Synthesis of Distributed Algorithms with Parameterized Threshold Guards*

Marijana Lazić[1], Igor Konnov[2], Josef Widder[3], and Roderick Bloem[4]

1 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`lazic@forsyte.at`
2 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`konnov@forsyte.at`
3 **TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria**
`widder@forsyte.at`
4 **TU Graz, Inffeldgasse 16a/II, 8010 Graz, Austria**
`roderick.bloem@iaik.tugraz.at`

Abstract

Fault-tolerant distributed algorithms are notoriously hard to get right. In this paper we introduce an automated method that helps in that process: the designer provides specifications (the problem to be solved) and a sketch of a distributed algorithm that keeps arithmetic details unspecified. Our tool then automatically fills the missing parts.

Fault-tolerant distributed algorithms are typically parameterized, that is, they are designed to work for any number n of processes and any number t of faults, provided some resilience condition holds; e.g., $n > 3t$. In this paper we automatically synthesize distributed algorithms that work for *all* parameter values that satisfy the resilience condition. We focus on threshold-guarded distributed algorithms, where actions are taken only if a sufficiently large number of messages is received, e.g., more than t or $n/2$. Both expressions can be derived by choosing the right values for the coefficients a, b, and c, in the sketch of a threshold $a \cdot n + b \cdot t + c$. Our method takes as input a sketch of an asynchronous threshold-based fault-tolerant distributed algorithm — where the guards are missing exact coefficients — and then iteratively picks the values for the coefficients.

Our approach combines recent progress in parameterized model checking of distributed algorithms with counterexample-guided synthesis. Besides theoretical results on termination of the synthesis procedure, we experimentally evaluate our method and show that it can synthesize several distributed algorithms from the literature, e.g., Byzantine reliable broadcast and Byzantine one-step consensus. In addition, for several new variations of safety and liveness specifications, our tool generates new distributed algorithms.

1998 ACM Subject Classification F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.4.5 [*Software*]: Operating systems: Fault-tolerance, Verification

Keywords and phrases fault-tolerant distributed algorithms – Byzantine faults – parameterized model checking – program synthesis

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.0

* Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403, S11405, and S11406), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103). The computational results presented have been achieved [in part] using the Vienna Scientific Cluster (VSC).

1 Introduction

Design and implementation of parameterized fault-tolerant distributed systems are error-prone tasks. There is a mature *theory* regarding mathematical proof methods, which found their way into formal frameworks like I/O Automata [24] and TLA+ [22]. Recent approaches [17, 23, 31] provide tool support to establish correctness of implementations, by manually constructing proofs with an interactive theorem prover. Although, if successful, this approach provides a machine-checkable proof [9, 4], it requires huge manual efforts from the user. A logic for distributed consensus algorithms in the HO Model [8] was introduced in [13], which allows one to automatically check the invariants (for safety) and ranking functions (for liveness), that is, the manual effort is reduced to finding right invariants and ranking functions. Model checking of distributed algorithms promises a higher degree of automation. For consensus algorithms in the HO Model, the results of [25] reduce the verification to checking small systems of five or seven processes. For the asynchronous model, an efficient model checking technique for threshold-guarded distributed algorithms was introduced in [20, 19]. Notably, this technique verifies both safety and liveness properties. In all these methods, the user has to produce an implementation (or design), and the goal is to check (using techniques that vary in the degree of automation) whether this implementation satisfies a given specification.

In this paper we explore synthesis as it promises even more automation. The user just provides required properties and a sketch of an asynchronous algorithm, and our tool automatically finds a correct distributed algorithm. In this way we generate new fault-tolerant algorithms that are *correct by construction*. In our experiments we first focus on existing specifications [29, 7, 30, 28] from the literature, in order to be able to compare the output of our tool with known algorithms. We then give new variations of safety and liveness specifications, and our tool generates new distributed algorithms for them.

Parametrized synthesis. Similar to the verification approaches above, we are interested in the parameterized version of the problem: Rather than synthesizing a distributed algorithm that consists of, say, four processes and tolerates one fault, our goal is to synthesize an algorithm that works for n processes, out of which t may fail, for all values of n and t that satisfy a resilience condition, e.g., $n > 3t$. This is in contrast to recent work on synthesis of fault-tolerant distributed algorithms [16, 15, 14] that requires the user to fix the number of processes; typically to some $n < 10$. In some special cases, manual arguments or cut-off theorems generalize synthesis results for small systems to parameterized ones [5, 12, 26]. However, similar to parameterized verification [2, 6], the parameterized synthesis problem is in general undecidable [18]. As in the parameterized verification approach of [19], we will therefore limit ourselves to a specific class of distributed algorithms, namely, *threshold-guarded distributed algorithms*. These thresholds are arithmetic expressions over parameters, e.g., $n/2$, and determine for how many messages processes should wait (a majority in the example).

More specifically, the user provides as input a distributed algorithm with holes as in Figure 1: The user defines the control flow, and keeps the threshold expressions — noted as τ_{0toSE} and τ_{AC} in the figure — unspecified. As pseudo code has no formal semantics, it cannot be used as a tool input. Rather, our tool takes as input a sketch threshold automaton.

▶ **Example 1.** Figure 1 is a pseudo code representation of the input, and Figure 2 shows the corresponding sketch threshold automaton; they are related as follows. The initial locations ℓ_0 and ℓ_1 of the sketch threshold automaton in Figure 2 correspond to initial states in Figure 1 where *myval* is equal to 0 and 1, respectively. Edges are labeled by $g \mapsto \texttt{act}$, where expression g is a threshold guard, and the action `act` may increment a shared variable.

Code of a correct process i:
var $myval_i \in \{0,1\}$
var $accept_i \in \{false, true\} \leftarrow false$

while *true* **do** *(in one step)*
if $myval_i = 1$
and not sent ECHO before
then send ECHO **to** all

if received ECHO **from** $\geq \tau_{\mathsf{0toSE}}$ **distinct** processes
and not sent ECHO before
then send ECHO **to** all

if received ECHO **from** $\geq \tau_{\mathsf{AC}}$ **distinct** processes
then $accept_i \leftarrow true$
od

Figure 1 A single-round version of the reliable broadcast algorithm [29] with holes

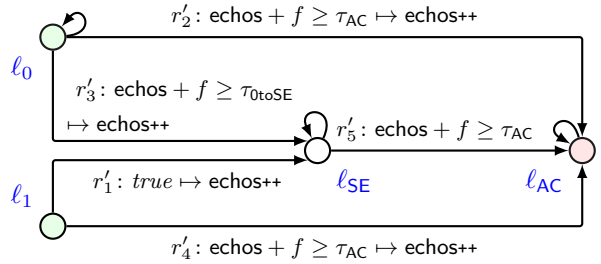

Figure 2 A sketch threshold automaton

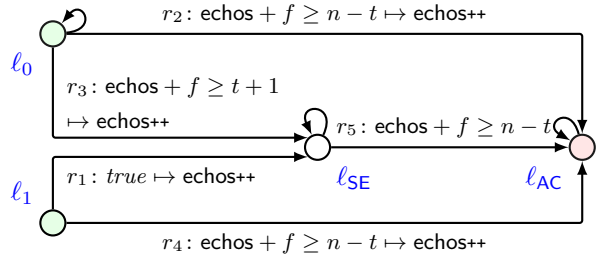

Figure 3 A synthesized threshold automaton

The action echos++ corresponds to the pseudo-code statement: `send <echo> to all`. (The message buffers are replaced by a shared variable that is increased whenever a message is sent. This typically can be done for algorithms that only count messages, and do not distinguish the senders. For instance, a bisimulation between models with message buffers and shared variables was proven in [21].) By going to local state ℓ_{SE}, a process records that it has sent *echo*. Finally, by going to the local state ℓ_{AC}, a process records that it has set accept to true. The "$+f$" terms in threshold guards model that messages from up to f Byzantine processes may be received ($f \leq t$), while we model only the $n - f$ correct processes explicitly.

We use self loops to capture the behavior that processes may take arbitrarily many steps before receiving a message sent to them, that is, asynchronous communication. For instance, the self loop in ℓ_0 allows processes to stay in ℓ_0 even if the guards of the other outgoing edges evaluate to true. That every message from a correct process is eventually received is a fairness constraint and is therefore not part of the threshold automaton, but is captured in the specifications. For instance, such a fairness constraint would be that if echos $\geq n - t$, then every process must eventually leave location ℓ_0. (In the fairness constraint, the "$+f$" does not appear, because messages from faulty processes are not guaranteed to arrive.)

