Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction*

Benjamin Aminof[1], Sasha Rubin[2], Ilina Stoilkovska[1](✉), Josef Widder[1], and Florian Zuleger[1]

[1] TU Wien, Vienna, Austria
{benj, stoilkov, widder, zuleger}@forsyte.at
[2] Università degli Studi di Napoli Federico II, Naples, Italy
sasha.rubin@unina.it

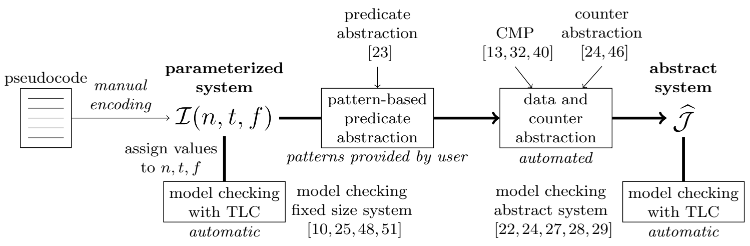
Abstract. Parameterized verification of fault-tolerant distributed algorithms has recently gained more and more attention. Most of the existing work considers asynchronous distributed systems (interleaving semantics). However, there exists a considerable distributed computing literature on synchronous fault-tolerant distributed algorithms: conceptually, all processes proceed in lock-step rounds, that is, synchronized steps of all (correct) processes bring the system into the next state.
We introduce an abstraction technique for synchronous fault-tolerant distributed algorithms that reduces parameterized verification of synchronous fault-tolerant distributed algorithms to finite-state model checking of an abstract system. Using the TLC model checker, we automatically verified several algorithms from the literature, some of which were not automatically verified before. Our benchmarks include fault-tolerant algorithms that solve atomic commitment, 2-set agreement, and consensus.

1 Introduction

Fault-tolerant distributed algorithms (FTDAs) are hard to design and prove correct. It is easy to introduce bugs when developing and "optimizing" such distributed algorithms [41]. As we currently see more and more implementations of FTDAs [8, 31, 42, 55], it is desirable to be able to quickly check, whether an optimization did not break the desired behavior. Hence, we observe increasing interest in tool support for eliminating design bugs in distributed algorithms by means of automated verification [1, 2, 17, 18, 26, 28, 35, 45, 52, 54].

The vast majority of the existing literature on verification of distributed systems considers asynchronous systems, that is, the methods are designed for interleaving semantics. Disentangling the methods from the interleaving semantics is challenging. At the same time, there is substantial literature on distributed

* This work is partially supported by the Austrian Science Fund (FWF) via NFN RiSE (S11403, S11405), project PRAVDA (P27722), and the doctoral college LogiCS W1255; and by the Vienna Science and Technology Fund (WWTF) through grant ICT12-059. S. Rubin is a Marie Curie fellow of the Istituto Nazionale di Alta Matematica.



predicate abstraction [23]
CMP [13, 32, 40]
counter abstraction [24, 46]
pseudocode
manual encoding
parameterized system
$\mathcal{I}(n,t,f)$
pattern-based predicate abstraction
data and counter abstraction
abstract system
$\widehat{\mathcal{J}}$
assign values to n,t,f
patterns provided by user
automated
model checking with TLC
automatic
model checking fixed size system [10, 25, 48, 51]
model checking abstract system [22, 24, 27, 28, 29]
model checking with TLC
automatic

Fig. 1: Overview of our approach and related work

algorithms that are *not* designed for interleaving semantics, namely round-based distributed algorithms [9,11,12,34,37,44,47,53]. In these algorithms, computations proceed in rounds, in which processes perform send, receive and compute transitions in lock-step. There are mainly three reasons for the interest in synchronous distributed algorithms: First, the assumption on synchrony circumvents impossibility results regarding fault-tolerance in asynchronous systems [21]. Second, for hard real-time systems, the underlying hardware and network infrastructure exhibits predictable timing behavior, so that designers of embedded and cyber-physical systems (e.g., in cars and planes) are willing to exploit these timing guarantees at the algorithmic level [30]. Finally, the abstraction of a round that is performed by all processes in lock-step makes it supposedly easier to design algorithms; although there are counterexamples where incorrect synchronous distributed algorithms have been published, as reported in [36].

