Information and Computation 252 (2017) 95–109

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability ☆

Igor Konnov, Helmut Veith, Josef Widder *

TU Wien (Vienna University of Technology), Austria

ARTICLE INFO

Article history:
Received 15 December 2014
Available online 2 March 2016

Keywords:
Model checking
Fault-tolerant distributed algorithms
Byzantine faults
Computational models

ABSTRACT

Counter abstraction is a powerful tool for parameterized model checking, if the number of local states of the concurrent processes is relatively small. In recent work, we introduced parametric interval counter abstraction that allowed us to verify the safety and liveness of threshold-based fault-tolerant distributed algorithms (FTDA). Due to state space explosion, applying this technique to distributed algorithms with hundreds of local states is challenging for state-of-the-art model checkers. In this paper, we demonstrate that reachability properties of FTDAs can be verified by bounded model checking. To ensure completeness, we need an upper bound on the distance between states. We show that the diameters of accelerated counter systems of FTDAs, and of their counter abstractions, have a quadratic upper bound in the number of local transitions. Our experiments show that the resulting bounds are sufficiently small to use bounded model checking for parameterized verification of reachability properties of several FTDAs, some of which have not been automatically verified before.

1. Introduction

A system that consists of concurrent anonymous (identical) processes can be modeled as a counter system: Instead of recording which process is in which local state, we record for each local state, how many processes are in this state. We have one counter per local state ℓ, denoted by $\kappa[\ell]$. Each counter is bounded by the number of processes. A step by a process that goes from local state ℓ to local state ℓ' is modeled by decrementing $\kappa[\ell]$ and incrementing $\kappa[\ell']$.

We consider a specific class of counter systems, namely those that are defined by *threshold automata*. The technical motivation to introduce threshold automata is to capture the relevant properties of fault-tolerant distributed algorithms (FTDAs). FTDAs are an important class of distributed algorithms that work even if a subset of the processes fails [26]. Typically, they are parameterized in the number of processes and the number of tolerated faulty processes. These numbers of processes are parameters of the verification problem. We show that the counter systems defined by threshold automata have a diameter whose bound is independent of the bound on the counters, but depends only on characteristics of the threshold automaton. This bound can be used for parameterized model checking of FTDAs, as we confirm by experimental evaluation.

☆ Supported by the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405) and project P27722 (PRAVDA), and by the Vienna Science and Technology Fund (WWTF) through project ICT15-103 (APALACHE) and grant PROSEED.

* Corresponding author.
E-mail address: widder@forsyte.at (J. Widder).

http://dx.doi.org/10.1016/j.ic.2016.03.006

Modeling FTDAs as counter systems defined by threshold automata A threshold automaton consists of rules that define the conditions and effects of changes to the local state of a process of a distributed algorithm. Conditions are *threshold guards* that compare the value of a shared variable to a linear combination of parameters, e.g., $x \geq n - t$, where x is a shared variable and n and t are parameters. This captures counting arguments which are used in FTDAs, e.g., a process takes a certain step only, if it has received a message from a majority of processes. To model this, we use the shared variable x as the number of processes that have sent a message, n as the number of processes in the system, and t as the assumed number of faulty processes. The condition $x \geq n - t$ then captures a majority under the resilience condition that $n > 2t$. Resilience conditions are standard assumptions for the correctness of an FTDA.[1] The effect of a rule of a threshold automaton is that a shared variable is increased, which naturally captures that a process has sent a message. As a process cannot undo the sending of a message, it is natural to consider threshold automata where shared variables are never decreased. In addition, we use shared variables to model the number of processes that have sent a specific message. To be able to do so, we have to restrict how often a process may send a specific message. In particular, to model the counting mechanism, we have to prevent that a process sends a message from within an infinite loop (or a loop where the number of iterations is unknown). We are thus led to consider threshold automata where rules that form cycles do not modify shared variables. While we add this restriction to derive our technical contribution, we do not consider it too limiting with respect to the application domain: Indeed, in all our case studies a process sends a given message at most once; this property appears natural if one considers distributed algorithms under the classic assumption of reliable communication.

Bounding the diameter For reachability it is not relevant whether we "move" processes one by one from local state ℓ to local state ℓ'. If several processes perform the same transition one after the other, we can model this as a single update on the counters: The sequence where b processes one after the other move from ℓ to ℓ' can be encoded as a single transition where $\kappa[\ell]$ is decreased by b and $\kappa[\ell']$ is increased by b. We call the value of b the *acceleration factor*. It may vary in a run depending on how many repetitions of the same transition should be captured. We call such runs of a counter system *accelerated*. The lengths of accelerated runs are the ones relevant for the diameter of the counter system.

Our central idea is that given a run that starts in configuration σ and ends in configuration σ', by swapping and accelerating transitions in that run, we can construct a run of bounded length that also starts in σ and ends in σ'. This bound then gives us the diameter. For deriving this bound, the main technical challenge comes from the interactions of shared variables and threshold guards. We address it with the following three ideas:

i. *Acceleration*. As discussed above.
ii. *Sorting*. Given an arbitrary run of a counter system, we can shorten it by changing the order of transitions such that there are possibly many consecutive transitions that can be merged according to (i), and the resulting run leads to the same configuration as the original run. However, as we have arithmetic threshold conditions, not all changes of the order result in allowed runs.
iii. *Segmentation*. We partition a run into segments, inside of which we can reorder the transitions; cf. (ii).

In combination, these three ideas enable us to prove the main theorem: *The diameter of a counter system is at most quadratic in the number of rules; more precisely, it is bounded by the product of the number of rules and the number of distinct threshold conditions.* In particular, the diameter is independent of the parameter values.

Using the bound for parameterized model checking Parameterized model checking is concerned with the verification of concurrent or distributed systems, where the number of processes is not a priori fixed, that is, a system is verified for all sizes [6]. In our case, the counter systems for all values of n and t that satisfy the resilience condition should be verified. A well-known parameterized model checking technique is to map all these counter systems to a *counter abstraction*, where the counter values are not natural numbers, but range over an abstract finite domain [30]. In [14], we developed a more general form of counter abstraction for expressions used in threshold guards, which leads, e.g., to the abstract domain of four values that capture the parametric intervals $[0, 1)$ and $[1, t + 1)$ and $[t + 1, n - t)$ and $[n - t, \infty)$. It is easy to see [14] that a counter abstraction simulates all counter systems for all parameter values that satisfy the resilience condition. The bound d on the diameter of counter systems implies a bound $\hat{d}$ on the diameter of the counter abstraction. From this and simulation follows that if an abstract state is not reachable in the counter abstraction within $\hat{d}$ steps, then no concretization of this state is reachable in any of the concrete counter systems. This allows us to efficiently combine counter abstraction with *bounded model checking* [5]. Typically, bounded model checking is restricted to finding bugs that occur after a bounded number of steps of the systems. However, if one can show that within this bound every state is reachable from an initial state, bounded model checking is a complete method for verifying reachability.

[1] Indeed much research in distributed algorithms is devoted to show that certain problems are solvable only under some resilience condition, e.g., the seminal result on Byzantine fault tolerance by Pease et al. [28].

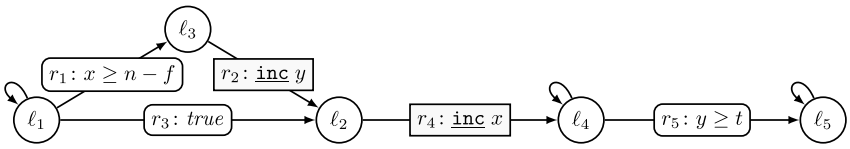

Fig. 1. Example of a threshold automaton.

2. Our approach at a glance

Fig. 1 represents a threshold automaton: The circles depict the local states, and the arrows represent rules (r_1 to r_5) that define how the automaton makes transitions. Rounded corner labels correspond to conditional rules, so that the rule can only be executed if the threshold guard evaluates to true. In our example, x and y are shared variables, and n, t, and f are parameters. We assume that they satisfy the resilience condition $n \geq 2t \wedge f \leq t$. The number of processes (that each execute the automaton) depends on the parameters, in this example we assume that n processes run concurrently. Finally, rectangular labels on arrows correspond to rules that increment a shared variable. The transitions of the counter system are then defined using the rules, e.g., when rule r_2 is executed, then variable y is incremented and the counters $\kappa[\ell_3]$ and $\kappa[\ell_2]$ are updated.

Consider a counter system in which the parameter values are $n = 3$, and $t = f = 1$. Let σ_0 be the configuration where $x = y = 0$ and all counters are 0 except $\kappa[\ell_1] = 3$. This configuration corresponds to a concurrent system where all three processes are in ℓ_1. For illustration, we assume that in this concurrent system processes have the identifiers 1, 2, and 3, and we denote by $r_i(j)$ that process j executes rule r_i. Recall that we have anonymous (symmetric) systems, so we use the identifiers only for illustration: the transition of the counter system is solely defined by the rule being executed.