The "holes" τ_{0toSE} and τ_{AC} in Figure 2 are the missing thresholds, which should be linear combination of the parameters n and t. Therefore τ_{0toSE} has the form $\mathbf{?}_1 \cdot n + \mathbf{?}_2 \cdot t + \mathbf{?}_3$, and τ_{AC} has the form $\mathbf{?}_4 \cdot n + \mathbf{?}_5 \cdot t + \mathbf{?}_6$. The unknown coefficients $\mathbf{?}_i$, for $1 \leq i \leq 6$, have to be found by the synthesis tool. One solution is $\mathbf{?}_1 = 0$, $\mathbf{?}_2 = 1$, $\mathbf{?}_3 = 1$, $\mathbf{?}_4 = 1$, $\mathbf{?}_5 = -1$, and $\mathbf{?}_6 = 0$, that is, $\tau_{\mathsf{0toSE}} = t + 1$ and $\tau_{\mathsf{AC}} = n - t$. This solution is depicted in Figure 3. ◁

In addition to a sketch threshold automaton, the user has to provide a specification, that is, safety and liveness properties the distributed algorithm should satisfy. Based on these inputs, our tool generates the required coefficients, that is, a threshold automaton as in Figure 3. The synthesis approach of this paper is enabled by a recent advance [19] in parameterized model checking of *safety and liveness* properties of distributed algorithms.

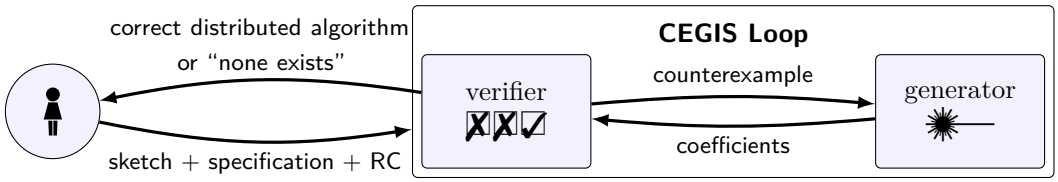

Figure 4 The synthesis loop implemented in this paper.

Existing model checking engine. The central idea of the verification approach in [19] is to formalize a distributed algorithm as counter system defined by a threshold automaton. A state of a counter system records how many processes are in which local state (e.g., $\ell_0, \ell_1, \ell_{\mathsf{SE}}, \ell_{\mathsf{AC}}$). A transition checks the guard and decreases or increases the related process counters. A transition can be written as a set of constraints in linear integer arithmetic LIA. (Threshold guards with rational coefficients, e.g., $\mathsf{echos} > \frac{n}{2}$, can be converted to integer constraints, e.g., $2 \cdot \mathsf{echos} > n$.) Thus, one can check for existence of specific executions by using SMT solvers, which extend SAT solvers (for Boolean satisfiability) with first-order theories, in our case, LIA. As shown in [20, 19], resilience conditions, executions of threshold-guarded distributed algorithms, and specifications can be encoded as logical formulas, whose satisfiability can be checked by solvers such as Z3 [11] and CVC4 [3]. In particular, the queries used in [20, 19] correspond to counterexamples to a specification: If the SMT solver finds all queries to be unsatisfiable, the distributed algorithm is correct. Otherwise, if a query is satisfiable, the SMT solver outputs a satisfying assignment, that is an error trace, called *counterexample*.

The synthesis approach of this paper. Figure 4 gives an overview of our method that takes as input (i) a sketch of a distributed algorithm, (ii) a set of safety and liveness specifications, and (iii) a resilience condition like $n > 3t$, and produces as output a correct distributed algorithm, or informs the user that none exist.

We follow the CEGIS approach to synthesis [1], which proceeds in a refinement loop. Roughly speaking, the verifier starts by picking default values for the missing coefficients — e.g., a vector of zeroes — and checks whether the algorithm is correct with these coefficients. Typically this is not the case and the *verifier* produces a counterexample. By automatically analyzing this counterexample, the *generator* learns constraints on the coefficients that are known to produce counterexamples. The generator gives these constraints to an SMT solver that generates new values for the coefficients, which are used in a new verifier run. If the verifier eventually reports that the current coefficients induce a correct distributed algorithm, we output this algorithm. The theory from [19] then implies correctness of the algorithm.

Termination of synthesis. The remaining theoretical problem that we address in this paper is termination of the refinement loop: In principle, the generator can produce infinitely many vectors of coefficients. In case there is no solution (which is typically the case in Byzantine fault tolerance if $n \leq 3t$), the naïve approach from the previous paragraph does not terminate, unless we restrict the guards to "reasonable" values. In this paper, we require the guards to lie in the interval $[0, n]$. We call such guards *sane*. For instance, although syntactically the expressions $\mathsf{echos} \leq -42n$ and $\mathsf{echos} > 2n$ are threshold guards, they are not sane, while $\mathsf{echos} \geq t + 1$ is sane. We mathematically prove that all sane guards of a specific structure have coefficients within a hyperrectangle. We call this hypperrectangle a *sanity box*, and prove that its boundaries depend only on the resilience condition. Within the sanity box,

there is only a finite number of coefficients, if we restrict them to integers or rationals with a fixed denominator. We thus obtain a finite search space and a completeness result for the synthesis loop.

Safety, liveness, and the fraction of faults. We consider the conjunction of *safety* and *liveness* specifications, as these specifications in isolation typically have trivial solutions; e.g., "do nothing" is always safe. If just given a safety specification, our tool generates thresholds like n for all guards, which leads to all guards evaluating to false initially. Hence, no action can ever be taken, which is a valid solution if liveness is not required.

Besides, our tool treats resilience conditions precisely. On the one hand, given the sketch from Figure 2, and the resilience condition $n > 3t$, in a few seconds our tool generates the threshold automaton in Figure 3. On the other hand, in the case of $n \geq 3t$, our tool reports (also within seconds) that no such algorithm exists, which in fact constitutes an automatically generated impossibility result for sane thresholds and a fixed sketch.

Experimental evaluation. We extended the tool ByMC [20] with our technique and conducted experiments based on the freely available benchmarks from [20]: folklore reliable broadcast [7], consistent broadcast [29, 30], and one-step Byzantine asynchronous consensus BOSCO [28]. For these benchmarks, we replaced the threshold guards by threshold guards with holes. By experimental evaluation, we show that our method can be used to generate coefficients even for quite intricate fault-tolerant distributed algorithms that tolerate Byzantine faults. In particular BOSCO proved to be a hard instance. It has to satisfy constraints derived from different safety an liveness specifications under different resilience conditions $n > 3t$, $n > 5t$, and $n > 7t$. Our tool is able to derive the three different threshold guards the algorithm requires. Finally, we give variations of specifications, and synthesize distributed algorithms from them that have not been produced before.

2 Modelling Threshold-Guarded Distributed Algorithms

Threshold-guarded algorithms are formalized by threshold automata. We recall the notions of threshold automata [19] and introduce the new concept of sketches. As usual, $\mathbb{N}_0$ is the set of natural numbers including 0, and $\mathbb{Q}$ is the set of rational numbers. The set Π is a finite set of *parameter variables* that range over $\mathbb{N}_0$. Typically, Π consists of three variables: n for the total number of processes, f for the number of actual faults in a run, and t for an upper bound on f. The parameter variables from Π are usually restricted to admissible combinations by a formula that is called a *resilience condition*, e.g., $n > 3t \wedge t \geq f \geq 0$. The set Γ is a finite set that contains *shared variables* that store the number of distinct messages sent by distinct (correct) processes, the variables in Γ also range over $\mathbb{N}_0$. In the example in Figure 3, $\Gamma = \{\mathsf{echos}\}$. For the variables from Γ, we will use names echos, x, y, etc.

For a set of variables V, a function $\nu : V \to \mathbb{Q}$ is called *an assignment*; its domain V is denoted with $dom(\nu)$. In this paper, we use Φ, Ψ, and Θ for first-order logic (FOL) formulas; e.g., when encoding linear integer constraints in SMT. For a FOL formula Φ, we write $free(\Phi)$ for the set of Φ's free variables, that is, the variables not bound with a quantifier. (For convenience, we assume that quantified variables have unique names and they are different from the names of the free variables.) Given an assignment $\nu : V \to \mathbb{Q}$ and a FOL formula Φ, we define a *substitution* $\Phi[\nu]$ as a FOL formula that is obtained from Φ by replacing all the variables from $V \cap free(\Phi)$ with their values in ν.

To introduce sketches of threshold automata — such as in Figure 2 — we define unknowns such as $\mathbf{?}_1$. The set U is a finite set of *unknowns* that range over $\mathbb{Q}$. For the variables from U, we use the names $\mathbf{?}_1$, $\mathbf{?}_2$, etc. We denote the rational values of unknowns with a, b, c, etc.

Generalized threshold guards, or just *guards*, are defined according to the grammar:

$$\begin{array}{ll} \mathit{Guard} ::= \mathit{Shared} \geq \mathit{LinForm} \mid \mathit{Shared} < \mathit{LinForm} & \mathit{Shared} ::= \langle \text{variable from } \Gamma \rangle \\ \mathit{LinForm} ::= \mathit{FreeCoeff} \mid \mathit{Prod} \mid \mathit{Prod} + \mathit{LinForm} & \mathit{Param} ::= \langle \text{a variable from } \Pi \rangle \\ \mathit{FreeCoeff} ::= \mathit{Rat} \mid \mathit{Unknown} & \mathit{Unknown} ::= \langle \text{a variable from } U \rangle \\ \mathit{Prod} ::= \mathit{Rat} \times \mathit{Param} \mid \mathit{Unknown} \times \mathit{Param} & \mathit{Rat} ::= \langle \text{a rational from } \mathbb{Q} \rangle \end{array}$$

For convenience, we assume that every parameter appears in *LinForm* at most once. Let $\bar{\pi}$ denote the vector $(\pi_1, \ldots, \pi_{|\Pi|}, 1)$ that contains all the parameter variables from Π in a fixed order as well as number 1 as the last element. Then, every generalized guard can be written in one of the two following forms $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, where x is a shared variable from Γ, and $\bar{u}$ is a vector of elements from $U \cup \mathbb{Q}$. When a parameter does not appear in a generalized guard, its corresponding component in $\bar{u}$ equals zero. We say that a guard is a *sketch guard* if its vector $\bar{u}$ contains a variable from U. A guard that is not a sketch guard is called a *fixed guard*. Previous work [19] was only concerned with fixed guards.