We focus on verification of synchronous FTDAs, and will adapt and combine several verification methods that were originally designed for asynchronous systems, and apply them in the synchronous setting. Fig. 1 gives an overview of our work together with references of related approaches for the asynchronous case. Our main contribution lies in parameterized verification of FTDAs, that is, we want to verify a distributed algorithm that is executed by n processes in an environment where f processes fail, and designed to work if at most t processes fail, for all values n, t, and f that satisfy some arithmetic conditions, e.g., $f \leq t < n$. This algorithm is formalized as a parameterized system $\mathcal{I}(n,t,f)$, to which we apply abstraction to obtain a finite abstract system $\widehat{\mathcal{J}}$ which serves as input to TLC [50], the model checker associated with TLA+ [33].

To understand the trade-off between parameterized model checking and model checking of fixed size systems, we also did verification of the latter, that is, we fix n, t, and f to small values, e.g., 5, 3, and 2. The resulting fixed size system again serves as input to TLC, as shown in the figure. Our experiments show that model checking fixed sized systems quickly runs into combinatorial state space explosion. This confirms that to verify systems of bigger size, one needs to rely on abstractions, which give verification results for systems of all sizes.

There are several existing approaches for verifying round-based distributed algorithms. Fixed size systems, i.e., (small) instances, were verified using model checking, e.g., in [10, 48, 51]. The following two approaches to parameterized verification are most related as they also target the round-based model from [12]: [17] proposes invariant checking using decision procedures, requiring the user to provide invariants manually. [38] gives a cut-off theorem for reducing the parameterized problem to verification of small systems (5 to 7 processes). This cut-off theorem considers only consensus algorithms [12], while we are also interested in other algorithms, e.g., k-set agreement or atomic commitment. We discuss the relation of [38] to our work in more detail in Section 6.

Contributions. We introduce a new technique for parameterized model checking of synchronous distributed algorithms.

- We introduce a special *guarded command language* for distributed algorithms, and show that this language allows effective verification by abstraction.
- We combine *automated abstractions* [15,24,32,40,46] that had been introduced for asynchronous systems, and adapt them to synchronous systems.
- Our *modeling framework* uses an independent environment to express the semantics of the code in the presence of faults. While we focus on crash faults in this paper, in the future this will allow us to express semantics of other faults models (e.g., omission, Byzantine) in a modular way.
- We introduce pattern-based *predicate abstraction* for termination guards. This allows verification engineers to specify verification conditions for specific guards and environments. For termination guards found in many synchronous algorithms, we provide verification conditions, which can be reused.
- We do experiments on several synchronous FTDAs [9, 11, 37, 47], some of which were not automatically verified before. Our experiments show that parameterized model checking performs better than checking fixed-size systems already for few (typically 5) processes.

2 Overview on our Approach

A synchronous distributed algorithm runs on a fully connected network of $n \in \mathbb{N}$ processes, which communicate with each other by exchanging messages. The computations are organized in rounds; each round consists of two phases: (1) the *message exchange phase* in which each process broadcasts a message to all others, and (2) the *state transition phase* in which processes update their variables based on the messages received. The processes work synchronously in the sense that they simultaneously switch to the next phase of every round.

We focus on *fault-tolerant agreement* algorithms [5, 37, 47], where processes irrevocably decide a value depending on the initial values of all processes. There are multiple agreement problems in the literature that differ in the way the decision values are related to the initial values. In *consensus*, the processes reach agreement on a value that has been initially proposed by at least one process. In *2-set agreement*, processes may decide on one out of two different values from the set of initial values. In *non-blocking atomic commit*, the processes decide the value *abort* if there is at least one process that initially proposed *abort*, and decide the value *commit* if all processes initially proposed *commit*.