As we are interested in the diameter, we have to consider the distance between configurations in terms of length of runs. In this example, we consider the distance of σ_0 to a configuration where $\kappa[\ell_5] = 3$, that is, all three processes are in local state ℓ_5. First, observe that the rule r_5 is locked in σ_0 as $y = 0$ and $t = 1$. Hence, we require that rule r_2 is executed at least once so that the value of y increases. However, due to the precedence relation on the rules, before that, r_1 must be executed, which is also locked in σ_0. The sequence of transitions $\tau_1 = r_3(1), r_4(1), r_3(2), r_4(2)$ leads from σ_0 to the configuration where $\kappa[\ell_1] = 1$, $\kappa[\ell_4] = 2$, and $x = 2$; we denote it by σ_1. In σ_1, rule r_1 is unlocked, so we may apply $\tau_2 = r_1(3), r_2(3)$, to arrive at σ_2, where $y = 1$, and thus r_5 is unlocked. To σ_2 we may apply $\tau_3 = r_5(1), r_5(2), r_4(3), r_5(3)$ to arrive at the required configuration σ_3 with $\kappa[\ell_5] = 3$.

In order to exploit acceleration as much as possible, we would like to group together occurrences of the same rule. In τ_1, we can actually swap $r_4(1)$ and $r_3(2)$ as locally the precedence relation of each process is maintained, and both rules are unconditional. Similarly, in τ_3, we can move $r_4(3)$ to the beginning of the sequence τ_3. Concatenating these altered sequences, the resulting schedule is $\tau = r_3(1), r_3(2), r_4(1), r_4(2), r_1(3), r_2(3), r_4(3), r_5(1), r_5(2), r_5(3)$. We can group together the consecutive occurrences for the same rules r_i, and write the schedule using pairs consisting of rules and acceleration factors, that is, $(r_3, 2)$, $(r_4, 2)$, $(r_1, 1)$, $(r_2, 1)$, $(r_4, 1)$, $(r_5, 3)$.

In schedule τ, the occurrences of all rules are grouped together except for r_4. That is, in the accelerated schedule we have two occurrences for r_4, while for the other rules one occurrence is sufficient. Actually, there is no way around this: We cannot swap $r_2(3)$ with $r_4(3)$, as we have to maintain the local precedence relation of process 3. More precisely, in the counter system, r_4 would require us to decrease the counter $\kappa[\ell_2]$ at a point in the schedule where $\kappa[\ell_2] = 0$. We first have to increase the counter value by executing a transition according to rule r_2, before we can apply r_4. Moreover, we cannot move the subsequence $r_1(3), r_2(3), r_4(3)$ to the left, as $r_1(3)$ is locked in the prefix.

In this paper we characterize such cases. The issue here is that r_4 can unlock r_1 (we use the notation $r_4 \prec_U r_1$), while r_1 precedes r_4 in the control flow of the processes ($r_1 \prec_P^+ r_4$). We coin the term *milestone* for transitions like $r_1(3)$ that cannot be moved, and show that the same issue arises if a rule r locks a threshold guard of rule r', where r precedes r' in the control flow. As processes do not decrease shared variables, we have at most one milestone per threshold guard. The sequence of transitions between milestones is called a segment. We prove that transitions inside a segment can be swapped, so that one can group transitions for the same rule in so-called batches. Each of these batches can then be replaced by a single accelerated transition that leads to the same configuration as the original batch. Hence, any segment can be replaced by an accelerated one whose length is at most the number of rules of a process. This, and the number of milestones, gives us the required bound on the diameter. This bound is independent of the parameters, and only depends on the number of threshold guards and the precedence relation between the rules of the processes.

Our main result is that the diameter is independent of the parameter values. In contrast, reachability of a specific local state depends on the parameter values: In order for a process to reach ℓ_5 in our example, at least $n - f$ processes must execute r_4 before at least t other processes must execute r_2. That is, the system must contain at least $(n - f) + t$ processes. In case of $t > f$, we obtain $(n - f) + t > n$, which is a contradiction, and ℓ_5 cannot be reached for such parameter values. The *model checking problem* we are interested in is whether *a given state is unreachable for all parameter values* that satisfy the resilience condition.

3. Parameterized counter systems

3.1. Threshold automata

A threshold automaton describes a process in a concurrent system. It is defined by its local states, shared variables, parameters, and by rules that define the state changes and their conditions and effects on shared variables. Formally, a *threshold automaton* is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ defined below.

States The set $\mathcal{L}$ is the finite set of *local states*, and $\mathcal{I} \subseteq \mathcal{L}$ is the set of *initial local states*. (As we later will index counters by local states, for simplicity we use the convention that $\mathcal{L} = \{1, \dots, |\mathcal{L}|\}$.) The set Γ is the finite set of *shared variables* that range over $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. To simplify the presentation, we view the variables as vectors in $\mathbb{N}_0^{|\Gamma|}$. The finite set Π is a set of *parameter variables* that range over $\mathbb{N}_0$, and the *resilience condition* RC is a predicate over $\mathbb{N}_0^{|\Pi|}$. Then, we denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : RC(\mathbf{p})\}$.

Rules A rule defines a conditional transition between local states that may update the shared variables. The semantics is defined via counter systems in Section 3.2. Here we only give informal explanations of the semantics.

Formally, a *rule* is a tuple $(\mathit{from}, \mathit{to}, \varphi^{\leq}, \varphi^{>}, \mathbf{u})$: The local states *from* and *to* are from $\mathcal{L}$. Intuitively, they capture from which local state to which a process moves, or, in terms of counter systems, which counters decrease and increase, respectively. A rule is only executed if the conditions $\varphi^{\leq}$ and $\varphi^{>}$ evaluate to true. Each condition consists of multiple guards. Each guard is defined using some shared variable $x \in \Gamma$, and coefficients $a_0, \dots, a_{|\Pi|} \in \mathbb{Z}$, so that

$$a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i \ \leq\ x \qquad \text{and} \qquad a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i \ >\ x$$

are a *lower guard* and *upper guard*, respectively (both, variables and coefficients, may differ for different guards). The *condition* $\varphi^{\leq}$ is a conjunction of lower guards, and the condition $\varphi^{>}$ is a conjunction of upper guards. Rules may increase shared variables. We model this using an update vector $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$, which is added to the vector of shared variables, when the rule is executed. Then $\mathcal{R}$ is the finite set of rules.

Definition 1 *(Precedence).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *precedence relation* $\prec_P$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows:

$$r_1 \prec_P r_2 \text{ iff } r_1.\mathit{to} = r_2.\mathit{from}.$$

We denote by $\prec_P^+$ the transitive closure of $\prec_P$. If $r_1 \prec_P^+ r_2 \wedge r_2 \prec_P^+ r_1$, or if $r_1 = r_2$, we write $r_1 \sim_P r_2$.

The precedence relation thus captures the control flow of a process. For instance, in the example of Fig. 1 it captures that a process must execute rule r_4 before it can execute rule r_5.

Definition 2 *(Unlock relation).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *unlock relation* $\prec_U$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows: $r_1 \prec_U r_2$ iff there is a $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ and a $\mathbf{p} \in \mathbf{P}_{RC}$ satisfying

- $(\mathbf{g}, \mathbf{p}) \models r_1.\varphi^{\leq} \wedge r_1.\varphi^{>}$,
- $(\mathbf{g}, \mathbf{p}) \not\models r_2.\varphi^{\leq} \wedge r_2.\varphi^{>}$, and
- $(\mathbf{g} + r_1.\mathbf{u}, \mathbf{p}) \models r_2.\varphi^{\leq} \wedge r_2.\varphi^{>}$.

In the example of Fig. 1, rule r_4 increases the shared variable x, and by that may unlock rule r_1. We thus write $r_4 \prec_U r_1$. Similarly, r_2 unlocks r_5 in the example.

Definition 3 *(Lock relation).* For a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the *lock relation* $\prec_L$ as subset of $\mathcal{R} \times \mathcal{R}$ as follows: $r_1 \prec_L r_2$ iff there is a $\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$ and a $\mathbf{p} \in \mathbf{P}_{RC}$ satisfying

- $(\mathbf{g}, \mathbf{p}) \models r_1.\varphi^{\leq} \wedge r_1.\varphi^{>}$,
- $(\mathbf{g}, \mathbf{p}) \models r_2.\varphi^{\leq} \wedge r_2.\varphi^{>}$, and
- $(\mathbf{g} + r_1.\mathbf{u}, \mathbf{p}) \not\models r_2.\varphi^{\leq} \wedge r_2.\varphi^{>}$.

Our analysis in Section 4 will show that only two types of conditions of the threshold automaton contribute to the diameter we are interested in. First, these are the conditions that appear in a rule r that can be unlocked by a rule r' that comes after rule r or is parallel to r in the control flow. More precisely, rule r' does not appear before r in the control flow. The other conditions we are interested in are those that appear in a rule r that can be locked by a rule r'' that is before r or parallel to r in the control flow; more precisely, r'' does not appear after r in the control flow. This leads to the definition of the following quantities.