Since threshold guards are a special case of FOL formulas, we can apply substitutions to them. For instance, given an assignment $\nu : U \to \mathbb{Q}$ and a threshold guard g, the substitution $g[\nu]$ replaces every occurence of an unknown $\mathbf{?}_i \in U$ in g with the rational $\nu(\mathbf{?}_i)$.

Threshold automata, denoted by TA, are edge-labeled graphs, where vertices are called *locations*, and edges are called *rules*. Rules are labeled by $g \mapsto \texttt{act}$, where expression g is a fixed threshold guard, and the action `act` may increment a shared variable. We define *generalized threshold automata* GTA, in the same way as threshold automata, with the only difference that expressions g in the edge labeling are generalized threshold guards. If all generalized guards in a GTA are fixed, then that GTA is a TA. If at least one of the edges of a GTA is labeled by a sketch guard, then we call this automaton a *sketch threshold automaton*, and we denote it by STA. Given an STA and an assignment $\nu : U \to \mathbb{Q}$, we obtain a threshold automaton STA$[\nu]$ by applying substitution $g[\nu]$ to every sketch guard g in STA.

Counter systems. Executions of threshold automata are formalized as counter systems. Since processes just wait for messages until a threshold is reached and do not distinguish the senders, the systems we consider are symmetric. This allows us to represent a global state — a configuration — by (process) counters: Instead of recording which process is in which local state (which is done typically in distributed algorithms theory), we capture for each local state, how many processes are in it, and then use the rules of the threshold automaton to define the transitions between configurations. In the following, we quickly sketch the semantics to the extent necessary for this paper. Complete definitions can be found in [19].

For every TA we define a counter system as a transition system. First, for every location ℓ we introduce a counter $\boldsymbol{\kappa}[\ell]$ that keeps track of the number of processes in that particular location. A configuration σ is defined as an assignment of all counters of locations, all shared variables from Γ, and all parameters from Π, that respects the resilience condition. If a rule r is an edge (ℓ, ℓ') of a TA, then a transition (r, m) represents m processes moving from the location ℓ to ℓ'. We call m the *acceleration factor*. If $m = 1$ for all transitions in an execution, we get asynchronous executions where one process moves at a time, that is, interleaving semantics. If the rule r has a label $g \mapsto \texttt{act}$, then (r, m) can be applied only in

a configuration in which the counter $\boldsymbol{\kappa}[\ell]$ has a value at least m, and g evaluates to true, and remains true during $m-1$ applications of `act`. In other words, only if the threshold from g is reached, and there are enough processes in location ℓ; see [19] for details. After executing the transition (r, m), counters are updated such that $\boldsymbol{\kappa}[\ell]$ is decreased by m and $\boldsymbol{\kappa}[\ell']$ is increased by m, and shared variables are updated according to the action `act`, m times.

▶ **Example 2.** Consider the TA from Figure 3. One configuration is the following: parameters are $n = 7$, $f = t = 2$, satisfying resilience condition $n > 3t \geq 0 \wedge f \leq t$, counters have values $\boldsymbol{\kappa}[\ell_0] = 2$, $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 3$, $\boldsymbol{\kappa}[\ell_1] = \boldsymbol{\kappa}[\ell_{\mathsf{AC}}] = 0$ and shared variable $\mathsf{echos} = 3$. (As we only model correct process explicitly, the counters add up to $n - f = 5$.) As in this configuration we have that $\mathsf{echos} + f = 5 \geq 5 = n - t$, and $\boldsymbol{\kappa}[\ell_0] = 2$, we can execute transition $(r_2, 2)$. The obtained configuration has the same parameter values, but counters are changed: $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 3$, and $\boldsymbol{\kappa}[\ell_{\mathsf{AC}}] = 2$. Also, as the action of the rule r_2 is $\mathsf{echos}{+}{+}$, and two processes are moving along this edge, then the new value of echos is 5. ◁

With a TA we associate a set of predicates $\mathcal{P}_{\mathsf{TA}}$ that track properties of the system states. The set $\mathcal{P}_{\mathsf{TA}}$ consists of the TA's threshold guards and a test $\boldsymbol{\kappa}[\ell] = 0$ for every location ℓ in TA. For every configuration σ, one can compute the set $\rho(\sigma) \subseteq \mathcal{P}_{\mathsf{TA}}$ of the predicates that hold true in σ. As was demonstrated in [19], the predicates from $\mathcal{P}_{\mathsf{TA}}$ and linear temporal logic are sufficient to express the safety and liveness properties of threshold-guarded distributed algorithms found in the literature. Essentially, the test $\boldsymbol{\kappa}[\ell] = 0$ evaluates to true if no process is in location ℓ, and $\boldsymbol{\kappa}[\ell] \neq 0$ evaluates to true if there is at least one process at ℓ. That all processes are in specific locations can be expressed by a condition that states that "no processes are in the other locations", that is, as a Boolean combination of tests for zero. We include the threshold guards in $\mathcal{P}_{\mathsf{TA}}$ to be able to express the fairness properties such as: if $\mathsf{echos} \geq t + 1$, then every process should eventually make one of the transitions labelled with $\mathsf{echos} + f \geq t + 1$. Examples of such properties for our benchmarks are given in Section 5.

A system execution is expressed as a path in the counter system. Formally, a *path* is an infinite alternating sequence of configurations and transitions, that is, $\sigma_0, t_1, \sigma_1, \ldots, t_i, \sigma_i, \ldots$, where σ_0 is an initial configuration, and σ_{i+1} is the result of applying t_{i+1} to σ_i for $i \geq 0$. The infinite sequence $\rho(\sigma_0), \rho(\sigma_1), \ldots$ is called the path *trace*. With $\mathsf{Traces}_{\mathsf{TA}}$ we denote the set of all path traces in the TA's counter system. Correctness of a distributed algorithm then means that all traces in $\mathsf{Traces}_{\mathsf{TA}}$ satisfy a specification expressed in linear temporal logic [10]. The verification approach from [19] discussed in Section 3 specifically looks for traces that violate the specification. Such traces are characterized by the temporal logic $\mathsf{ELTL}_{\mathsf{FT}}$ that allows one to express *negations of specifications* relevant for fault-tolerant distributed algorithms.

3 Verification machinery

In [19] we introduced a technique for parameterized verification of threshold-based distributed algorithms. Given a fixed threshold automaton TA, a resilience condition RC, and a set $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$ of $\mathsf{ELTL}_{\mathsf{FT}}$ formulas representing negation of specifications, we check whether there is an execution violating the specification $(\varphi_1 \wedge \ldots \wedge \varphi_k)$. Thus, as an output, the algorithm from [19] either confirms correctness, or gives a counterexample. In this paper, we use this technique as a black box, that is, we assume that there is a function

$$\mathsf{verify}_{\mathsf{ByMC}}(\mathsf{TA}, RC, \{\neg\varphi_1, \ldots, \neg\varphi_k\})$$

that either reports a counterexample, or that TA is correct. As our synthesis approach learns from counterexamples, we recall the form of the counterexamples reported by the verifier.

Representative executions and schemas. The idea of [19] lies in automatically computing length and structure of representative executions in advance, whose shape we call schemas. A *schema* is an alternating sequence of contexts (sets of guards) and sequences of rules. Precise definition of a schema and its encoding can be found in [20, 19]. Intuitively, a schema is a concatenation of multiple *simple* schemas. A simple schema has the form $\{g_1, \ldots, g_k\}\, r_1 \ldots r_s\, \{g'_1, \ldots, g'_{k'}\}$, where $k, k', s \in \mathbb{N}$, $g_1, \ldots, g_k, g'_1, \ldots, g'_{k'}$ are guards, and $r_1, \ldots, r_s$ are rules. Given acceleration factors m_i, $1 \leq i \leq s$, such that $0 \leq m_i \leq n$, the simple schema generates an execution where all the guards $g_1, \ldots, g_k$ hold in its initial configuration, and after executing $(r_1, m_1), \ldots, (r_s, m_s)$, we arrive in a configuration where all the guards $g'_1, \ldots, g'_{k'}$ hold. As proven in [19], specific schemas of fixed length represent infinite executions (that end in an infinite loop) as required for counterexamples to liveness.

Our verification tool considers each schema in isolation, and basically searches for an evaluation ν of the parameters $(n,\ t,\ f)$, an initial configuration (values of counters of initial local states), and all the acceleration factors, such that the resulting execution is admissible (only enabled rules are executed, etc.). Such an execution — if found — constitutes a counterexample, and the tool reports the corresponding pair $(\mathsf{schema},\ \nu)$.