***FloodSet* algorithm:**

```
The message alphabet consists of subsets of $W = \{0, 1\}$.
$v_0 \in W$ is a default decision value
$r \in \mathbb{N}$ is the round number, initially 0
states_i:
  $w \subseteq W$, initially containing $i$'s initial value
  $d \in \{0, 1, un\}$, initially $un$
msgs_i
1. if $r \leq t$ then
2.     send $w$ to all other processes
3. $r := r + 1$
trans_i
4. let $X_j$ be the message from $j$, for each $j$
   from which a message arrives
5. $w := w \cup \bigcup_j X_j$
6. if $r = t + 1$ then
7.     if $|w| = 1$ then $d := v$, where $w = \{v\}$
8.     else $d := v_0$
```

(a) The pseudocode of *FloodSet*

Validity: If all processes start with the same initial value, then this is the only possible decision value.
Agreement: No two correct processes decide on different values.
Termination: All correct processes eventually decide.

(b) Specifications

$$\begin{array}{ccccccc} s.\mathbf{w} & s.\mathbf{d} & s.\mathbf{Msg} & s.r & s.\mathbf{cr} & s.\mathbf{fld} & s.\mathbf{rcv} \\ \begin{bmatrix}\{0\}\\\{0\}\\\{1\}\\\{0\}\\\{0\}\end{bmatrix} & \begin{bmatrix}un\\un\\un\\un\\un\end{bmatrix} & \begin{bmatrix}\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\\\bot&\bot&\bot&\bot&\bot\end{bmatrix} & 1 & \begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\end{bmatrix} & \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} & \begin{bmatrix}\top&\top&\top&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\end{bmatrix} \end{array}$$

(c) $s \in \mathcal{S}(n, t, f)$, $n = 5, t = 3, f = 2$

$$\begin{array}{ccccccc} s'.\mathbf{w} & s'.\mathbf{d} & s'.\mathbf{Msg} & s'.r & s'.\mathbf{cr} & s'.\mathbf{fld} & s'.\mathbf{rcv} \\ \begin{bmatrix}\{0\}\\\{0\}\\\{1\}\\\{0\}\\\{0\}\end{bmatrix} & \begin{bmatrix}un\\un\\un\\un\\un\end{bmatrix} & \begin{bmatrix}\{0\}&\{0\}&\{1\}&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\\\{0\}&\{0\}&\bot&\{0\}&\{0\}\end{bmatrix} & 1 & \begin{bmatrix}\bot\\\bot\\\top\\\bot\\\bot\end{bmatrix} & \begin{bmatrix}\bot\\\bot\\\bot\\\bot\\\bot\end{bmatrix} & \begin{bmatrix}\top&\top&\top&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\\\top&\top&\bot&\top&\top\end{bmatrix} \end{array}$$

(d) $s' \in \mathcal{S}(n, t, f)$, for $n = 5, t = 3, f = 2$

$$(r = t + 1) \wedge (w = \{0\}) \wedge (\forall j\ msg[j] \neq \{1\}) \wedge (\forall j\ msg[j] \neq \{0, 1\}) \ \rightarrow\ w := \{0\}, d := 0$$

(e) A guarded assignment defined for *FloodSet*

$$pr \wedge (w = \{0\}) \wedge (\forall j\ msg[j] \neq \{1\}) \wedge (\forall j\ msg[j] \neq \{0, 1\}) \ \rightarrow\ w := \{0\}, d := 0$$

(f) Predicate abstraction: the termination guard $(r = t + 1)$ is replaced by predicate pr

Fig. 2: The *FloodSet* algorithm

We aim at checking that the algorithms for fault-tolerant agreement satisfy their specifications in the presence of at most t faulty processes, where t satisfies some constraint, e.g., $t < n$. We focus on *crash faults*, exhibited by processes that stop working and cannot restart. As a process can crash in the middle of its execution, it can be the case that it sends a message only to a subset of processes.