Definition 4 *(Number of relevant conditions).* Given a threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$, we define the following quantities:

$$\mathcal{C}^{\leq} = |\{r.\varphi^{\leq} : r \in \mathcal{R}, \exists r' \in \mathcal{R}.\, r' \not\prec_P^+ r \ \wedge\ r' \prec_U r\}|$$
$$\mathcal{C}^{>} = |\{r.\varphi^{>} : r \in \mathcal{R}, \exists r'' \in \mathcal{R}.\, r \not\prec_P^+ r'' \ \wedge\ r'' \prec_L r\}|$$
$$\mathcal{C} = \mathcal{C}^{\leq} + \mathcal{C}^{>}.$$

To determine these quantities, we have to check whether a specific condition can potentially lock (or unlock) another one, as defined in Definition 2 (or Definition 3). Observe that this can be done efficiently using an SMT solver.

We consider specific threshold automata, namely those that naturally capture FTDAs, where rules that form cycles do not increase shared variables.

Definition 5 *(Canonical threshold automaton).* A threshold automaton $(\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ is canonical, if $r.\mathbf{u} = \mathbf{0}$ for all rules $r \in \mathcal{R}$ that satisfy $r \prec_P^+ r$.

The relation $\sim_P$ defines equivalence classes of rules. For a given set of rules $\mathcal{R}$ let $\mathcal{R}/\sim$ be the set of equivalence classes defined by $\sim_P$. We denote by $[r]$ the equivalence class of rule r. For two classes c_1 and c_2 from $\mathcal{R}/\sim$ we write $c_1 \prec_C c_2$ iff there are two rules r_1 and r_2 in $\mathcal{R}$ satisfying $[r_1] = c_1$ and $[r_2] = c_2$ and $r_1 \prec_P^+ r_2$ and $r_1 \not\sim_P r_2$. Observe that the relation $\prec_C$ is a strict partial order (irreflexive and transitive). Hence, there are linear extensions of $\prec_C$. Below, we fix an *arbitrary* of these linear extensions. We will later use it to sort transitions in a schedule.

Notation. We denote by $\prec_C^{lin}$ a linear extension of $\prec_C$.

Proposition 6. *If* $[r_2] \prec_C^{lin} [r_1]$ *then* $r_1 \not\prec_P^+ r_2$.

Proof. Suppose by contradiction that $[r_2] \prec_C^{lin} [r_1]$ and $r_1 \prec_P^+ r_2$. We derive a contradiction by distinguishing two cases:

- if $r_1 \not\sim_P r_2$, then by the definition of $\prec_C$, we get $[r_1] \prec_C [r_2]$;
- otherwise, that is, if $r_1 \sim_P r_2$, it follows that $[r_1] = [r_2]$.

In both cases we derive a contradiction to $[r_2] \prec_C^{lin} [r_1]$. □

The semantics of threshold automata are defined with respect to counter systems in the following section.

3.2. Counter systems

Given a threshold automaton $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ and admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, we define in the following a counter system as a transition system (Σ, I, R) that consists of the set of configurations Σ, the set of initial configurations I, and the transition relation R.

Configurations A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ consists of a vector of *counter values* $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$, a vector of *shared variable values* $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$, and a vector of *parameter values* $\sigma.\mathbf{p} = \mathbf{p}$. The set Σ is the set of all configurations. The function $N : \mathbf{P}_{RC} \to \mathbb{N}_0$ formalizes the number of processes to be modeled. In our example, the number of processes is given by the value of the parameter n. In [14], we discussed a case study where $N(n, t, f) = n - f$. The set of initial configurations I contains the configurations that satisfy

- $\sigma.\mathbf{g} = \mathbf{0}$,
- $\sum_{i \in \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and
- $\sum_{i \notin \mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$.

Transition relation A *transition* is a pair $t = (rule, factor)$ of a rule of the threshold automaton and a non-negative integer called the *acceleration factor*, or just factor for short. To simplify notation, for a transition $t = (rule, factor)$ we refer by $t.\mathbf{u}$ and $t.\varphi^{>}$ etc. to $rule.\mathbf{u}$ and $rule.\varphi^{>}$ etc., respectively. We say a transition t is *unlocked* in configuration σ if $\forall k \in \{0, \ldots, t.factor - 1\}$. $(\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\leq} \wedge t.\varphi^{>}$. For transitions t_1 and t_2 we say that the two transitions are related iff $t_1.rule$ and $t_2.rule$ are related, e.g., $t_1 \prec_P t_2$ iff $t_1.rule \prec_P t_2.rule$.

A transition t is *applicable to (or enabled in)* configuration σ, if it is unlocked, and if $\sigma.\boldsymbol{\kappa}[t.from] \geq t.factor$. We say that σ' is the result of applying the (enabled) transition t to σ, and use the notation $\sigma' = t(\sigma)$, if

- t is enabled in σ
- $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.factor \cdot t.\mathbf{u}$
- $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
- if $t.from \neq t.to$ then
  - $\sigma'.\boldsymbol{\kappa}[t.from] = \sigma.\boldsymbol{\kappa}[t.from] - t.factor$,
  - $\sigma'.\boldsymbol{\kappa}[t.to] = \sigma.\boldsymbol{\kappa}[t.to] + t.factor$, and
  - for all ℓ in $\mathcal{L} \setminus \{t.from, t.to\}$ it holds that $\sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$
- if $t.from = t.to$ then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$

The transition relation $R \subseteq \Sigma \times \Sigma$ of the counter system is defined as follows: $(\sigma, \sigma') \in R$ iff there is a $r \in \mathcal{R}$ and a $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$. As updates to shared variables do not decrease their values, we obtain:

Proposition 7. *For all configurations σ, all rules r, and all transitions t applicable to σ, the following holds:*

1. *If $\sigma \models r.\varphi^{\leq}$ then $t(\sigma) \models r.\varphi^{\leq}$*
2. *If $t(\sigma) \not\models r.\varphi^{\leq}$ then $\sigma \not\models r.\varphi^{\leq}$*
3. *If $\sigma \not\models r.\varphi^{>}$ then $t(\sigma) \not\models r.\varphi^{>}$*
4. *If $t(\sigma) \models r.\varphi^{>}$ then $\sigma \models r.\varphi^{>}$*

The proposition formalizes a crucial property of our systems that will allow us to bound the diameter below: For instance, by repeated application of points 1 and 2 we obtain that once a condition of form $\varphi^{\leq}$ evaluates to true it will always do so in the future, while if $\varphi^{\leq}$ evaluates to false, it always has in the past. We conclude that for each condition, the evaluation changes at most once.

Schedules A *schedule* is a sequence of transitions. A schedule $\tau = t_1, \ldots, t_m$ is called *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \ldots, \sigma_m$ such that $\sigma_i = t_i(\sigma_{i-1})$ for $0 < i \leq m$. A schedule $t_1, \ldots, t_m$ where $t_i.factor = 1$ for $0 < i \leq m$ is a *conventional schedule*. If there is a $t_i.factor > 1$, then a schedule is called *accelerated*.

We write $\tau \cdot \tau'$ to denote the concatenation of two schedules τ and τ', and treat a transition t as schedule. If $\tau = \tau_1 \cdot t \cdot \tau_2 \cdot t' \cdot \tau_3$, for some τ_1, τ_2, and τ_3, we say that transition t precedes transition t' in τ, and denote this by $t \rightarrow_\tau t'$.

4. Diameter of counter systems

In this section, we will present the outline of the proof of our main theorem:

Theorem 8. *Given a canonical threshold automaton* TA*, for each* $\mathbf{p}$ *in* $\mathbf{P}_{RC}$ *the diameter of the counter system is less than or equal to* $d(\mathsf{TA}) = (\mathcal{C} + 1) \cdot |\mathcal{R}| + \mathcal{C}$*, and thus independent of* $\mathbf{p}$.

From the theorem it follows that for all parameter values, reachability in the counter system can be verified by exploring runs of length at most $d(\mathsf{TA})$. However, the theorem alone is not sufficient to solve the parameterized model checking problem. For this, we combine the bound with the abstraction method in [14]. More precisely, the counter abstraction in [14] simulates the counter systems for *all* parameter values that satisfy the resilience condition. Consequently, the bound on the length of the run of the counter systems entails a bound for the counter abstraction. As we explain in Section 4.5, we exploit this in the experiments in Section 5.

4.1. Proof idea

Given a rule r, a schedule τ and two transitions t_i and t_j, with $t_i \rightarrow_\tau t_j$, the subschedule $t_i \cdot \ldots \cdot t_j$ of $\tau = t_1, \ldots, t_m$ is a *batch of rule* r if $t_\ell.rule = r$ for $i \leq \ell \leq j$, and if the subschedule is maximal, that is, $i = 1 \vee t_{i-1} \neq r$ and $j = m \vee t_{j+1} \neq r$. Similarly, we define a batch of a class c as a subschedule $t_i \cdot \ldots \cdot t_j$ where $[r_\ell] = c$ for $i \leq \ell \leq j$, and where the subschedule is maximal as before.