Solver. Our tool encodes a schema as an SMT formula over parameters, counters of local states, global variables, and acceleration factors. This formula is a conjunction of equalities and inequalities in linear integer arithmetic. Inequalities come from guards, and have shared variables and parameters as free variables. Equalities come from transitions, as every transition is encoded as updating counters of local states and shared variables. The tool ByMC [19] calls an SMT solver to check satisfiability of the formula.

4 Synthesis

Synthesis problem. A temporal logic formula φ in $\mathsf{ELTL_{FT}}$ describes an (infinite) set of bad traces that the synthesized algorithm must avoid. Therefore, we consider the following formulation of the *synthesis problem*. Given a sketch threshold automaton STA and an (infinite) set of bad traces $\mathsf{Traces_{Bad}}$, either:

- find an assignment $\mu : U \to \mathbb{Q}$, in order to obtain the fixed threshold automaton $\mathsf{STA}[\mu]$ whose traces $\mathsf{Traces_{STA}}[\mu]$ *do not intersect* with $\mathsf{Traces_{Bad}}$, or
- report that no such assignment exists.

Our approach is to find values for the unknowns in a synthesis refinement loop and test them with the verification technique from Section 3.

Synthesis loop. Figure 5 shows the pseudo-code of the synthesis procedure $\mathsf{synt_{ByMC}}$. At its input the procedure receives a sketch threshold automaton STA, a resilience condition, and a set of $\mathsf{ELTL_{FT}}$ formulas $\{\neg\varphi_1, \ldots, \neg\varphi_k\}$, which capture the bad traces $\mathsf{Traces_{Bad}}$. In line 2, formula Θ_0, which captures constraints on the unknowns from U, is initialized using a function $\mathtt{bound}_U$. In principle, $\mathtt{bound}_U$ can be initialized to true (no constraints). However, to ensure termination, we will discuss later in this section, how we obtain constraints that bound the coefficients of sane guards. After initialization we enter the synthesis loop.

The SMT solver checks whether Θ_i has a satisfying assignment to the unknowns in U (line 4). If Θ_i is unsatisfiable, the loop terminates with a *negative outcome* in line 5. Otherwise, the SMT solver gives us an assignment $\mu : U \to \mathbb{Q}$ that is a solution candidate. To check feasibility of μ, the verifier is called for the fixed threshold automaton $\mathsf{STA}[\mu]$ in line 7. The verifier generates multiple schemas, each being one SMT query, which are checked either

```
1  procedure synt_ByMC(STA, RC, {¬φ_1, ..., ¬φ_k})
2    Θ_0 := bound_U(RC) and i := 0
3    while (true)
4      call check_SMT(Θ_i)
5      case unsat ⇒ print 'no more solutions' and exit()
6      case sat(μ) ⇒ /* μ assigns rationals to the variables in U */
7        call verify_ByMC(STA[μ], RC, {¬φ_1, ..., ¬φ_k})
8        case correct ⇒
9          print 'solution μ' /* exclude this solution and continue */
10         Θ_{i+1} := Θ_i ∧ ⋁_{?_j ∈ U} ?_j ≠ μ[?_j] and i := i+1
11       case counterexample(S, ν) ⇒ /* dom(ν) ∩ U = ∅ */
12         S_U := generalize(S, STA)
13         Ψ := formula_SMT(S_U)
14         Θ_{i+1} := Θ_i ∧ ¬Ψ[ν] and i := i+1
```

Figure 5 Pseudo-code of the synthesis loop

sequentially or in parallel. If the verifier reports that a schema that produces a counterexample does not exist, then the candidate assignment μ and threshold automaton $\mathsf{STA}[\mu]$ give us a solution to the synthesis problem. If we were interested in just one solution, the loop would terminate here with a *positive outcome*. However, because we want to enumerate all solutions, our function does a complete search, such that we exclude the solution μ for the future search in line 10, and continue.

If the verifier finds a counterexample, the loop proceeds with the branch in line 11. A counterexample is a schema S of $\mathsf{STA}[\mu]$ and a satisfying assignment $\nu : V \to \mathbb{Q}$ to the free variables V of the SMT formula $\mathsf{formula}_{\mathsf{SMT}}$, which include the parameters Π, shared variables x^j for $x \in \Gamma$, and counters $\boldsymbol{\kappa}^j[\ell]$ for each local state $\ell \in \mathcal{L}$ and every configuration j. In principle, we could exclude μ from consideration similar to line 10. For efficiency, we want to exclude a larger set of evaluations, namely all that lead to the same counterexample: We produce a *generalized* schema S_U, by replacing the rules and guards in S, which belong to the threshold automaton $\mathsf{STA}[\mu]$ with the rules and guards of the sketch threshold automaton STA (line 12). In line 13, we generate a generalized counterexample Ψ. As Ψ is derived from a counterexample with valuations μ and ν, we know that $\Psi[\nu][\mu]$ is true. Further, for every evaluation of the unknowns μ', if $\Psi[\nu][\mu']$ is true, then $\Psi[\nu][\mu']$ is a counterexample. To exclude all these evaluations μ' at once, we conjoin $\neg\Psi[\nu]$ with Θ_i in line 14, which gives us new constraints on the unknowns, before entering the next loop iteration.

The synthesis loop terminates only in line 5, that is, if Θ_i is unsatisfiable. As, in this case, Θ_i is equivalent to false, the following observation guarantees that all satisfying assignments of Θ_0 have been explored and all solutions (if any exists) have been reported.

▶ Observation 1. At the beginning of every iteration $i \geq 0$ of the synthesis loop in lines 3–14, the following invariant holds: if $\mu : U \to \mathbb{Q}$ is a satisfying assignment of formula $\Theta_0 \wedge \neg\Theta_i$, then either: (1) μ was previously reported as a solution in line 9, or (2) μ was previously excluded in line 14 and thus is not a solution. ◁

Completeness and termination for sane guards. Without restricting Θ_0, the search space for coefficients is infinite. In the following, we show that restricting the synthesis problem to sane guards bounds the search space.

The role of threshold guards is typically to check whether the number of distinct senders, from which messages are received, reaches a threshold. We also use threshold guards in our

models to bound the number of processes that go into a special crash state. In both cases, one counts distinct processes and it is therefore natural to consider only those thresholds whose value is in $[0, n]$. More precisely, if the guard has a form $x \geq \bar{u} \cdot \bar{\pi}^\intercal$ or $x < \bar{u} \cdot \bar{\pi}^\intercal$, then for all parameter values that satisfy resilience condition it holds that $0 \leq \bar{u} \cdot \bar{\pi}^\intercal \leq n$. We call such guards *sane* for a given resilience condition.

Theorem 3 considers a general case of hybrid failure models [30] where different failure bounds exist for different failure models (e.g., t_1 Byzantine faults and t_2 crash faults), and these failure bounds are related to the number of processes n by a resilience condition[1] of the form $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$. We bound the values of the coefficients of sane guards.

▶ **Theorem 3.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, for $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{or} \quad x < an + (b_1 t_1 + \ldots + b_k t_k) + c,$$

where $x \in \Gamma$, and $a, b_1, \ldots, b_k, c \in \mathbb{Q}$. If the guard is sane for the resilience condition, then

$$0 \leq\ a\ \leq 1, \tag{1}$$

$$-\delta_i - 1 <\ b_i\ < \delta_i + 1,\ \textit{for all } i = 1, \ldots, k \tag{2}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq\ c\ \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{3}$$

The case when $k = 1$ gives us the classical resilience condition where the system model assumes *one* type of faults (e.g., crash), and the assumed number of faults t is related to the total number of processes n, by a condition $n > \delta t \geq 0$ for some $\delta > 0$. If the guard that compares a shared variable and $an + bt + c$ is sane for the resilience condition, then we obtain that $0 \leq a \leq 1$, $-\delta - 1 < b < \delta + 1$, and $-2\delta - 2 \leq c \leq 2\delta + 2$. Any restriction of the intervals from Theorem 3 to finite sets gives us completeness: If we reduce the domain of variables from U to integers, or to rationals with fixed denominator (e.g., $\frac{z}{10}$ for $z \in \mathbb{Z}$), one reduces the search space to a finite set of valuations. All threshold-based distributed algorithms we are aware of, use guards with coefficients that are either integers or rationals with a denominator not greater than 3. Thus, we restrict our intervals by intersecting them with the set of rational numbers whose denominator is at most D, for a given $D \in \mathbb{N}$.

The following corollary is a direct consequence of Theorem 3, and it tells us how to modify intervals if the coefficients are rational numbers with a fixed denominator.