We discuss the characteristics of these algorithms by using the *FloodSet* consensus algorithm from Fig. 2a as example. Each process has several variables, e.g., in *FloodSet* each process has the variables d and w. The variable $d \in \{0, 1, un\}$ stores the value the process decides on (un refers to the process being undecided), and $w \subseteq W = \{0, 1\}$ stores the values the process has seen so far (starting with its own initial value, and the ones received in messages). The processes communicate via messages of a finite message alphabet. In *FloodSet*, the message alphabet is the power set of W, and the message that a process sends is the value of its variable w. In the $(t + 1)$-st round, each process decides as follows: if $w = \{v\}$, for some $v \in W$, then $d = v$; otherwise d is assigned a default value v_0. We have to verify that the *FloodSet* algorithm tolerates t process crashes where $t < n$.

2.1 Modeling

We model a distributed algorithm as a transition system that is composed of n processes and an environment. The environment captures the fault model and the round number. The system obtained in such a way is *parameterized* in the parameters n and t, as well as the parameter f which refers to the actual number of crashed processes during an execution of the algorithm. The processes use the values of the parameters n and t; the parameter f is used only by the environment. We are led to distinguish between f and t, as some of our case studies (early deciding consensus) terminate in $min(f + 2, t + 1)$ rounds [9, 11]. Full definitions of the modeling sketched in this section are found in Section 3.

Processes and environment. To model a process, we define *process variables* and *process functions*. The process variables either store values from a finite domain, or are one-dimensional arrays of size n that store information about the other processes, e.g., the messages received in the previous round. The process functions define the way in which the values of the process variables get updated.

The environment describes how processes behave in the presence of crashes and thus it depends only on the fault model. The environment keeps track of the round number, the crashed processes, and for each crashed process, the subset of processes that receive a message from it in the round in which it crashes. The processes and the environment are defined in more detail in Section 3.1.

Global states. The *(global) states* of the parameterized system contain information about the states of the n processes and the environment. For example, in *FloodSet* (e.g., Fig. 2c), we have a one-dimensional array $s.\mathbf{w}$ of size n that store the sets of values w, a one-dimensional array $s.\mathbf{d}$ that stores the decision values d for every process, a two-dimensional array $s.\mathbf{Msg}$ that stores the messages exchanged by the processes, and environment variables: the round number r, the arrays $\mathbf{cr}$ and $\mathbf{fld}$ which store information about process crashes in the current and up to the current round, respectively, and the array $\mathbf{rcv}$, where the (i, j)-th cell flags whether process i received a message from process j in the current round. Fig. 2d shows the global state after process 3 crashes and only send a message (i.e., $\{1\}$) to process 1. The parameterized system is formally defined in Section 3.2.

Global transitions. A transition models the following steps: (i) the environment increments the round number, and non-deterministically decides on new crashes and new receiver lists; (ii) every process computes a message, which is delivered by the environment (depending on the values of the environment variables); (iii) every correct process updates its finite domain variables, using a set of *guarded assignments* (described below), and its array variables.

The language of *guarded assignments* that we define is powerful enough to capture constructs that typically occur in synchronous distributed algorithms, such as conditional constructs and iteration over process ids. For instance, one can check if there is a process j from which a message was received in the current and the previous round. This construct is used in *early deciding/stopping* consensus algorithms. The guards that compare the round number against a parameter, which we call *termination guards*, are typically used in synchronous agreement algorithms to check whether a certain round is reached, i.e., whether it is safe for a correct process to make a decision (e.g., line 6 of the pseudocode). The guarded assignments are formalized in Section 3.1. We introduce guarded assignments in order to perform the abstraction steps syntactically, more details of which can be found in Section 4.3.