Definition 9 *(Sorted schedule).* Given a schedule τ, and the relation $\prec_c^{lin}$, we define $sort(\tau)$ as the schedule that satisfies:

1. $sort(\tau)$ is a permutation of schedule τ;
2. two transitions from the same equivalence class maintain their relative order, that is, if $t \rightarrow_\tau t'$ and $t \sim_P t'$, then $t \rightarrow_{sort(\tau)} t'$;
3. if $t \rightarrow_{sort(\tau)} t'$, then $t \sim_P t'$ or $[t] \prec_c^{lin} [t']$.

Proposition 10. *Given a schedule τ, and the relation $\prec_c^{lin}$, for each equivalence class defined by $\sim_P$ there is at most one batch in $sort(\tau)$.*

Proof. Assume by contradiction that there are two batches, that is, there is an equivalence class c such that there are two transitions t_1 and t_2 in c, and there is a transition $t' \notin c$ such that $t_1 \rightarrow_{sort(\tau)} t'$ and $t' \rightarrow_{sort(\tau)} t_2$. By Definition 9(3) it follows from $t_1 \rightarrow_{sort(\tau)} t'$ that $c \prec_c^{lin} [t']$ and from $t' \rightarrow_{sort(\tau)} t_2$ that $[t'] \prec_c^{lin} c$. As $\prec_c^{lin}$ is a total order we arrive at the required contradiction. □

Note that from Proposition 10 and Definition 9 (Points 1 and 2) it follows that $sort(\tau)$ is indeed unique for a given τ.

The crucial observation to prove Theorem 8 is that if we have a schedule $\tau_1 = t \cdot t'$ applicable to configuration σ with $t.rule = t'.rule$, we can replace it with another applicable (one-transition) schedule $\tau_2 = t''$, with $t''.rule = t.rule$ and $t''.factor = t.factor + t'.factor$, such that $\tau_1(\sigma) = \tau_2(\sigma)$. Thus, we can reach the same configuration with a shorter schedule. More generally, we may replace a batch of a rule by a single accelerated transition whose factor is the sum of all factors in the batch.

To bound the diameter, we have to bound the distance between any two configurations σ and σ' for which there is a schedule τ applicable to σ satisfying $\sigma' = \tau(\sigma)$. A simple case is if $sort(\tau)$ is applicable to σ and each equivalence class defined by the precedence relation consists of a single rule (e.g., the threshold automaton is a directed acyclic graph). Then by Proposition 10 we have at most $|\mathcal{R}|$ batches in $sort(\tau)$, that is, one per rule. By the reasoning of above we can replace each batch by a single accelerated transition.

However, in general $sort(\tau)$ may not be applicable to σ, or there are equivalence classes containing multiple rules, i.e., rules form cycles in the precedence relation. The first issue comes from locking and unlocking, and as discussed in Section 2, we identify milestone transitions, and show that we can apply *sort* to the segments between milestones in Section 4.3. We also deal with the issue of cycles in the precedence relation. It is ensured by *sort* that within a segment, all transitions that belong to a cycle form a batch. In Section 4.2, we replace such a batch by a batch where the remaining rules do not form a cycle. Removing cycles requires the assumption that shared variables are not incremented in cycles.

4.2. Dealing with cycles

We consider the distance between two configurations σ and σ' that satisfy $\sigma.\mathbf{g} = \sigma'.\mathbf{g}$, i.e., along any schedule connecting these configurations, the values of shared variables are unchanged, and so are thus the evaluations of guards. By Definition 5, we can apply this section's result to batches of a class of canonical threshold automata. The following definition captures how often processes go to and leave specific states, respectively, and the updates on the variables.

Definition 11. Given a schedule $\tau = t_1, t_2, \dots t_k$, we denote by $|\tau|$ the length k of the schedule. Further, we define the following vectors

$$\mathbf{in}(\tau)[\ell] = \sum_{\substack{1 \le i \le |\tau| \\ t_i.to = \ell}} t_i.factor,$$

$$\mathbf{out}(\tau)[\ell] = \sum_{\substack{1 \le i \le |\tau| \\ t_i.from = \ell}} t_i.factor,$$

$$\mathbf{up}(\tau) = \sum_{1 \le i \le |\tau|} t_i.\mathbf{u}.$$

From the definition of a counter system, we directly obtain:

Proposition 12. *For all configurations σ, and all schedules τ applicable to σ, if $\sigma' = \tau(\sigma)$, then $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa} + \mathbf{in}(\tau) - \mathbf{out}(\tau)$, and $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + \mathbf{up}(\tau)$.*

The proposition directly implies the following:

Proposition 13. *For all configurations σ, and all schedules τ and τ' applicable to σ, if $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathbf{up}(\tau) = \mathbf{up}(\tau')$, then $\tau(\sigma) = \tau'(\sigma)$.*

Given a schedule $\tau = t_1, t_2, \dots$ we say that the index sequence $i(1), \dots, i(j)$ is a cycle in τ, if for all b, $1 \le b < j$, it holds that $t_{i(b)}.to = t_{i(b+1)}.from$, and $t_{i(j)}.to = t_{i(1)}.from$, and $t_{i(c)} \neq t_{i(d)}$ for $1 \le c < d \le j$. Let $\mathcal{R}(\tau)$ be the set of rules appearing in τ, that is, $\{r\colon t_i \in \tau \wedge t_i.rule = r\}$.

In the following proposition we are concerned with removing cycles, without considering applicability of the resulting schedule. We consider applicability later in Theorem 16.

Proposition 14. *For all schedules τ, if τ contains a cycle, then there is a schedule τ' satisfying $|\tau'| < |\tau|$, $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

Proof. Let $I = i(1), \ldots, i(j)$ be a cycle in $\tau = t_1, t_2, \ldots$, let $\tau_I = t_{i(1)}, \ldots, t_{i(j)}$, and let θ be the schedule $t'_1, \ldots, t'_j$ satisfying for $1 \leq k \leq j$ that

- $t'_k.rule = t_{i(k)}.rule$, and
- $t'_k.factor = t_{i(k)}.factor - \min\{t_b.factor \colon b \in I\}$.

As I is a cycle, by definition, for each local state s, the number of transitions that go out of s is equal to the number of transitions that enter s, that is, $|\{b \colon b \in I, t_b.from = s\}| = |\{b \colon b \in I, t_b.to = s\}|$. As θ is constructed from τ_I by reducing the factor of each transition by $\min\{t_b.factor \colon b \in I\}$, it follows that:

$$\mathbf{in}(\theta) - \mathbf{out}(\theta) = \mathbf{in}(\tau_I) - \mathbf{out}(\tau_I) \tag{1}$$

Denote with $\tau[\theta/I]$ the schedule obtained from τ by substituting all transitions in the positions $i(1), i(2), \ldots, i(j)$ with the transitions $t'_1, t'_2, \ldots, t'_j$ of the schedule θ, respectively. From (1), we immediately conclude that $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau[\theta/I]) - \mathbf{out}(\tau[\theta/I])$. Further, by construction, the schedule θ contains at least one transition t'_k with $t'_k.factor = 0$ for $1 \leq k \leq j$. Let τ' be the schedule obtained from the schedule $\tau[\theta/I]$ by removing all the transitions in the positions $i(1), i(2), \ldots, i(j)$ that have *factor* equal to zero. As removal of such transitions does not change **in** and **out**, we immediately conclude that $\mathbf{in}(\tau') - \mathbf{out}(\tau') = \mathbf{in}(\tau[\theta/I]) - \mathbf{out}(\tau[\theta/I])$ and thus $\mathbf{in}(\tau') - \mathbf{out}(\tau') = \mathbf{in}(\tau) - \mathbf{out}(\tau)$. Moreover, as we have removed at least one transition, it holds that $|\tau'| < |\tau|$, and as we have not added new rules, it holds that $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$. □

Repeated application of the proposition leads to a cycle-free schedule (possibly the empty schedule), and we obtain:

Theorem 15. *For all schedules τ, there is a schedule τ' that contains no cycles, $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

The issue with this theorem is that τ' is not necessarily applicable to the same configurations as τ. In the following theorem, we prove that if a schedule satisfies a specific condition on the order of transitions, then it is applicable.

Theorem 16. *Let σ and σ' be two configurations with $\sigma.\mathbf{g} = \sigma'.\mathbf{g}$, and τ be a schedule with $\mathbf{up}(\tau) = \mathbf{0}$, every transition t is unlocked in σ and satisfies $t.from \neq t.to$, and where if $t_i \rightarrow_\tau t_j$, then $t_j \not\prec_P t_i$. If $\sigma'.\boldsymbol{\kappa} - \sigma.\boldsymbol{\kappa} = \mathbf{in}(\tau) - \mathbf{out}(\tau)$, then τ is applicable to σ.*

Proof. The proof is by induction on $|\tau|$.
Base: $|\tau| = 0$. Follows trivially.
Step: $|\tau| > 0$. Let $\tau = t \cdot \tau'$ for some τ'. We first prove that t is applicable to σ, and then that $t(\sigma)$ and τ' satisfy the induction hypothesis. Then, the theorem follows.