▶ **Corollary 4.** *Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i.\ t_i \geq 0$ be a resilience condition, where $k \in \mathbb{N}$, $\delta_i \in \mathbb{Q}$ and $\delta_i > 0$, $1 \leq i \leq k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard*

$$x \geq \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k\right) + \frac{\tilde{c}}{D} \quad \text{or} \quad x < \frac{\tilde{a}}{D} n + \left(\frac{\tilde{b_1}}{D} t_1 + \ldots + \frac{\tilde{b_k}}{D} t_k\right) + \frac{\tilde{c}}{D},$$

where $x \in \Gamma$, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$, $D \in \mathbb{N}$. If the guard is sane for the resilience condition then

$$0 \leq\ \tilde{a}\ \leq D, \tag{4}$$

$$D(-\delta_i - 1) <\ \tilde{b_i}\ < D(\delta_i + 1),\ \textit{for all } i = 1, \ldots, k, \tag{5}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 1) \leq\ \tilde{c}\ \leq D(2(\delta_1 + \ldots + \delta_k) + k + 1). \tag{6}$$

[1] Because a guard that is sane for a weaker resilience condition, is also sane for a stronger one, Theorem 3 and Corollary 4 also hold for any resilience condition that follows from this one, e.g., $n > \max\{\delta_1 t_1, \ldots, \delta_k t_k\} \wedge \forall i.\ t_i \geq 0$. We can use the same intervals, confirmed by the same proofs as in Appendix A. However, our benchmarks use the form of resilience conditions of Theorem 3.

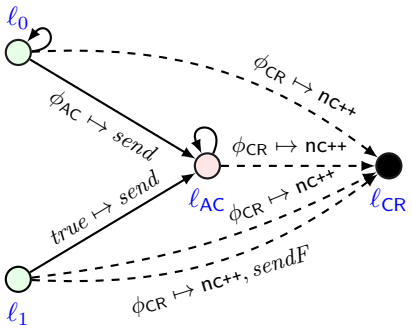

Figure 6 A sketch threshold automaton for folklore reliable broadcast

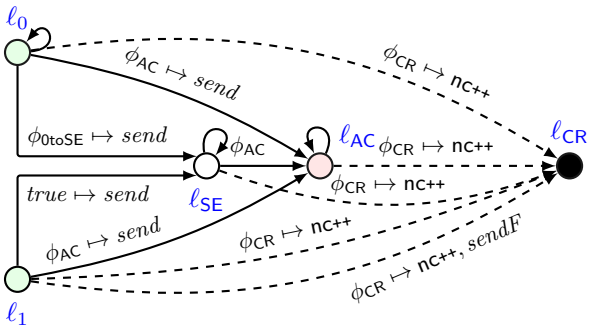

Figure 7 A sketch threshold automaton for reliable broadcast with Byzantine and crash faults

Constraints (4)–(6) constitute the sanity box that function bound_U computes in Figure 5. By fixing D, we restrict Θ_0 to have finitely many satisfying assignments (integers). Hence, the loop terminates. Statements similar to Theorem 3 and Corollary 4 can be derived for other forms of threshold guards, e.g., for thresholds with floor or ceiling functions.[2]

5 Case Studies and Experiments

We have extended ByMC [20, 19] with the synthesis technique presented in this paper. A virtual machine with the tool and the benchmarks is available from: `http://forsyte.at/software/bymc`.[3] ByMC is written in OCaml and uses Z3 [11] as a backend SMT solver. We ran the experiments on two systems: a laptop and the Vienna Scientific Cluster (VSC-3). The laptop is equipped with 16 GB of RAM and Intel® Core™ i5-6300U processor with 4 cores, 2.4 GHz. The cluster VSC-3 consists of 2020 nodes, each equipped with 64 GB of RAM and 2 processors (Intel® Xeon™ E5-2650v2, 2.6 GHz, 8 cores) and is internally connected with an Intel QDR-80 dual-link high-speed InfiniBand fabric: `http://vsc.ac.at`.

We synthesize thresholds for asynchronous fault-tolerant distributed algorithms. We consider *reliable broadcast* and *fast decision* for a consensus algorithm. In the case of reliable broadcast we consider different fault models, namely, crashes [7] and Byzantine faults [29], as well as a hybrid fault model [30] with both, Byzantine and crash failures. For fast decision, we consider the one-step consensus algorithm BOSCO for Byzantine faults [28].

Reliable broadcast for crash and/or Byzantine failures. Figure 7 shows a sketch threshold automaton of a reliable broadcast that should tolerate $f_c \leq t_c$ crash and $f_b \leq t_b$ Byzantine faults under the resilience condition $n > 3t_b + 2t_c$. For our experiments under simpler failure models — only Byzantine and crash faults — we use the sketch threshold automata from Figures 2 and 6. However, the same thresholds can be obtained by setting $t_c = f_c = 0$ and $t_b = f_b = 0$ in the automaton from Figure 7, respectively. In Figure 2, we do not need a dedicated crash state, as we only model correct processes explicitly, while Byzantine faults are modeled via the guards (cf. Example 1). The automaton from Figure 6 can be obtained from Figure 7 by removing the location ℓ_{SE}.

[2] Theorem 6 and Corollary 7 in Appendix B consider floor and ceiling functions. Our benchmarks do not make use of such thresholds.

[3] See `http://forsyte.at/opodis17-artifact/` for detailed instructions on using the tool.

Table 1 Synthesized solutions for reliable broadcast that tolerates: crashes (Figure 6), Byzantine faults (Figure 2), and Byzantine & crash faults (Figure 7). We used the laptop in the experiments.

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > t_c,\ t_b = 0$	U, C, R	1	*true*	1	12	6
$n > 3t_b,\ t_c = 0$	U, C, R	3	$n - 2t_b$ $t_b + 1$ $t_b + 1$	$n - t_b$ $2t_b + 1$ $n - t_b$	31	16
$n \geq 3t_b,\ t_c = 0$	U, C, R	None	—	—	25	7
$n > 3t_b + 2t_c$	U, C, R	3	$n - 2t_b - 2t_c$ $t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c$ $n - t_b - t_c$	34	50
$n \geq 3t_b + 2t_c$	U, C, R	None	—	—	21	12
$n > 3t_b + t_c$	U, C, R	None	—	—	29	24

The algorithms we consider are the core of broadcasting algorithms, and establish agreement on whether to accept the message by the broadcaster. Similar to Example 1, processes start in locations ℓ_1 and ℓ_0, which capture that the process has received and has not received a message by the broadcaster, respectively. A correctly designed algorithm should satisfy the following properties [29]:

(U) *Unforgeability:* If no correct process starts in ℓ_1, then no correct process ever enters ℓ_{AC}.

(C) *Correctness:* If all correct processes start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

(R) *Relay:* Whenever a correct process enters ℓ_{AC}, all correct processes eventually enter ℓ_{AC}.

In the following discussion we use Figure 7 as example. We have to sketch the guards ϕ_{CR}, ϕ_{0toSE}, and ϕ_{AC}. At most f_c processes can move to the crashed state ℓ_{CR}. The algorithm designer does not have control over the crashes, and thus we fix the guard ϕ_{CR} to be $\mathsf{nc} < f_c$: The shared variable nc maintains the actual number of crashes (initially zero), which is used only to model crashes and thus cannot be used in guards other than ϕ_{CR}. To properly model that a processes can crash during a "send to all" operation (*non-clean crash*), we introduce two shared variables: the variable echos stores the number of echo messages that are sent by the correct processes (some of them may crash later), and the variable $\mathsf{echosCF}$ stores the number of echo messages that are sent by the correct processes and the faulty processes when crashing. Hence, the action *send* increases both echos and $\mathsf{echosCF}$, whereas the action *sendF* increases only $\mathsf{echosCF}$.

We define the thresholds τ_{0toSE} and τ_{AC} as $(\mathbf{?}_a^{\mathsf{SE}} \cdot n + \mathbf{?}_b^{\mathsf{SE}} \cdot t_b + \mathbf{?}_c^{\mathsf{SE}} \cdot t_c + \mathbf{?}_d^{\mathsf{SE}})$ and $(\mathbf{?}_a^{\mathsf{AC}} \cdot n + \mathbf{?}_b^{\mathsf{AC}} \cdot t_b + \mathbf{?}_c^{\mathsf{AC}} \cdot t_c + \mathbf{?}_d^{\mathsf{AC}})$ respectively. Hence, ϕ_{0toSE} and ϕ_{AC} are defined as $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{0toSE}}$ and $\mathsf{echosCF} + f_b \geq \tau_{\mathsf{AC}}$. As discussed in the introduction, we add f_b to $\mathsf{echosCF}$ to reflect that the correct processes may — although do not have to — receive messages from Byzantine processes. For *reliable communication*, we have to enforce:

Every correct process eventually receives at least echos *messages.* (RelComm)

As threshold automata do not explicitly store the number of received messages, we transform (RelComm) into a fairness constraint, which forces processes to eventually leave a location if the messages by correct processes alone enable a guard of an edge that is outgoing from this location. That is, *there is a time after which the following holds forever:*

$$\boldsymbol{\kappa}[\ell_1] = 0 \wedge (\mathsf{echos} < \tau_{\mathsf{0toSE}} \vee \boldsymbol{\kappa}[\ell_0] = 0) \wedge (\mathsf{echos} < \tau_{\mathsf{AC}} \vee (\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_{\mathsf{SE}}] = 0)). \quad \text{(Fair)}$$

Table 2 Synthesized solutions for variations of reliable broadcast and specifications (X)–(Z).