A guarded assignment defined for the *FloodSet* algorithm is given in Fig. 2e. It defines the update of the finite domain variables w and d in the case when the current round number is equal to $t+1$ (that is, when the processes decide). The guard is a conjunction of smaller guards, the first one of which is a termination guard. This guarded assignment models one possible outcome of the execution of the pseudocode between lines 5 and 8. If the set of values of the process is $\{0\}$, and there are no messages sent to that process that contain the value 1, then in the new control state, the set of values remains the same, and the decision value is set to 0. The remaining guarded assignments that model the pseudocode between lines 5 and 8 follow a similar pattern.

2.2 Abstraction

We build a single abstract finite state system, which is not parameterized, and simulates the behavior of every concrete system. Our abstraction is applied in two steps: first t and f are eliminated using *pattern-based predicate abstraction*, and then n is eliminated using *data and counter abstraction.*

Predicate abstraction. The set of guarded assignments defined for the algorithm can contain termination guards that feature the parameter t. For each such guard, we introduce a Boolean predicate, which is true when the guard is satisfied. For every newly defined predicate in this abstraction step, a constraint that ensures that the predicate is eventually satisfied is introduced. This eliminates the parameter t. The parameter f is eliminated by introducing a constraint which states that the faults eventually stop appearing. The predicate abstraction step is described in more detail in Section 4.1.

Fig. 2f shows the guarded assignment from Fig. 2e in which the termination guard $r = t + 1$ is replaced by a Boolean predicate pr. We add the constraint $\mathsf{F}\, pr$ for this predicate, saying that eventually the $(t+1)$-st round is reached.

Data and counter abstraction. Using ideas from [13,32,40], we fix a small number of processes (two or three), whose behaviors we keep concrete, and abstract the remaining processes depending on the current values of their variables. The choice of the number of fixed processes depends on the properties we are interested in verifying. For the *FloodSet* algorithm, we fix this number to two, as in order to check the agreement property (Fig. 2b), we need to check whether every pair of processes agree on a value. Using data and counter abstraction [24,46], we reduce the size of the array variables in the global state from n to a fixed number, which depends on the number of fixed processes, and the states the remaining processes are in, but is independent of n. The main idea is to store whether there are no processes (*zero*) or at least one process (*many*) that has some particular state. Section 4.2 formally describes the zero-many data and counter abstractions.

Consider the state s' of *FloodSet* in Fig. 2d. We fix processes 1 and 2, and abstract the behavior of processes 3, 4, and 5. Process 3 has a different valuation of the variables w and d than processes 4 and 5, that have the same valuation. Thus, in state s', we say that, e.g., there are *many* processes in the state where $w = \{0\}$ and $d = un$, and there are *zero* processes in the state where $w = \{0, 1\}$ and $d = un$, as there are no processes in s' with these values for the variables.

3 Modeling and Specifications of Synchronous FTDAs

We formalize FTDAs by introducing process variables, process functions, environment variables and parameters n, t, and f. As we consider crash faults, we assume that the parameters satisfy the *resilience condition* $f \le t < n$. In this section we define a transition system $\mathcal{I}(n, t, f) = \langle \mathcal{S}(n, t, f), \mathcal{S}_0(n, t, f), \mathcal{Q}(n, t, f)\rangle$, called an FTDA instance, for each value of n, t, and f that satisfies the resilience condition.

Notation. A *transition system* is a tuple $M = \langle S, S^0, R\rangle$ where S is a set of *states*, $S^0 \subseteq S$ is a set of *initial states*, and $R \subseteq S \times S$ is a *transition relation.* An *execution* is a path in M that starts in an initial state. Typically, states are valuations of some fixed set of variables X. We write $s.x$ for the value of variable $x \in X$ at state s. For $n \in \mathbb{N}$ we write $[n]$ for the set $\{1, 2, \cdots, n\}$.