By assumption, τ' does not contain a transition t' satisfying $t' \prec_P t$, that is, for all t_i in τ', $t_i.to \neq t.from$, and by assumption, $t.from \neq t.to$, and therefore

$$\mathbf{in}(\tau)[t.from] = 0 \tag{2}$$

Recall that from the definition of a configuration,

$$\sigma'.\boldsymbol{\kappa}[t.from] \geq 0. \tag{3}$$

By assumption

$$\sigma'.\boldsymbol{\kappa}[t.from] - \sigma.\boldsymbol{\kappa}[t.from] = \mathbf{in}(\tau)[t.from] - \mathbf{out}(\tau)[t.from].$$

Applying (3) we obtain $\sigma.\boldsymbol{\kappa}[t.from] \geq \mathbf{out}(\tau)[t.from] - \mathbf{in}(\tau)[t.from]$, and further from (2) we get $\sigma.\boldsymbol{\kappa}[t.from] \geq \mathbf{out}(\tau)[t.from]$. As t is in τ, from Definition 11 follows that $\mathbf{out}(\tau)[t.from] \geq t.factor$, and finally $\sigma.\boldsymbol{\kappa}[t.from] \geq t.factor$. It follows that t is applicable to σ.

It remains to prove that

$$\sigma'.\boldsymbol{\kappa} - t(\sigma).\boldsymbol{\kappa} = \mathbf{in}(\tau') - \mathbf{out}(\tau'), \tag{4}$$

which allows us to invoke the induction hypothesis. To do so, we consider the components of $\sigma.\kappa$. Observe that for all local states s, $s \notin \{t.from, t.to\}$, we have $t(\sigma).\boldsymbol{\kappa}[s] = \sigma.\boldsymbol{\kappa}[s]$, $\mathbf{in}(\tau')[s] = \mathbf{in}(\tau)[s]$, and $\mathbf{out}(\tau')[s] = \mathbf{out}(\tau)[s]$.

Hence, $\sigma'.\boldsymbol{\kappa}[s] - t(\sigma).\boldsymbol{\kappa}[s] = \mathbf{in}(\tau')[s] - \mathbf{out}(\tau')[s]$. To prove (4), it remains to consider the indices $t.from$ and $t.to$. Recall that by assumption, $t.from \neq t.to$.

Component *t.from* of (4). The counter for $t.from$ changes, that is, $\sigma.\boldsymbol{\kappa}[t.from] = t(\sigma).\boldsymbol{\kappa}[t.from] + t.factor$. As τ' is obtained by removing t from τ, we have $\mathbf{out}(\tau)[t.from] = \mathbf{out}(\tau')[t.from] + t.factor$, and $\mathbf{in}(\tau)[t.from] = \mathbf{in}(\tau')[t.from]$.

From the assumption $\sigma'.\boldsymbol{\kappa} - \sigma.\boldsymbol{\kappa} = \mathbf{in}(\tau) - \mathbf{out}(\tau)$ it follows that

$$\sigma'.\boldsymbol{\kappa}[t.from] - t(\sigma).\boldsymbol{\kappa}[t.from] - t.factor = \mathbf{in}(\tau')[t.from] - \mathbf{out}(\tau')[t.from] - t.factor,$$

and the case follows.

Component *t.to* of (4). The counter for $t.to$ changes, that is, $\sigma.\boldsymbol{\kappa}[t.to] = t(\sigma).\boldsymbol{\kappa}[t.to] - t.factor$. As τ' is obtained by removing t from τ, we have $\mathbf{in}(\tau)[t.to] = \mathbf{in}(\tau')[t.from] + t.factor$ and $\mathbf{out}(\tau)[t.to] = \mathbf{out}(\tau')[t.to]$.

Again, it follows from the assumption that

$$\sigma'.\boldsymbol{\kappa}[t.to] - t(\sigma).\boldsymbol{\kappa}[t.to] + t.factor = \mathbf{in}(\tau')[t.to] + t.factor - \mathbf{out}(\tau')[t.to].$$

Hence $\sigma'.\boldsymbol{\kappa} - t(\sigma).\boldsymbol{\kappa} = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and the theorem follows. □

Given a configuration σ, and a schedule τ applicable to σ, with $\mathbf{up}(\tau) = \mathbf{0}$, by Theorem 15 there is a cycle-free schedule τ' with $\mathbf{in}(\tau) - \mathbf{out}(\tau) = \mathbf{in}(\tau') - \mathbf{out}(\tau')$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$. As τ' contains no cycle, we may re-order the transitions in τ' according to $\prec_P$, as required by Theorem 16, such that there is at most one block per rule. By Theorem 16, the resulting schedule is applicable, and we obtain:

Corollary 17. *For all configurations σ, and all schedules τ applicable to σ, with $\mathbf{up}(\tau) = \mathbf{0}$, there is a schedule with at most one batch per rule applicable to σ satisfying that τ' contains no cycles, $\tau'(\sigma) = \tau(\sigma)$, and $\mathcal{R}(\tau') \subseteq \mathcal{R}(\tau)$.*

4.3. Defining milestones and swapping transitions

In this section we deal with locking and unlocking. To this end, we define milestones, and show that transitions that are not milestones can be swapped.

Proposition 18. *For all configurations σ, and all transitions t_1 and t_2, if t_2 is applicable to $t_1(\sigma)$ and t_1 is applicable to $t_2(\sigma)$, then $t_2(t_1(\sigma)) = t_1(t_2(\sigma))$.*

Proof. Follows from commutativity of addition applied to counters and shared variables. □

As discussed in Section 4.1, we would like to replace a schedule (or subschedule) τ by $sort(\tau)$, so that the resulting schedule $sort(\tau)$ is applicable. To do so, we have to show that if we start with τ and swap adjacent transitions until we reach $sort(\tau)$, all the intermediate schedules and the final schedule are applicable. However, due to locking and unlocking, we cannot always swap transitions. For instance, if t' appears directly before t in a schedule, and t' unlocks t (that is, t is locked in the configuration in which t' is applied and unlocked after the application of t'), swapping t' and t leads to a schedule which is not applicable. This is because t is not applicable. We observe that this problem occurs

- because we want to swap t' and t (t is before t' in the linear extension of the precedence relation, that is, t' is not before t in the precedence relation),
- t' unlocks t, and
- t is locked in the beginning.

In such cases, t must not be moved "to the left" in the schedule, and we call t a *left milestone*. We capture this intuition in the following definition.

Definition 19 *(Left milestone).* Given a configuration σ and a schedule $\tau = \tau' \cdot t \cdot \tau''$ applicable to σ, the transition t is a left milestone for σ and τ, if

1. there is a transition t' in τ' satisfying $t' \not\prec_P^+ t \ \wedge \ t' \prec_U t$,
2. $t.\varphi^{\leq}$ is locked in σ, and
3. for all t' in τ' it holds that $t'.\varphi^{\leq} \neq t.\varphi^{\leq}$.

The following definition of right milestones is analogous but, instead of unlocking and considering transitions that are locked in the beginning, considers the locking relation and transitions that are locked after application of the schedule.

Definition 20 *(Right milestone).* Given a configuration σ and a schedule $\tau = \tau' \cdot t \cdot \tau''$ applicable to σ, the transition t is a right milestone for σ and τ, if

1. there is a transition t'' in τ'' satisfying $t \not\prec_P^+ t'' \wedge t'' \prec_L t$,
2. $t.\varphi^>$ is locked in $\tau(\sigma)$, and
3. for all t'' in τ'' it holds that $t''.\varphi^> \neq t.\varphi^>$.

Milestones divide schedules into segments that are defined as follows.

Definition 21 *(Segment).* Given a schedule τ and configuration σ, τ' is a segment if it is a subschedule of τ, and does not contain a milestone for σ and τ.

The following theorem shows that two transitions that are not milestones can be swapped. Together with the fact that by definition the number of milestones is bounded by $\mathcal{C}$, repeated application of the theorem eventually leads to a schedule where milestones and sorted schedules alternate.

Theorem 22. *Let σ be a configuration, τ a schedule applicable to σ, and $\tau = \tau_1 \cdot t_1 \cdot t_2 \cdot \tau_2$. If transitions t_1 and t_2 are not milestones for σ and τ, and satisfy $[t_2] \prec_C^{lin} [t_1]$, then*

i. schedule $\tau' = \tau_1 \cdot t_2 \cdot t_1 \cdot \tau_2$ is applicable to σ, and
ii. $\tau'(\sigma) = \tau(\sigma)$.