Resilience condition	Specs	#Solutions	Threshold τ_{0toSE}	Threshold τ_{AC}	Calls to verifier	Time, seconds
$n > 3t_b,\ t_c = 0$	X, C, R	None	—	—	15	2
$n > 3t_b + 2,\ t_c = 0$	X, C, R	3	$n - 2t_b$ $t_b + 3$ $t_b + 3$	$n - t_b$ $2t_b + 3$ $n - t_b$	35	12
$n > 3t_b,\ t_c = 0$	Y, C, R	None	—	—	28	6
$n > 4t_b,\ t_c = 0$	Y, C, R	3	$n - 2t_b$ $2t_b + 1$ $2t_b + 1$	$n - t_b$ $3t_b + 1$ $n - t_b$	33	12
$n > 3t_b + 2t_c$	U, Z, R	2	$t_b + 1$ $t_b + 1$	$n - t_b - t_c$ $2t_b + t_c + 1$	41	31

Table 1 summarizes the experimental results for reliable broadcast, when looking for integer solutions only. The cases $t_b = 0$ and $t_c = 0$ correspond to the algorithms that tolerate only crashes (Figure 6) and only Byzantine faults (Figure 2) respectively. For these cases, we obtained the solutions known from the literature [29, 7] and some variations. Moreover, when the resilience condition is changed from $n > 3t_b$ to $n \geq 3t_b$, our tool reports no solution, which also complies with the literature [29]. In the case of f_c crashes and f_b Byzantine faults, the tool reports three solutions. Moreover, when we tried to relax the resilience condition to $n \geq 3t_b + 2t_c$ and $n > 3t_b + t_c$, the tool reported that there is no solution, as expected.

Variations of the specification. Our logic allows us to easily change the specifications. For instance, we can replace the precondition of unforgeability "if *no* correct process starts in ℓ_1" by giving an upper bound (number or parameter) on correct processes starting in ℓ_1 that still prevents entering ℓ_{AC}, in specifications (X) and (Y). We also changed the precondition of correctnesss "if *all* correct processes start in ℓ_1" in specification (Z):

(X) If *at most two* correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.

(Y) If *at most* t_b correct processes start in ℓ_1, then no correct process ever enters ℓ_{AC}.

(Z) If *at least* $t_b + t_c + 1$ non-Byzantine processes (correct or crash faulty) start in ℓ_1, then there exists a correct process that eventually enters ℓ_{AC}.

Interestingly, we obtain new distributed computing problems that put quantitative conditions on the initial state. These specifications are related to the specifications of condition-based consensus [27]. Our tool automatically generates solutions, or shows their absence in the case resilience conditions are too strong. Table 2 summarizes these results.

Byzantine one-step consensus. Figure 8 shows a sketch threshold automaton of a one-step Byzantine consensus algorithm that should tolerate $f \leq t$ Byzantine faults under the assumption $n > 3t$. It is a formalization of the BOSCO algorithm [28]. The purpose of the algorithm is to quickly reach consensus if (a) $n > 5t$ and $f = 0$, or (b) $n > 7t$. In this encoding, correct processes make a "fast" decision on 0 or 1 by going in the locations ℓ_{D0} and ℓ_{D1}, respectively. When neither (a) nor (b) holds, the processes precompute their votes in the first step and then go to the locations ℓ_{U0} and ℓ_{U1}, from which an *underlying consensus* algorithm is taking over. In this sense, BOSCO can be seen as an asynchronous preprocessing step for general consensus algorithms, and the properties given below contain preconditions

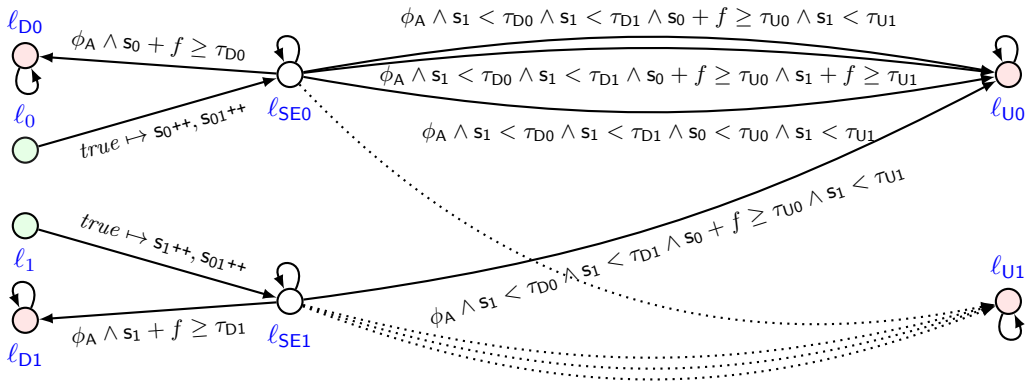

Figure 8 A sketch threshold automaton for one-step Byzantine consensus. Labels of dashed edges are omitted; they can be obtained from the respective solid edges by swapping 0 and 1.

for calling consensus in a safe way (see Fast Agreement below). Every run of a synthesized threshold automaton must satisfy the following properties (for $i \in \{0,1\}$ and $j = 1 - i$):

(A) *Fast agreement [28, Lemmas 3–4]:* Condition $\boldsymbol{\kappa}[\ell_{Di}] \neq 0$ implies $\boldsymbol{\kappa}[\ell_{Dj}] = \boldsymbol{\kappa}[\ell_{Uj}] = 0$.

(O) *One step:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it always holds that $\boldsymbol{\kappa}[\ell_{Dj}] = 0$ and $\boldsymbol{\kappa}[\ell_{U0}] = \boldsymbol{\kappa}[\ell_{U1}] = 0$. That is, the underlying consensus is never called.

(F) *Fast termination:* If $n > 5t \wedge f = 0$ or $n > 7t$, and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

(T) *Termination:* It eventually holds that $\boldsymbol{\kappa}[\ell_0] = \boldsymbol{\kappa}[\ell_1] = 0$ and $\boldsymbol{\kappa}[\ell_{SE0}] = \boldsymbol{\kappa}[\ell_{SE1}] = 0$.

We define thresholds τ_A, τ_{D0}, τ_{D1}, τ_{U0}, τ_{U1} as $\mathbf{?}_a^x \cdot n + \mathbf{?}_b^x \cdot t + \mathbf{?}_c^x$ for $x \in \{A, D0, D1, U0, U1\}$. Then, the guard ϕ_A is defined as: $s_{01} + f \geq \tau_A$. Interestingly, the thresholds appear in different roles in the guards, e.g., $s_0 + f \geq \tau_{D0}$ and $s_0 < \tau_{D0}$. These cases correspond to BOSCO's decisions on how many messages *have been received* and how many messages *have not been received* "modulo Byzantine faults."

As with reliable broadcast, we model reliable communication with the following fairness constraint: For $i \in \{0,1\}$, *from some point on*, the following holds: $\boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0 \wedge (s_{01} < \tau_A \vee s_i < \tau_{Di} \vee \boldsymbol{\kappa}[\ell_{SEi}] = 0)$.

We bound *denominators of rationals with two* and use the sanity box provided by Corollary 4. To reduce the search space, we assume that the guards for 0 and 1 are *symmetric*, that is $\mathbf{?}_a^{D0} = \mathbf{?}_a^{D1}$ and $\mathbf{?}_a^{U0} = \mathbf{?}_a^{U1}$. Still, BOSCO is a challenging benchmark for verification [19] and synthesis. Since the verification procedure from Section 3 independently checks schemas with SMT, we *parallelized* schema checking with OpenMPI, and ran the experiments at Vienna Scientific Cluster (VSC-3) using 8–128 cores; Table 3 summarizes the results. The tool has found four solutions for the guards: $\tau_A = n - t$ $[-\frac{1}{2}]$, $\tau_{D0} = \tau_{D1} = \frac{n+3t+1}{2}$, and $\tau_{U0} = \tau_{U1} = \frac{n-t}{2}$ $[+\frac{1}{2}]$. In addition to the guards from [28], the tool also reported that one can add or subtract ½ from several guards. Figure 9 demonstrates that increasing the number of cores above 64 slows down synthesis times for this benchmark.

Variations of the BOSCO specifications. We relaxed the precondition for fast termination:

(U) If $n \geq 5t \wedge f = 0$ and initially $\boldsymbol{\kappa}[\ell_j] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

Specs	Nr. of solutions	Calls to verifier	Nr. of cores	Time min.
AOFT	4	516	128	39
AOFT	4	432	96	25
AOFT	4	425	64	24
AOFT	4	502	16	44
AOFT	4	440	8	51
AOUT	0	376	8	40
AOVT	0	337	8	33

Table 3 Experiments for one-step Byzantine consensus for $n > 3t$ running the parallel verifier at VSC-3

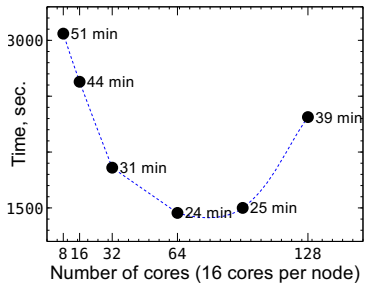

Figure 9 Synthesis times for BOSCO at Vienna Scientific Cluster (VSC-3)

(V) If $n \geq 7t$ and initially $\boldsymbol{\kappa}[\ell_{\mathsf{I}}] = 0$, then it eventually holds that $\boldsymbol{\kappa}[\ell] = 0$ for all local states different from ℓ_{Di}.