3.1 Processes and Environment

A *process* is modeled using *process variables* and *process functions.*

Definition 1 (Process variables). *Let V be a finite set of* process variables*, partitioned into* process control variables $\mathit{cntl}(V) = \{x_1, \cdots, x_{|\mathit{cntl}(V)|}\}$ *and* process neighborhood arrays $\mathit{nbhd}(V) = \{y_1, \cdots, y_{|\mathit{nbhd}(V)|}\}$*. For $v \in V$, let D_v denote the finite set of* values *that v can take if $v \in \mathit{cntl}(V)$, or that each cell in v can take if $v \in \mathit{nbhd}(V)$. We assume that for every $y \in \mathit{nbhd}(V)$, the domain D_y contains a special null value $\bot$ which signifies an* empty cell.

A special neighborhood array is $\mathit{msg} \in \mathit{nbhd}(V)$, which is used to store the messages the process receives in the current round. For convenience, we write M instead of D_{msg}, and call it the message alphabet*.*

Definition 2 (Process states). *A* process state *p is a valuation of all the variables in V, i.e., an element of $P = \prod_{x\in \mathit{cntl}(V)} D_x \times \prod_{y\in \mathit{nbhd}(V)} (D_y)^n$. We write p.control, called a* process control state*, for the valuation restricted to $\mathit{cntl}(V)$. Let $C = \prod_{x\in \mathit{cntl}(V)} D_x$ denote the set of all process control states, and let $C_0 \subseteq C$ denote a set of* initial control states*.*

We define the domain and range of three *process functions*, the last one being parameterized by n and r (the round number).

Definition 3 (Process functions). *Let F be the set of* process functions $F = \{\mathit{snd\_msg}\} \cup \{h_y : y \in \mathit{nbhd}(V) \setminus \{\mathit{msg}\}\} \cup \{\mathit{update}_{n,r} : n, r \in \mathbb{N}\}$*, where*

$snd\_msg : C \to M$ maps process control states to messages; $h_y : M \to D_y$ maps messages to values in D_y and satisfies the restriction that $h_y(\bot) = \bot$; and $update_{n,r} : P \to C$ maps process states to control states.

We use process functions to formally break down and encode the algorithm executed by the processes. Note that the functions *snd_msg* and h_y (for every y) are fixed and finite, whereas $update_{n,r}$ is parameterized by n and r and represents an infinite set of finite functions. This infinite set of functions is defined using a finite set of guarded assignments from the following language.

Each *guarded assignment* is of the form $g \to asg$, where g is a *guard* and *asg* is an *assignment*. An *assignment asg* is a partial function with domain $cntl(V)$ such that if $asg(x)$ is defined then $asg(x) \in D_x$. The *guards* are Boolean combinations (negation and conjunction) of *basic guards*, and are evaluated over process states. The following are the basic guards:

guard	notation		evaluation
empty	$g^{\mathbf{true}}$		**true**
control	$g^{x,v}$	where $x \in cntl(V)$ and $v \in D_x$	$x = v$
termination	$g^{\triangleright,\phi(n,t)}$	where $\triangleright \in \{>, =\}$, and $\phi(n,t)$ is a linear combination of n and t	$r \triangleright \phi(n,t)$
neighborhood	g^{Ψ}	where Ψ is a set of triples $(y, \Box, v)$ s.t. $y \in nbhd(V), \Box \in \{=, \neq\},$ and $v \in D_y$	$\exists j \in [n]$ $\bigwedge_{\Psi}(y[j] \,\Box\, v)$

We write $p \models g$ to signify that process state p satisfies the guard g.

Given a guarded assignment $g \to asg$ and parameters n, r, we define the induced function $update_{n,r}$ as follows. Let $p \in P$ be a process state. If $p \not\models g$ then $update_{n,r}(p) = p.control$. If $p \models g$ then $update_{n,r}(p) = c$ where $c.x = p.x$ if $asg(x)$ not defined, and $c.x = asg(x)$ otherwise.