Proof. We prove (i) by showing that (a) t_2 is applicable to $\sigma' = \tau_1(\sigma)$, and (b) t_1 is applicable to $t_2(\sigma')$. From (a), (b), and Proposition 18, Point (i) then follows.
(a) We prove that t_2 is applicable to σ' by case distinction

- $t_1 \not\prec_U t_2$. As the rule of t_1 never unlocks the rule of t_2, and because t_2 is unlocked in $t_1(\sigma')$, t_2 must also be unlocked in σ' due to Proposition 7.
- Otherwise, that is, $t_1 \prec_U t_2$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. It follows that t_2 satisfies Definition 19(1).
  Now assume by way of contradiction that t_2 is locked in σ'. We will show that from this assumption it follows that t_2 is a left milestone[2] for σ and τ to derive a contradiction and conclude that t_2 is unlocked in σ': As t_2 is locked in σ', from repeated application of Proposition 7(2) we obtain that $t_2.\varphi^\leq$ is locked in σ, and all intermediate configurations until σ'. As it is locked in σ, transition t_2 satisfies Definition 19(2). As the transition is locked in all intermediate configurations no transition that is guarded with the same condition can appear in the prefix, that is, for each transition t' in τ_1 it holds that $t'.\varphi^\leq \neq t_2.\varphi^\leq$, which satisfies Definition 19(3). Hence, t_2 is a left milestone, which contradicts that t_2 is not a milestone. We conclude that t_2 is unlocked in σ'.

As t_2 is unlocked in σ', by the definition of applicability, it is sufficient to prove that $\sigma'.\boldsymbol{\kappa}[t_2.from] \geq t_2.factor$. By assumption, t_2 is applicable to $t_1(\sigma')$ so that by the definition of applicability

$$t_1(\sigma').\boldsymbol{\kappa}[t_2.from] \geq t_2.factor \tag{5}$$

We have to distinguish two cases:

- If $t_1.from = t_2.from$, then $t_1(\sigma').\boldsymbol{\kappa}[t_2.from] = \sigma'.\boldsymbol{\kappa}[t_2.from] - t_1.factor$. From (5) it follows that $\sigma'.\boldsymbol{\kappa}[t_2.from] \geq t_1.factor + t_2.factor$ and this case follows.
- Otherwise, that is, if $t_1.from \neq t_2.from$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. From $t_1 \not\prec_P^+ t_2$ follows that $t_1 \not\prec_P t_2$ and thus $t_1.to \neq t_2.from$. Hence, $t_1(\sigma').\boldsymbol{\kappa}[t_2.from] = \sigma'.\boldsymbol{\kappa}[t_2.from]$. Consequently, this case follows at once from (5).

(b) As t_1 is applicable to σ', it is unlocked in σ'. We again distinguish two cases:

- $t_2 \not\prec_L t_1$. As the rule of t_2 never locks the rule of t_1, and because t_1 is unlocked in σ', the transition t_1 must also be unlocked in $t_2(\sigma')$.
- Otherwise, that is, $t_2 \prec_L t_1$. Due to Proposition 6, from $[t_2] \prec_C^{lin} [t_1]$ follows that $t_1 \not\prec_P^+ t_2$. Hence, t_1 satisfies Definition 20(1). Now assume by way of contradiction that t_1 is locked in $t_2(\sigma')$. We will show that t_1 is a right milestone to arrive at the required contradiction: As t_1 is unlocked in σ and locked in $t_2(\sigma')$, it is locked due to $t_1.\varphi^>$, which evaluates to false in $t_2(\sigma')$. As t_1 is locked in $t_2(\sigma')$, and as the values of global variables in $t_2(t_1(\sigma'))$ are greater than or equal to those of $t_2(\sigma')$, it follows that $t_1.\varphi^>$ evaluates to false in $t_2(t_1(\sigma'))$. From this and repeated application

[2] Intuitively, as τ' is obtained from τ by moving t_2 one position to the left, we argue about t_2 being a left milestone here. In point (b) below, we view τ' being obtained from τ by moving t_1 one position to the right, and consequently derive a contradiction using the notion of right milestone for t_1.

of Proposition 7(3) we obtain that $t_1.\varphi^{>}$ is locked in $\tau(\sigma)$, which satisfies Definition 20(2). Further as t_1 is locked in $t_2(\sigma')$, for each transition t'' in τ_2 it holds that $t''.\varphi^{>} \neq t_2.\varphi^{>}$, which satisfies Definition 20(3). Hence, t_1 is a right milestone, which provides the required contradiction.

We conclude that t_1 is unlocked in $t_2(\sigma')$.

It remains to show that $t_2(\sigma').\boldsymbol{\kappa}[t_1.\mathit{from}] \geq t_1.\mathit{factor}$, which can be proven analogously to the argument on counters in (a).

By assumption, τ_2 is applicable to $t_2(t_1(\tau_1(\sigma)))$, and from Proposition 18 follows that $t_2(t_1(\tau_1(\sigma))) = t_1(t_2(\tau_1(\sigma)))$. Consequently, τ_2 is applicable to $t_1(t_2(\tau_1(\sigma)))$. Hence, τ' is applicable to σ, and Point (ii) of the theorem statement follows from Proposition 13. □

4.4. Proof of main theorem

Theorem 8. *Given a canonical threshold automaton* TA*, for each* $\mathbf{p}$ *in* $\mathbf{P}_{RC}$ *the diameter of the counter system is less than or equal to* $d(\mathsf{TA}) = (\mathcal{C}+1)\cdot|\mathcal{R}| + \mathcal{C}$*, and thus independent of* $\mathbf{p}$.

Proof. We can view a schedule τ applicable to σ as alternation of segments τ_i and milestones m_i. We obtain from repeated application of Theorem 22, that each schedule applicable to σ can be transformed into a schedule $\mathit{sort}(\tau_1)\cdot m_1 \cdot \mathit{sort}(\tau_2)\cdot m_2 \cdot \ldots$ that is also applicable to σ. By Proposition 10 there is at most one batch per equivalence class in $\mathit{sort}(\tau_i)$. If this equivalence class consists of a single rule, the batch can be replaced by a single (accelerated) transition. Otherwise, that is, if a class consists of say x rules, as we consider canonical threshold automata that do not have updates to shared variables in rules r with $r \prec_P^+ r$, we can use the construction of Section 4.2 to replace the batch of this class by at most x accelerated transitions. We arrive at a segment that contains at most one transition per rule, that is, at most $|\mathcal{R}|$ transitions. It remains to bound the number of milestones.

As by Definition 19(3) and Definition 20(3) there is at most one milestone per condition, we have at most $\mathcal{C}$ milestones as defined in Definition 4. To conclude, we obtain an accelerated schedule, consisting of $\mathcal{C}$ milestones and $\mathcal{C}+1$ segments of length at most $|\mathcal{R}|$. □

4.5. Applying our result

In the proof of Theorem 8, we bound the length of all segments by $|\mathcal{R}|$. However, by Definition 19, segments to the left of a left milestone cannot contain transitions for rules with the same condition as the milestone. The same is true for segments to the right of right milestones. As we will see in Section 5.4, our tool ByMC explores all orders of milestones, an uses this observation about milestones to compute a more precise bound $\boldsymbol{d}^{\star}$ for the diameter.

Our encoding of the counter abstraction only increments and decrements counters. If $|\hat{D}|$ is the size of the abstract domain, a transition in a counter system is simulated by at most $|\hat{D}|-1$ steps in the counter abstraction; this leads to the diameter $\hat{\boldsymbol{d}}$ for counter abstractions, which we use in our experiments.

5. Experimental evaluation

We have implemented the techniques discussed in this article in our tool ByMC [16]. The input are the descriptions of our benchmarks given in our parametric extension of Promela [15], which describe parameterized processes. Hence, as preliminary step, ByMC computes the PIA data abstraction [14] in order to obtain finite state processes. Based on this, ByMC does preprocessing to compute threshold automata and the locking and unlocking relations, and to generate the inputs for our model checking back-ends.

5.1. Preprocessing

To apply our results, we have to compute the set of rules $\mathcal{R}$. Recall that a rule is a tuple $(\mathit{from}, \mathit{to}, \varphi^{\leq}, \varphi^{>}, \mathbf{u})$. ByMC computes the reachable local states. In the case of the CBC case study, e.g., this step reduces the local states under consideration from 2000 to 100, approximately. All our experiments — including the ones with FASTer [2] — are based on the reduced local state space.

Then, for each pair $(\mathit{from}, \mathit{to})$, ByMC explores symbolic paths to compute the guards and update vectors for the pair, and removes the infeasible paths using an SMT solver. From this we get the set of rules $\mathcal{R}$. Then, ByMC encodes Definition 2 in the SMT solver Yices, to construct the lock $\prec_L$ and unlock $\prec_U$ relations. ByMC computes the relations $\{(r,r') \colon r' \not\prec_P^+ r \wedge r' \prec_U r\}$ and $\{(r,r'') \colon r \not\prec_P^+ r'' \wedge r'' \prec_L r\}$ as required by Definition 4. This provides the bounds we use for bounded model checking.