As can be seen from Table 3, specifications (U) and (V) have no solutions.

6 Discussions

The classic approach to establish correctness of a distributed algorithm is to start with a system model, a specification, and pseudo code, all given in natural language and mathematical definitions, and then write a manual proof that confirms that "all fits together." Manual correctness proofs mix code inspection, system assumptions, and reasoning about events in the past and the future. Slight modifications to the system assumptions or the code require us to redo the proof. Thus, the proofs often just establish correctness of the algorithm, rather than deriving details of the algorithm — like the threshold guards — from the system assumption or the specification.

We introduced an automated method that synthesizes a correct distributed algorithm from the specifications and the basic assumptions. Our tool computes threshold expressions from the resilience condition and the specification, by learning the constraints that are derived from counterexamples. Learning dramatically reduces the number of verifier calls. In case of BOSCO, the sanity box contains 2^{36} vectors of unknowns, which makes exhaustive search impractical, while our technique only needs to check approximately 500 vectors.

In addition to synthesizing known algorithms from the literature, we considered several modified specifications. For some of them, our tool synthesizes thresholds, while for others it reports that no algorithm of a specific form exists. The latter results are indeed impossibility results (lower bounds on the fraction of correct processes) for fixed sketch threshold automata.

To ensure termination of the synthesis loop, we restrict the search space, and thus the class of algorithms for which the impossibility result formally applies. First, while we restrict the search to sane guards, the same synthesis loop can also be used to synthesize other guards. However, in order to ensure termination, a suitable characterization of sought-after guards should be provided by the user. Second, for reliable broadcast we consider only threshold guards with integer coefficients that can express thresholds like $n - t$ or $2t + 1$. For BOSCO, we only allow division by 2, and can express thresholds like $\frac{n}{2}$ or $\frac{n-t}{2}$. While from a theoretical viewpoint these restrictions limit the scope of our results, we are not aware of a distributed algorithm where processes wait for messages from, say, $\frac{n}{7}$ or $\frac{n}{1000}$ processes. To strengthen our completeness claim, we would need to formally explain why only small denominators are used in fault-tolerant distributed algorithms.

References

1 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

2 K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

3 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, pages 171–177, 2011.

4 Benjamin Bisping, Paul-David Brodmann, Tim Jungnickel, Christina Rickmann, Henning Seidler, Anke Stüber, Arno Wilhelm-Weidner, Kirstin Peters, and Uwe Nestmann. A constructive proof for FLP. *Archive of Formal Proofs*, 2016.

5 Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and Byzantine-resilient distributed systems. In *CAV*, volume 9779 of *LNCS*, pages 157–176, 2016.

6 Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

7 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

8 Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

9 Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.

10 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

11 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.

12 Danny Dolev, Keijo Heljanko, Matti Järvisalo, Janne H. Korhonen, Christoph Lenzen, Joel Rybicki, Jukka Suomela, and Siert Wieringa. Synchronous counting and computational algorithm design. *J. Comput. Syst. Sci.*, 82(2):310–332, 2016.

13 Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

14 Fathiyeh Faghih and Borzoo Bonakdarpour. SMT-based synthesis of distributed self-stabilizing systems. *TAAS*, 10(3):21:1–21:26, 2015.

15 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *FORTE*, volume 9688 of *LNCS*, pages 124–141, 2016.

16 Adrià Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *NFM*, volume 8430 of *LNCS*, pages 270–284. Springer, 2014.

17 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.

18 Swen Jacobs and Roderick Bloem. Parameterized synthesis. *LMCS*, 10(1:12), 2014.

19 Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In *POPL*, pages 719–734, 2017.

20 Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

21 Igor Konnov, Josef Widder, Francesco Spegni, and Luca Spalazzi. Accuracy of message counting abstraction in fault-tolerant distributed algorithms. In *VMCAI*, pages 347–366, 2017.

22 Leslie Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002.

23 Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

24 Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

25 Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV*, pages 217–237, 2017.

26 Laure Millet, Maria Potop-Butucaru, Nathalie Sznajder, and Sébastien Tixeuil. On the synthesis of mobile robots algorithms: The case of ring gathering. In *SSS*, volume 8756 of *LNCS*, pages 237–251, 2014.

27 Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

28 Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

29 T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

30 Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Dist. Comp.*, 20(2):115–140, 2007.

31 James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

APPENDIX

A Detailed Proofs

In order to prove Theorem 3, we first prove mathematical background of it, i.e., Lemma 5.

▶ **Lemma 5.** *Fix a $k \in \mathbb{N}$, and for every $i \in \{1, \ldots, k\}$ fix $\delta_i > 0$. Let $a, b_1, \ldots, b_k, c$ be rationals for which the following holds: for every $n, t_1, \ldots, t_k \in \mathbb{N}$ such that $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$, it holds that $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$. Then it is the case that*

$$0 \leq a \leq 1, \tag{7}$$

$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \ldots, k \tag{8}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 1 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 1. \tag{9}$$

Proof. Let $\mathbf{P}_{RC}$ be the set of all tuples $(n, t_1, \ldots, t_k) \in \mathbb{N}^{k+1}$ that satisfy $n > \sum_{i=1}^{k} \delta_i t_i \geq 0$. Thus, we assume that for $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ the following holds:

$$0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{10}$$

We show that if any of the conditions (7)–(9) is violated, we obtain a contradiction by finding $(n^0, t_1^0, \ldots, t_k^0) \in \mathbf{P}_{RC}$ such that $0 \leq an^0 + \sum_{i=1}^{k} b_i t_i^0 + c \leq n^0$ does not hold.

Proof of (7). Let us first show that $0 \leq a \leq 1$.

Assume by contradiction that $a > 1$. From (10) we know that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^{k} b_i t_i + c$, that is, $(1-a)n \geq \sum_{i=1}^{k} b_i t_i + c$. Since $1 - a < 0$, we obtain

$$n \leq \frac{\sum_{i=1}^{k} b_i t_i + c}{1-a}, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{11}$$

Consider any tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $n^0 > \max\left\{\sum_{i=1}^{k} \delta_i t_i^0, \frac{\sum_{i=1}^{k} b_i t_i^0 + c}{1-a}\right\}$. By construction, we obtain: (i) the tuple is in $\mathbf{P}_{RC}$ because $n^0 > \sum_{i=1}^{k} \delta_i t_i^0$, and (ii) we have $n^0 > \frac{\sum_{i=1}^{k} b_i t_i^0 + c}{1-a}$, such that we arrive at the required contradiction to (11).

Assume now that $a < 0$. Again from (10) we have that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{i=1}^{k} b_i t_i + c \geq 0$, or in other words $an \geq -\sum_{i=1}^{k} b_i t_i - c$. As $a < 0$, this means that

$$n \leq \frac{-\sum_{i=1}^{k} b_i t_i - c}{a}, \text{ for every } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{12}$$

Consider a tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ with $n^0 > \max\left\{\sum_{i=1}^{k} \delta_i t_i^0, \frac{-\sum_{i=1}^{k} b_i t_i^0 - c}{a}\right\}$. By construction it holds that $n^0 > \sum_{i=1}^{k} \delta_i t_i^0$, and thus the tuple is in $\mathbf{P}_{RC}$. Also by construction it holds that $n^0 > \frac{-\sum_{i=1}^{k} b_i t_i^0 - c}{a}$ which is a contradiction with (12).

Proof of (8). Let us now prove that $-\delta_i - 1 < b_i < \delta_i + 1$, for an arbitrary $i \in \{1, \ldots, k\}$.

Assume by contradiction that $b_i \geq \delta_i + 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $an + \sum_{j=1}^{k} b_j t_j + c \leq n$, or in other words $(1-a)n \geq \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0, 1]$, then $(1-a)n \leq n$, for every $n \geq 0$. Since $b_i \geq \delta_i + 1$, and $t_i \geq 0$, it holds that $b_i t_i \geq (\delta_i + 1) t_i$. Thus, we have that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds that

$$n \geq (1-a)n \geq \sum_{j=1}^{k} b_j t_j + c \geq (\delta_i + 1) t_i + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$(n - \delta_i t_i) - \sum_{j \neq i} b_j t_j - c \geq t_i, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{13}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ such that $t_i^0 = \max\{1, \sum_{j \neq i} (\delta_j - b_j) - c + 2\}$, $t_j^0 = 1$ for $j \neq i$, and $n^0 = \sum_{j=1}^{k} \delta_j t_j^0 + 1 = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{j=1}^{k} \delta_j t_j^0$. Let us check the inequality from (13). By construction we have $(n^0 - \delta_i t_i^0) - \sum_{j \neq i} b_j t_j^0 - c = \sum_{j \neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 - \sum_{j \neq i} b_j - c$, that is, $\sum_{j \neq i} (\delta_j - b_j) - c + 1$, which is strictly smaller than t_i^0 by construction. Thus, we obtained a contradiction with (13).

