than $r + t$.*

The cutoff follows from the fact that our specifications only require that we enter the “legitimate states” specified by $\phi(i)$ within t steps, and never leave them afterwards, and within these t steps only information from nodes with this distance can enter the radius around i that $\phi(i)$ talks about. Because of full symmetry in these systems, it is sufficient to require $\varphi(v)$ instead of $\forall i.\ \varphi(i)$.

Specifications that can be expressed as purely first-order formulae can be rewritten as local specifications [9, 49]. This suggests that local formulae are expressive enough to be of interest: for instance, consensus is local despite k-set agreement being non-local.[6] [7]

6 Experimental Results

A preliminary implementation was written in OCaml, using Sickert’s formally proven correct tool [50] to convert LTL specifications to automata, and de Moura and Bjørner’s Z3 [42] as the backend SMT solver.

[6] It is not sufficient to prove that k-set agreement is not expressible in FO. However, k-set agreement – for any given $k \geq 2$ – can easily be proven non-local by contradiction.

[7] **RB: what is the significance of the results in this section?**

Experiments were run on a number of computers equipped with 64GiB of memory and eight cores clocked at 2.6GHz. Note that our solver is sequential and does not take advantage of multicore machines.

We were able to reproduce the results from Dolev *et al.* [15] regarding synchronous 2-counting with a single Byzantine adversary ($f = 1$). Each experiment – for a fixed set of parameters – took at most one hour. As in previous results [15], those solutions can be extended to any system of greater size while keeping the number f of failures, the stabilisation time t and the local state size s constants.

Moreover, we were able to synthesise a symmetric solution for 4-counting, for 4 processes with 5 states each, and stabilisation time 10. This improves on the solution suggested by Dolev et al. to simply duplicate a 2-counter to obtain a 4-counter, which would result in an implementation with 6 local states and a stabilisation time of at least 14 in this case. To our knowledge, this is the first instance of a solution to n-counting (with $n > 2$) ever synthesised. This result shows that our more general approach allows us to obtain even more efficient implementations than that of Dolev et al., without the need to manually devise a new encoding and argue about its correctness.

Class	processes (n)	local states (s)	Total states	Stabilisation time (t)
symmetric	4	3	12	7
	5	3	15	6
	6	3	18	3
	7	2	21	8
	8	2	16	4
general	4	4	16	5
	5	3	15	4
	6	2	12	6

Fig. 5: Synthesised algorithms for 2-counting with a single Byzantine failure

Attempts to replicate these results using directly a first-order model finder – such as CVC3 [5] – or existing extensions of CEGIS [31] resulted either in timeout (no result within 12h) or running out of memory.

Moreover, as mentioned in Section 3, we use a translation from LTL to Rabin automata [50]: we compared that approach to encoding universal co-Büchi automata obtained with ltl3ba [4] and observed a speedup from 25% to 50% depending on the instance.

7 Conclusion

We have presented a method to automatically synthesise distributed systems that are self-stabilising and resilient to Byzantine failures. We assume that the systems are specified in LTL. Our results apply to finite network graphs and extend to parameterised synthesis of processes on a graph of arbitrary size under reasonable conditions. The approach follows the basic idea of Bounded Synthesis.

It constructs an SMT formula with two quantifier alternations that states that a fault-tolerant implementation of a given size exists, and it is complete if a bound on the size of the process implementation is given. We have presented a CEGIS-style decision procedure to decide such formulas that is far more efficient than existing approaches for the formulas we have encountered. Finally, we show that we can efficiently synthesise a small solution for the 2-counter problem.

In this work, we only consider the synthesis of basic building blocks of distributed systems, modelled as labelled transition systems. To obtain actual large-scale implementations, many additional layers of complexity need to be addressed, and in practice there will be a trade-off between formality and automation on the one hand, and scale or precision of the system model on the other hand, as for example demonstrated in recent work of Hawbitzel et al. [28].

In the near future, we plan to extend our approach to more general timing models and to study more general specifications for parameterised synthesis. In particular, we want to extend our approach to the system model of the PSync language of Dragoi et al. [18], which enables reasoning about asynchronous systems by introducing a notion of "communication rounds", and will make our approach applicable to a much larger class of problems. Furthermore, we will look into optimisations of the encoding, as described by Khalimov et al. [32,33] for parameterised synthesis of systems without fault-tolerance.

Acknowledgements. We thank Igor Konnov, Ulrich Schmid, Josef Widder, and the late Helmut Veith for interesting discussions on formal methods for distributed systems. We also thank the anonymous reviewers for their detailed and insightful comments.