Table 1
Benchmark overview giving the article from which we formalized the distributed algorithm, the number of shared variables, and the model and the size of the abstract domain.

Benchmark	Reference	Shared vars	Abs. domain size
FRB	[9]	1	2
STRB	[32]	1	4
ABA0	[8]	2	4
ABA1	[8]	2	5
CBC0	[27]	4	4
CBC1	[27]	4	5
NBAC(C)	[31]	4	4

5.2. Back-ends

ByMC generates the PIA counter abstraction [14] to be used by the following back-end model checkers. We have also implemented an automatic abstraction refinement loop for the counterexamples provided by NuSMV.

BMC. NuSMV 2.5.4 [10] (using MiniSAT) performs incremental bounded model checking with the bound $\hat{d}$. If a counterexample is reported, ByMC refines the system as explained in [14], if the counterexample is spurious.

BMCL. This technique combines the power of NuSMV 2.5.4 and of the state-of-the-art multi-core SAT solver Plingeling ats1 [4]. NuSMV performs incremental bounded model checking for 30 steps. If a spurious counterexample is found, then ByMC refines the system description. When NuSMV does not report a counterexample, NuSMV generates a single CNF with the bound $\hat{d}$. Satisfiability of this formula is then checked with Plingeling.

BDD. NuSMV 2.5.4 performs BDD-based symbolic checking.

FAST. FASTer 2.1 [2] performs reachability analysis using the plugin Mona-1.3.

5.3. Benchmarks

We encoded several asynchronous FTDAs in our parametric Promela, following the technique in [15]; they can be obtained from our git repository.[3] All models contain transitions with lower threshold guards. The benchmarks CBC also contain upper threshold guards. If we ignore self-loops, the precedence relation of all but NBAC and NBACC, which have non-trivial cycles, are partial orders. We provide the most relevant data on these benchmarks in Table 1, and discuss them in more detail below.

Folklore reliable broadcast (FRB) In this FTDA, n processes have to agree on whether a process has broadcast a message, in the presence of $f \leq n$ crashes. Our model of FRB has one shared variable and the abstract domain of two intervals $[0, 1)$ and $[1, \infty)$. In this paper, we are concerned with the safety property *unforgeability*: If no process is initialized with value 1 (message from the broadcaster), then no correct process ever accepts.

Consistent broadcast (STRB) Here, we have $n - f$ correct processes and $f \geq 0$ Byzantine faulty ones. The resilience condition is $n > 3t \wedge t \geq f$. There is one shared variable and the abstract domain of four intervals $[0, 1)$, $[1, t + 1)$, $[t + 1, n - t)$, and $[n - t, \infty)$. In the experiments reported here, we check only unforgeability (see FRB), whereas in [14] we checked also liveness properties.

Byzantine agreement (ABA) There are $n > 3t$ processes, $f \leq t$ of them Byzantine faulty. The model has two shared variables. We have to consider two different cases for the abstract domain, namely, case ABA0 with the domain $[0, 1)$, $[1, t + 1)$, $[t + 1, \lceil \frac{n+t}{2} \rceil)$, and $[\lceil \frac{n+t}{2} \rceil, \infty)$ and case ABA1 with the domain $[0, 1)$, $[1, t + 1)$, $[t + 1, 2t + 1)$, $[2t + 1, \lceil \frac{n+t}{2} \rceil)$, and $[\lceil \frac{n+t}{2} \rceil, \infty)$. As for FRB, we check unforgeability. This case study, and all below, run out of memory when using Spin for model checking the counter abstraction [14].

Condition-based consensus (CBC) This is a restricted variant of consensus solvable in asynchronous systems. We consider the binary version of condition-based consensus in the presence of clean crashes, which requires four shared variables. Under the resilience condition $n > 2t \wedge f \geq 0$, we have to consider two different cases depending on f: If $f = 0$ we have case CBC0 with the domain $[0, 1)$, $[1, \lceil \frac{n}{2} \rceil)$, $[\lceil \frac{n}{2} \rceil, n - t)$, and $[n - t, \infty)$. If $f \neq 0$, case CBC1 has the domain: $[0, 1)$, $[1, f)$, $[f, \lceil \frac{n}{2} \rceil)$, $[\lceil \frac{n}{2} \rceil, n - t)$, and $[n - t, \infty)$. We verified several properties, all of which resulted in experiments with similar characteristics. We only give $validity_0$ in the table, i.e., no process accepts value 0, if all processes initially have value 1.

[3] https://github.com/konnov/fault-tolerant-benchmarks/tree/master/concur14.

Table 2
Summary of experiments on AMD Opteron®Processor 6272 with 192 GB RAM and 32 CPU cores. Plingeling used up to 16 cores. "TO" denotes timeout of 24 hours; "OOM" denotes memory overrun of 64 GB; "ERR" denotes runtime error; "RTO" denotes that the refinement loop timed out. When BMC and BMCL times out, we indicate the maximum length of the explored computations in percentage of the predicted diameter bound.

Input FTDA	Threshold A.				Bounds			Time, [HH:]MM:SS				Memory, GB			
	$\lvert\mathcal{L}\rvert$	$\lvert\mathcal{R}\rvert$	$\mathcal{C}^{\leq}$	$\mathcal{C}^{>}$	d	$d^{\star}$	$\hat{d}$	BMCL	BMC	BDD	FAST	BMCL	BMC	BDD	FAST
Fig. 1	5	5	1	0	11	9	27	00:00:03	00:00:04	00:01	00:00:08	0.01	0.02	0.02	0.06
FRB	6	8	1	0	17	10	10	00:00:13	00:00:13	00:06	00:00:08	0.01	0.02	0.02	0.01
STRB	7	15	3	0	63	30	90	00:00:09	00:00:06	00:04	00:00:07	0.02	0.03	0.02	0.07
ABA0	37	180	6	0	1266	586	1758	00:21:26	02:20:10	00:15	00:08:40	6.37	1.49	0.07	3.56
ABA1	61	392	8	0	3536	1655	6620	TO 25%	TO 12%	00:33	02:36:25	TO	TO	0.08	15.65
CBC0	43	204	0	0	204	204	612	01:38:54	TO 57%	OOM	ERR	1.28	TO	OOM	ERR
CBC1	115	896	1	1	2690	2180	8720	TO 05%	TO 11%	TO	TO	TO	TO	TO	TO
NBACC	109	1724	6	0	12 074	5500	16 500	RTO	RTO	TO	TO	RTO	RTO	TO	TO
NBAC	77	1356	6	0	9498	4340	13 020	RTO	RTO	TO	TO	RTO	RTO	TO	TO
When a Bug is Introduced															
ABA0	32	139	6	0	979	469	1407	00:00:16	00:00:18	TO	00:05:57	0.04	0.04	TO	2.70
ABA1	54	299	8	0	2699	1305	5220	00:00:22	00:00:21	TO	ERR	0.06	0.06	TO	ERR

Non-blocking atomic commitment (NBAC and NBACC) Here, n processes are initialized with Yes or No and decide on whether to commit a transaction. The transaction must be aborted if at least one process is initialized to No. We consider the cases NBACC and NBAC of clean crashes and crashes, respectively. Both models contain four shared variables, and the abstract domain is $[0, 1)$ and $[1, n)$ and $[n-1, n)$, and $[n, \infty)$. The algorithm uses a failure detector, which is modeled as local variable that changes its value non-deterministically.

5.4. Evaluation

Table 2 summarizes the experiments. For the threshold automata, we give the number of local states $|\mathcal{L}|$, the number of rules $|\mathcal{R}|$, and conditions according to Definition 4, i.e., $\mathcal{C}^{\leq}$ and $\mathcal{C}^{>}$. The column $\boldsymbol{d}$ provides the bound on the diameter as in Theorem 8, whereas the column $\boldsymbol{d^{\star}}$ provides an improved diameter, and $\hat{d}$ the diameter of the counter abstraction, both discussed in Section 4.5.

As the experiments show, all techniques rapidly verify FRB, STRB, and Fig. 1. We had already verified FRB and STRB before using Spin [14]. The more challenging examples are ABA0 and ABA1, where BDD clearly outperforms the other techniques. Bounded model checking is slower here, because the diameter bound does not exploit knowledge on the specification. FAST performs well on these benchmarks. We believe this is because many rules are always disabled, due to the initial states as given in the specification. To confirm this intuition, we introduced a bug into ABA0 and ABA1, which allows the processes to non-deterministically change their value to 1. This led to a dramatic slowdown of BDD and FAST, as reflected in the last two lines.

Using the bounds of this paper, BMCL verified CBC0, whereas all other techniques failed. BMCL did not reach the bounds for CBC1 with our experimental setup. In this case, we report the percentages of the bounds we reached with bounded model checking.

In the experiments with NBAC and NBACC, the refinement loop timed out. We are convinced that we can address this issue by integrating the refinement loop with an incremental bounded model checker.