Let us now assume $b_i \leq -\delta_i - 1$. Recall from (10) that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{j=1}^{k} b_j t_j + c$. Since $a \in [0, 1]$, for every $n \in \mathbb{N}$ holds $an \leq n$, and since $b_i \leq -\delta_i - 1$, we have $b_i t_i \leq -\delta_i t_i - t_i$, for every $t_i \geq 0$. Thus, for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ we have

$$0 \leq an + \sum_{j=1}^{k} b_j t_j + c \leq n + (-\delta_i t_i - t_i) + \sum_{j \neq i} b_j t_j + c.$$

In other words, we have that

$$t_i \leq (n - \delta_i t_i) + \sum_{j \neq i} b_j t_j + c, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \tag{14}$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0) \in \mathbb{N}^{k+1}$ where $t_i^0 = \max\{\sum_{j\neq i}(\delta_j + b_j) + c + 2, 1\}$, $t_j^0 = 1$, for every $j \neq i$, and $n^0 = \sum_{j=1}^k \delta_j t_j^0 + 1 = \sum_{j\neq i} \delta_j + \delta_i t_i^0 + 1$. This tuple is in $\mathbf{P}_{RC}$, since $n^0 > \sum_{i=1}^k \delta_i t_i^0$. Let us check the inequality from (14). By construction we have $(n^0 - \delta_i t_i^0) + \sum_{j\neq i} b_j t_j^0 + c = \sum_{j\neq i} \delta_j + \delta_i t_i^0 + 1 - \delta_i t_i^0 + \sum_{j\neq i} b_j + c$, that is, $\sum_{j\neq i}(\delta_j + b_j) + c + 1$, which is strictly smaller than t_i^0 by construction. This gives us a contradiction with (14).

Proof of (9). And finally, let us prove that $-2\sum_{i=1}^k \delta_i - k - 1 \leq c \leq 2\sum_{i=1}^k \delta_i + k + 1$.

Assume by contradiction that $c > 2\sum_{i=1}^k \delta_i + k + 1$. Recall that for every $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $n \geq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \geq 0$, $b_i > -\delta_i - 1$, for every $i = 1, \ldots, k$, and $c > 2\sum_{i=1}^k \delta_i + k + 1$, then we have that

$$n \geq an + \sum_{i=1}^k b_i t_i + c > \sum_{i=1}^k (-\delta_i - 1) t_i + 2\sum_{i=1}^k \delta_i + k + 1, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \quad (15)$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. The tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $\sum_{i=1}^k (-\delta_i - 1) t_i^0 + 2\sum_{i=1}^k \delta_i + k + 1 = \sum_{i=1}^k \delta_i + 1 = n^0$, which is a contradiction with (15).

Assume by contradiction that $c < -2\sum_{i=1}^k \delta_i - k - 1$. Recall that for all $(n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}$ holds $0 \leq an + \sum_{i=1}^k b_i t_i + c$, by (10). Since $a \leq 1$, $b_i < \delta_i + 1$, for every $i = 1, \ldots, k$, and $c < -2\sum_{i=1}^k \delta_i - k - 1$, then we have that

$$0 \leq an + \sum_{i=1}^k b_i t_i + c < n + \sum_{i=1}^k (\delta_i + 1) t_i - 2\sum_{i=1}^k \delta_i - k - 1, \text{ for all } (n, t_1, \ldots, t_k) \in \mathbf{P}_{RC}. \quad (16)$$

Consider the tuple $(n^0, t_1^0, \ldots, t_k^0)$ where $t_1^0 = \ldots = t_k^0 = 1$, and $n^0 = \sum_{i=1}^k \delta_i t_i^0 + 1 = \sum_{i=1}^k \delta_i + 1$. This tuple is in $\mathbf{P}_{RC}$ since $n^0 > \sum_{i=1}^k \delta_i t_i^0$, but by construction it holds that $n^0 + \sum_{i=1}^k (\delta_i + 1) t_i^0 - 2\sum_{i=1}^k \delta_i - k - 1 = 0$, which is a contradiction with (16). ◀

Proof of Theorem 3. As the given guard is sane for the resilience condition, the number compared against a shared variable should have a value from 0 to n. For every tuple $(n, t_1, \ldots, t_k)$ of parameter values satisfying the resilience condition, it should hold that $0 \leq an + \sum_{i=1}^k b_i t_i + c \leq n$. We may thus apply Lemma 5 and the theorem follows. ◀

Proof of Corollary 4. Using the fact that $x \leq \frac{\tilde{d}}{D} \leq y$ implies that $Dx \leq \tilde{d} \leq Dy$, for a $D \in \mathbb{N}$, this corollary follows directly from Theorem 3. ◀

B Thresholds with floor and ceiling functions

The following theorem considers threshold guards that use the ceiling or the floor function. It uses the same reasoning as in Theorem 3, combined with the properties of these functions. Namely, for every $x \in \mathbb{R}$ it holds that $x \leq \lceil x \rceil < x + 1$ and $x - 1 < \lfloor x \rfloor \leq x$.

▶ **Theorem 6.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^k \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \ldots, k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right) \quad \text{or} \quad x < f\left(an + (b_1 t_1 + \ldots + b_k t_k) + c\right),$$

where $x \in \Gamma$ is a shared variable, $a, b_1, \ldots, b_k, c \in \mathbb{Q}$ are rationals, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds that

$$0 \leq a \leq 1, \tag{17}$$

$$-\delta_i - 1 < b_i < \delta_i + 1, \text{ for all } i = 1, \ldots, k, \tag{18}$$

$$-2(\delta_1 + \ldots + \delta_k) - k - 2 \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k, \text{ if } f \text{ is floor, or} \tag{19}$$

$$-2(\delta_1 + \ldots + \delta_k) - k \leq c \leq 2(\delta_1 + \ldots + \delta_k) + k + 2, \text{ if } f \text{ is ceiling.} \tag{20}$$

Proof sketch. The proof largely follows the arguments of the proof of Lemma 5 with fixed denominators as in Corollary 4. The only remaining issue is that instead of constraints of the form $0 \leq an + \sum_{i=1}^{k} b_i t_i + c \leq n$, that are considered in Lemma 5, here we have to argue about constraints of the form $0 \leq f\left(an + \sum_{i=1}^{k} b_i t_i + c\right) \leq n$, where f is the ceiling or the floor function.

Let us first discuss the case when f is the ceiling function. As for every $x \in \mathbb{R}$ holds that $x \leq \lceil x \rceil < x + 1$, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil < an + (b_1 t_1 + \ldots + b_k t_k) + c + 1.$$

Still, as the guard is sane, we have that $0 \leq \lceil an + (b_1 t_1 + \ldots + b_k t_k) + c \rceil \leq n$. Combining these two constraints, we obtain that

$$0 < an + (b_1 t_1 + \ldots + b_k t_k) + (c + 1) \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + c \leq n.$$

With these constraints, we can derive a contradiction following the proof of Lemma 5.

Similarly, if f is the floor function, we use the fact that for every $x \in \mathbb{R}$ holds that $x - 1 < \lfloor x \rfloor \leq x$. Therefore, we have that

$$an + (b_1 t_1 + \ldots + b_k t_k) + c - 1 < \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq an + (b_1 t_1 + \ldots + b_k t_k) + c.$$

As $0 \leq \lfloor an + (b_1 t_1 + \ldots + b_k t_k) + c \rfloor \leq n$, we obtain that

$$0 \leq an + (b_1 t_1 + \ldots + b_k t_k) + c \quad \text{and} \quad an + (b_1 t_1 + \ldots + b_k t_k) + (c - 1) < n.$$

And again, the rest of the proof follows the line of the proof of Lemma 5. ◀

If coefficients in guards have a fixed denominator, we can obtain intervals for numerators as a direct consequence of Theorem 6.

▶ **Corollary 7.** *Fix a $k \in \mathbb{N}$. Let $n > \sum_{i=1}^{k} \delta_i t_i \wedge \forall i. t_i \geq 0$ be a resilience condition, where $\delta_i > 0$, $i = 1, \ldots, k$, and $n, t_1, \ldots, t_k \in \Pi$ are parameters. Fix a threshold guard of the form*

$$x \geq f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right) \quad \text{or} \quad x < f\left(\frac{\tilde{a}}{D} n + \sum_{i=1}^{k} \frac{\tilde{b_i}}{D} t_i + \frac{\tilde{c}}{D}\right),$$

where $x \in \Gamma$ is a shared variable, $\tilde{a}, \tilde{b_1}, \ldots, \tilde{b_k}, \tilde{c} \in \mathbb{Z}$ are integers, $D \in \mathbb{N}$, and f is either the ceiling or the floor function. If the guard is sane for the resilience condition, then it holds

$$0 \leq a \leq D, \tag{21}$$

$$D(-\delta_i - 1) < b_i < D(\delta_i + 1), \text{ for all } i = 1, \ldots, k, \tag{22}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k - 2) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k), \text{ if } f \text{ is floor, or} \tag{23}$$

$$D(-2(\delta_1 + \ldots + \delta_k) - k) \leq c \leq D(2(\delta_1 + \ldots + \delta_k) + k + 2), \text{ if } f \text{ is ceiling.} \tag{24}$$