To fully characterize the function $update_{n,r}$, we associate with it a finite set G of guarded assignments, where the guards are pairwise mutually exclusive.

The guards capture various constructs found in the distributed computing literature. For example, the empty guard captures simple assignments, Boolean combinations of control and termination guards capture conditionals, and Boolean combinations of the neighborhood guards capture iteration over process ids when traversing the process neighborhood arrays.

Since the process functions serve as the building blocks of the transition relation (as we formally explain later), in Section 4 where we abstract the transition system, we will also have to abstract the process functions. Towards this end, a key step will involve abstracting the guarded assignments. This step is done syntactically, by defining abstract versions of the basic guards.

Definition 4 (Environment variables V^e). *The environment variables are: r, with domain $D^e_r = \mathbb{N}$, is the current round number; $\mathbf{cr}$, with domain $D^e_{\mathbf{cr}} = \{\bot, \top\}^n$, flags the crashed processes in the current round, with the value $\top$ indicating a crash; $\mathbf{fld}$, with domain $D^e_{\mathbf{fld}} = \{\bot, \top\}^n$, flags the processes that crashed in some previous round; and $\mathbf{rcv}$, with domain $D^e_{\mathbf{rcv}} = \{\bot, \top\}^{n \cdot n}$, stores a receivers list for every process, that defines the subset of processes to which the* *process sends a message in the current round, with the value $\top$ in the (i,j)-th cell indicating that process i receives the message from process j.*

3.2 FTDA Instance $\mathcal{I}(n,t,f)$

We define the transition system $\mathcal{I}(n,t,f) = \langle \mathcal{S}(n,t,f), \mathcal{S}_0(n,t,f), \mathcal{Q}(n,t,f)\rangle$, as a combination of n processes and the environment, as follows.

Global states $\mathcal{S}(n,t,f)$. The set $\mathcal{S}(n,t,f)$ of *global states* of an FTDA instance is the set of all possible valuations of the following *FTDA variables* $\mathcal{V}$:

Definition 5 (FTDA variables). *The set $\mathcal{V}$ is the union of the sets of:*

- control variables $\mathit{cntl}(\mathcal{V})$*, containing one-dimensional array variables* $\mathbf{x}$ *of size n, that range over $(D_x)^n$, where $x \in \mathit{cntl}(V)$ is a process control variable.*
- neighborhood arrays $\mathit{nbhd}(\mathcal{V})$*, containing two-dimensional array variables* $\mathbf{Y}$ *of size $n \times n$, ranging over $(D_y)^{n\cdot n}$, with $y \in \mathit{nbhd}(V)$ a process neighborhood array. The neighborhood array corresponding to the process neighborhood array msg is denoted* $\mathbf{Msg}$*, and is called the* message channel*.*
- *environment variables V^e.*

Intuitively, the variables in $\mathit{cntl}(\mathcal{V}) \uplus \mathit{nbhd}(\mathcal{V})$ are used to store the values of the process variables of each of the n processes in the FTDA instance, and the value of $\mathbf{Msg}[i,j]$ is equal to the value of $msg[j]$ of process i.

To define the rest of the FTDA instance, we need the following notations. For a global state s and $i \in [n]$, we denote by:

- $s.\mathit{control}_i$ the tuple $\langle s.\mathbf{x}_1[i], \ldots, s.\mathbf{x}_{|\mathit{cntl}(\mathcal{V})|}[i]\rangle \in C$;
- $s.\mathit{row}_i^{\mathbf{Y}}$ the tuple $\langle s.\mathbf{Y}[i,1], \ldots, s.\mathbf{Y}[i,n]\rangle \in (D_y)^n$ (for $\mathbf{Y} \in \mathit{nbhd}(\mathcal{V})$);
- $s.\mathit{local}_i$ the tuple $\langle s.\mathit{control}_i, s.\mathit{row}_i^{\mathbf{Y}_1}, \ldots, s.\mathit{row}_i^{\mathbf{Y}_{|\mathit{nbhd}(\mathcal{V})|}}\rangle \in P$.
- $s.\mathit{location}_i$ the tuple $\langle s.\mathit{control}_i, s.\mathbf{fld}[i]\rangle \in \mathit{Loc}$, where $\mathit{Loc} = C \times \{\bot, \top\}$ is the set of *process locations*.