While we could not check all the benchmarks with the technique of this paper, a more aggressive offline partial order reduction in combination with SMT-based bounded model checking [17] allowed us to verify also these benchmarks.

6. Related work and discussions

Specific forms of counter systems can be used to model parameterized systems of concurrent processes. Lubachevsky [25] discusses *compact* programs that reach each state in a bounded number of steps, where the bound is independent of the number of processes. Besides, in [25] he gives examples of compact programs, and in [24] he proves that specific semaphore programs are compact. We not only show compactness, but give a bound on the diameter. In our case, communication is not restricted to semaphores, but we have threshold guards. Counter abstraction [30] follows this line of research, but as discussed by Basler et al. [3], does not scale well for large numbers of local states.

Another line of research is on *acceleration* in infinite state systems, e.g., in flat counter automata [22]. Acceleration is a technique that computes the transitive closure of a transition relation and applies it to the set of states. The tool FAST [1] uses the transitive closure of transitions to compute the set of reachable states in a symbolic procedure. This appears closely related to our transitions with acceleration factor. However, in [1] a transition is chosen and accelerated dynamically in the course of symbolic state space exploration, while we use acceleration factors and reordering to construct a bound as a formula over the characteristics of a threshold automaton (precedence, lock, and unlock relations). Our tool generates the cardinalities of these relations to compute length of computations for bounded model checking.

One can achieve completeness in bounded model checking by exploring all runs that are not longer than the diameter of the system [5]. This was later generalized to the notion of *completeness threshold* by Clarke et al. [11] in the presence of safety and liveness properties. To find a completeness threshold for a liveness property, it is sufficient to compute the diameter of the synchronous product of the transition system and a Büchi automaton, which represents the computations violating the property. As in general, computing the diameter is believed to be as hard as model checking, one can use a coarser bound provided by the reoccurrence diameter [19]. In practice, the reoccurrence diameter of counter abstraction is prohibitively large, so that we are interested bounds on the diameter.

Partial orders are a useful concept when reasoning about distributed systems [20]. In the context of model checking, *partial order reduction* [13,33,29] is a widely used technique to reduce the search space. It is based on the idea that changing the order of steps of concurrent processes leads to "equivalent" behavior with respect to the specification. Typically, partial order reduction is used on-the-fly to prune runs that are equivalent to representative ones. In contrast, in this paper, we bound the length of representative runs offline in order to ensure completeness of bounded model checking. Based on the ideas presented here, in [17] we introduce a more aggressive form of partial order reduction that, together with an encoding of a counter system in SMT, allowed us to verify reachability of even more involved fault-tolerant distributed algorithms. In the context of FTDAs, a partial order reduction was introduced by Bokor et al. [7]. Similar to this paper, they focus on "quorum transitions" that count messages. The technique by Bokor et al. [7] can be used for model checking small instances, while we focus on parameterized model checking.

Our technique of determining which transitions can be swapped in a run reminds of *movers* as discussed by Lipton [23], or more generally the idea to show that certain actions can be grouped into larger atomic blocks to simplify proofs [12,21]. However, movers address the issue of grouping many local transitions of a process together. In contrast, we conceptually group transitions of different processes together into one accelerated transition. Moreover, the definition of a mover by Lipton is independent of a specific run: a left mover (e.g., a "release" operation) is a transition that in *all runs* can "move to the left" with respect to transitions of other processes. In our work, we look at specific runs and identify which transitions (milestones) must not move in this run.

Our technique targets at threshold-based fault-tolerant distributed algorithms, that is, asynchronous distributed algorithms that communicate by sending messages to all and compare the number of received messages to linear combinations of parameters. As motivated by this application domain (and as discussed in the introduction), the systems we consider are symmetric, and the threshold automata we consider are restricted in that shared variables cannot be decreased, and rules that form a cycle in a threshold automaton may not increase shared variables. To model concurrent systems other than fault-tolerant distributed algorithms, it may be interesting to weaken the latter two restrictions. Our results on the diameter do not necessarily carry over to less restricted threshold automata and counter systems.

As next steps we will focus on liveness of fault-tolerant distributed algorithms. In fact the liveness specifications are in the fragment of linear temporal logic for which it is proven [18] that a formula can be translated into a cliquey Büchi automaton. For such automata, Kroening et al. provide a completeness threshold. Still, there are open questions related to applying our results to the idea by Kroening et al. [18].

References

[1] S. Bardin, A. Finkel, J. Leroux, L. Petrucci, FAST: acceleration from theory to practice, Int. J. Softw. Tools Technol. Transf. 10 (5) (2008) 401–424.
[2] S. Bardin, J. Leroux, G. Point, Fast extended release, in: CAV, in: LNCS, vol. 4144, 2006, pp. 63–66.
[3] G. Basler, M. Mazzucchi, T. Wahl, D. Kroening, Symbolic counter abstraction for concurrent software, in: CAV, in: LNCS, vol. 5643, 2009, pp. 64–78.
[4] A. Biere, Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013, Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions, 2013, pp. 51–52.
[5] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: TACAS, in: LNCS, vol. 1579, 1999, pp. 193–207.
[6] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, J. Widder, Decidability of parameterized verification, in: Synthesis Lectures on Distributed Computing Theory, 2015.
[7] P. Bokor, J. Kinder, M. Serafini, N. Suri, Efficient model checking of fault-tolerant distributed protocols, in: DSN, 2011, pp. 73–84.
[8] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840.
[9] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (March 1996) 225–267.
[10] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an opensource tool for symbolic model checking, in: CAV, in: LNCS, vol. 2404, 2002, pp. 359–364.
[11] E.M. Clarke, D. Kroening, J. Ouaknine, O. Strichman, Completeness and complexity of bounded model checking, in: VMCAI, in: LNCS, vol. 2937, 2004, pp. 85–96.
[12] T.W. Doeppner, Parallel program correctness through refinement, in: POPL, 1977, pp. 155–169.
[13] P. Godefroid, Using partial orders to improve automatic verification methods, in: CAV, in: LNCS, vol. 531, 1990, pp. 176–185.
[14] A. John, I. Konnov, U. Schmid, H. Veith, J. Widder, Parameterized model checking of fault-tolerant distributed algorithms by abstraction, in: FMCAD, 2013, pp. 201–209.
[15] A. John, I. Konnov, U. Schmid, H. Veith, J. Widder, Towards modeling and model checking fault-tolerant distributed algorithms, in: SPIN, in: LNCS, vol. 7976, 2013, pp. 209–226.
[16] I. Konnov, ByMC: Byzantine model checker, http://forsyte.tuwien.ac.at/software/bymc/, 2015, accessed: Dec. 1.
[17] I. Konnov, H. Veith, J. Widder, SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms, in: CAV (Part I), in: LNCS, vol. 9206, 2015, pp. 85–102.
[18] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, J. Worrell, Linear completeness thresholds for bounded model checking, in: CAV, in: LNCS, vol. 6806, 2011, pp. 557–572.
[19] D. Kroening, O. Strichman, Efficient computation of recurrence diameters, in: VMCAI, in: LNCS, vol. 2575, 2003, pp. 298–309.
[20] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[21] L. Lamport, F.B. Schneider, Pretending atomicity, Tech. rep. 44, SRC, 1989.
[22] J. Leroux, G. Sutre, Flat counter automata almost everywhere!, in: ATVA, in: LNCS, vol. 3707, 2005, pp. 489–503.
[23] R.J. Lipton, Reduction: a method of proving properties of parallel programs, Commun. ACM 18 (12) (1975) 717–721.
[24] B.D. Lubachevsky, An approach to automating the verification of compact parallel coordination programs, II, Tech. rep. 64, New York University, Computer Science Department, 1983.
[25] B.D. Lubachevsky, An approach to automating the verification of compact parallel coordination programs. I, Acta Inform. 21 (2) (1984) 125–169.
[26] N. Lynch, Distributed Algorithms, Morgan, Kaufman, 1996.
[27] A. Mostéfaoui, E. Mourgaya, P.R. Parvédy, M. Raynal, Evaluating the condition-based approach to solve consensus, in: DSN, 2003, pp. 541–550.
[28] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234.
[29] D. Peled, All from one, one for all: on model checking using representatives, in: CAV, in: LNCS, vol. 697, 1993, pp. 409–423.
[30] A. Pnueli, J. Xu, L. Zuck, Liveness with $(0, 1, \infty)$-counter abstraction, in: CAV, in: LNCS, vol. 2404, Springer, 2002, pp. 93–111.
[31] M. Raynal, A case study of agreement problems in distributed systems: non-blocking atomic commitment, in: HASE, 1997, pp. 209–214.
[32] T. Srikanth, S. Toueg, Simulating authenticated broadcasts to derive simple fault-tolerant algorithms, Distrib. Comput. 2 (1987) 80–94.
[33] A. Valmari, Stubborn sets for reduced state space generation, in: Advances in Petri Nets 1990, in: LNCS, vol. 483, Springer, 1991, pp. 491–515.