This work was supported by the Austrian Science Fund (FWF) through projects LogiCS (W1255-N23), QUAINT (I774-N23) and RiSE (S11406-N23), and by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center AVACS (SFB/TR 14) and through project ASDPS (JA 2357/2-1).

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design (FMCAD), 2013. pp. 1–8. IEEE (2013)
2. Aminof, B., Jacobs, S., Khalimov, A., Rubin, S.: Parameterized model checking of token-passing systems. In: McMillan, K.L., Rival, X. (eds.) VMCAI. LNCS, vol. 8318, pp. 262–281. Springer (2014), http://doai.io/10.1007/978-3-642-54013-4_15
3. Außerlechner, S., Jacobs, S., Khalimov, A.: Tight cutoffs for guarded protocols with fairness. In: VMCAI. LNCS, vol. 9583, pp. 476–494. Springer (2016)
4. Babiak, T., Kretínský, M., Rehák, V., Strejcek, J.: Ltl to büchi automata translation: Fast and more deterministic. In: TACAS. LNCS, vol. 7214, pp. 95–109. Springer (2012), http://doai.io/10.1007/978-3-642-28756-5_8
5. Barrett, C., Tinelli, C.: CVC3. In: CAV. LNCS, vol. 4590, pp. 298–302. Springer-Verlag (Jul 2007), berlin, Germany

6. Bloem, R., Chatterjee, K., Greimel, K., Henzinger, T.A., Hofferek, G., Jobstmann, B., Könighofer, B., Könighofer, R.: Synthesizing robust systems. Acta Informatica 51(3), 193–220 (2014), http://doai.io/10.1007/s00236-013-0191-5
7. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015), http://doai.io/10.2200/S00658ED1V01Y201508DCT013
8. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. J. Comput. Syst. Sci. 78(3), 911–938 (2012), http://doai.io/10.1016/j.jcss.2011.08.007
9. Bollig, B.: Logic for communicating automata with parameterized topology. In: CSL-LICS. pp. 18:1–18:10. ACM (2014)
10. Bradley, A.R.: Sat-based model checking without unrolling. In: VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011), http://doai.io/10.1007/978-3-642-18275-4_7
11. Canetti, R., Damgård, I., Dziembowski, S., Ishai, Y., Malkin, T.: On adaptive vs. non-adaptive security of multiparty protocols. In: Advances in CryptologyEUROCRYPT 2001, pp. 262–279. Springer (2001)
12. Cerný, P., Henzinger, T.A., Radhakrishna, A., Ryzhyk, L., Tarrach, T.: Efficient synthesis for concurrency by semantics-preserving transformations. In: Sharygina, N., Veith, H. (eds.) CAV. LNCS, vol. 8044, pp. 951–967. Springer (2013), http://doai.io/10.1007/978-3-642-39799-8_68
13. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (Nov 1974)
14. Dimitrova, R., Finkbeiner, B.: Synthesis of fault-tolerant distributed systems. In: ATVA. LNCS, vol. 5799, pp. 321–336. Springer (2009), http://doai.io/10.1007/978-3-642-04761-9_24
15. Dolev, D., Korhonen, J.H., Lenzen, C., Rybicki, J., Suomela, J.: Synchronous counting and computational algorithm design. In: SSS. LNCS, vol. 8255, pp. 237–250. Springer (2013), http://doai.io/10.1007/978-3-319-03089-0_17
16. Dolev, S.: Self-Stabilization. MIT Press (2000)
17. Dolev, S., Welch, J.L.: Self-stabilizing clock synchronization in the presence of byzantine faults. Journal of the ACM (JACM) 51(5), 780–799 (2004)
18. Dragoi, C., Henzinger, T.A., Zufferey, D.: PSync: a partially synchronous language for fault-tolerant distributed algorithms. In: POPL. pp. 400–415. ACM (2016), http://doai.io/10.1145/2837614.2837650
19. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003), http://doai.io/10.1142/S0129054103001881
20. Emerson, E., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) Automated Deduction - CADE-17, Lecture Notes in Computer Science, vol. 1831, pp. 236–254. Springer Berlin Heidelberg (2000)
21. Faghih, F., Bonakdarpour, B.: Smt-based synthesis of distributed self-stabilizing systems. TAAS 10(3), 21 (2015), http://doai.io/10.1145/2767133
22. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: VMCAI. LNCS, vol. 7148, pp. 219–234. Springer (2012)
23. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: (LICS 2005). pp. 321–330. IEEE Computer Society (2005), http://doai.io/10.1109/LICS.2005.53
24. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT 15(5-6), 519–539 (2013), http://doai.io/10.1007/s10009-012-0228-z

25. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985), http://doai.io/10.1145/3149.214121
26. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: CAV. Lecture Notes in Computer Science, vol. 3114, pp. 175–188. Springer (2004), http://doai.io/10.1007/978-3-540-27813-9_14
27. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
28. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: Ironfleet: proving practical distributed systems correct. In: SOSP. pp. 1–17. ACM (2015)
29. Jacobs, S.: Incremental instance generation in local reasoning. In: CAV. LNCS, vol. 5643, pp. 368–382. Springer (2009)
30. Jacobs, S., Bloem, R.: Parameterized synthesis. Logical Methods in Computer Science 10, 1–29 (2014), http://arxiv.org/abs/1401.3588
31. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving qbf with counterexample guided refinement. In: Theory and Applications of Satisfiability Testing–SAT 2012, pp. 114–128. Springer (2012)
32. Khalimov, A., Jacobs, S., Bloem, R.: PARTY parameterized synthesis of token rings. In: CAV. Lecture Notes in Computer Science, vol. 8044, pp. 928–933. Springer (2013)
33. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: VMCAI. LNCS, vol. 7737, pp. 108–127. Springer (2013)
34. Köksal, A.S., Pu, Y., Srivastava, S., Bodík, R., Fisher, J., Piterman, N.: Synthesis of biological models from mutation experiments. In: POPL. pp. 469–482. ACM (2013)
35. Konnov, I., Veith, H., Widder, J.: SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In: CAV. LNCS, vol. 9206, pp. 85–102. Springer (2015), http://doai.io/10.1007/978-3-319-21690-4_6
36. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Functional synthesis for linear arithmetic and sets. STTT 15(5-6), 455–474 (2013), http://doai.io/10.1007/s10009-011-0217-7
37. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS 2005. pp. 531–542. IEEE Computer Society (2005), http://doai.io/10.1109/SFCS.2005.66
38. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (Jul 1982), http://doai.io/10.1145/357172.357176
39. Lamport, L.: Brief announcement: Leaderless byzantine paxos. In: DISC. LNCS, vol. 6950, pp. 141–142. Springer (2011)
40. McMillan, K.L.: Applying sat methods in unbounded symbolic model checking. In: CAV. LNCS, vol. 2404, pp. 250–264 (2002), http://doai.io/10.1007/3-540-45657-0_19
41. de Moura, L.M., Bjørner, N.: Efficient e-matching for SMT solvers. In: CADE. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer (2007)
42. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008), http://doai.io/10.1007/978-3-540-78800-3_24
43. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM 27(2), 228–234 (1980)

44. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. 2013 IEEE 54th Annual Symposium on Foundations of Computer Science 0, 746–757 vol.2 (1990)
45. Qadir, J., Hasan, O.: Applying formal methods to networking: Theory, techniques and applications. CoRR abs/1311.4303 (2013), http://arxiv.org/abs/1311.4303
46. Saissi, H., Bokor, P., Muftuoglu, C.A., Suri, N., Serafini, M.: Efficient verification of distributed protocols using stateful model checking. In: SRDS. pp. 133–142. IEEE (2013), http://doai.io/10.1109/SRDS.2013.22
47. Saks, M.E., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000), http://doai.io/10.1137/S0097539796307698
48. Schewe, S.: Distributed synthesis is simply undecidable. Inf. Process. Lett. 114(4), 203–207 (2014), http://doai.io/10.1016/j.ipl.2013.11.012
49. Schwentick, T., Barthelmann, K.: Local normal forms for first-order logic with applications to games and automata. In: STACS 98, LNCS, vol. 1373, pp. 444–454. Springer Berlin Heidelberg (1998)
50. Sickert, S.: Converting linear temporal logic to deterministic (generalised) rabin automata. Archive of Formal Proofs 2015 (2015)
51. Solar Lezama, A.: Program Synthesis By Sketching. Ph.D. thesis, EECS Department, University of California, Berkeley (2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html
52. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS 2006. pp. 404–415. ACM (2006), http://doai.io/10.1145/1168857.1168907
53. Tixeuil, S.: Algorithms and Theory of Computation Handbook, Second Edition, chap. Self-stabilizing Algorithms, pp. 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures, CRC Press, Taylor & Francis Group (2009)
54. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: TACAS. LNCS, vol. 5505, pp. 139–154. Springer (2009), http://doai.io/10.1007/978-3-642-00768-2_13