A *process location* is a pair of the process control state and a failure flag fld, whose value is stored in the environment variable **fld**. As we will see in Section 4, we need the notion of process location in our abstractions, as we need to distinguish between correct and crashed processes that are in the same control state.

Initial global states $\mathcal{S}_0(n,t,f)$. A global state s is *initial* if the values it assigns to the different variables satisfy the following restrictions: the values of the control variables are initial, i.e., $s.\mathit{control}_i \in C_0$ for every $i \in [n]$ (recall that C_0 is the set of initial control states); all the cells of all the neighborhood arrays are empty, i.e., $s.\mathbf{Y}[i,j] = \bot$ for all $i,j \in [n]$ and all $\mathbf{Y} \in \mathit{nbhd}(\mathcal{V})$; and the environment variables are initialized as follows: (i) $s.r = 0$, (ii) $s.\mathbf{cr}[i] = \bot$, for all $i \in [n]$, (iii) $s.\mathbf{fld}[i] = \bot$, for all $i \in [n]$, and (iv) $s.\mathbf{rcv}[i,j] = \bot$, for all $i,j \in [n]$.

Transition relation $\mathcal{Q}(n,t,f)$. We define three transition relations: $\xrightarrow{\textsf{ENV}}$ updates the environment variables; $\xrightarrow{\textsf{MEP}}$ captures the message exchange phase; $\xrightarrow{\textsf{PROC}}$ updates the control variables and neighborhood arrays. A transition of the FTDA instance is an element of the composition $\xrightarrow{\textsf{ENV}}\xrightarrow{\textsf{MEP}}\xrightarrow{\textsf{PROC}}$, i.e., $(s, s''') \in \mathcal{Q}(n,t,f)$ iff there exist states $s', s'' \in \mathcal{S}(n,t,f)$ such that $s \xrightarrow{\textsf{ENV}} s' \xrightarrow{\textsf{MEP}} s'' \xrightarrow{\textsf{PROC}} s'''$.

Updating environment variables. We define $s \xrightarrow{\textsf{ENV}} s'$ as follows. First, the round number is incremented, i.e. $s'.r = s.r + 1$.

Second, the environment chooses which processes will crash in the current round, while keeping the number of crashed processes below the parameter f. That is, $s'.\mathbf{cr}$ is updated to a value that satisfies the following conditions: (i) for every $i \in [n]$, we have $s'.\mathbf{cr}[i] = \bot$ if $s.\mathbf{fld}[i] = \top$, and (ii) $|\{i \in [n] \mid s.\mathbf{fld}[i] \vee s'.\mathbf{cr}[i]\}| \leq f$. Intuitively, condition (ii) reflects the non-deterministic assignment of values to $\mathbf{cr}$, by allowing at most f processes to be flagged as crashed in every execution.

Finally, the receiver lists for the next round are updated by flagging that no message is received from processes that crashed in some previous round, receiving all messages from the correct processes, and non-deterministically choosing which processes receive messages from the processes that crash in the current round. That is, for every $i, j \in [n]$, the following holds for $s'.\mathbf{rcv}$: (i) if $s.\mathbf{fld}[j] = \top$ then $s'.\mathbf{rcv}[i,j] = \bot$, and (ii) if $s.\mathbf{fld}[j] = \bot$ and $s'.\mathbf{cr}[j] = \bot$, then $s'.\mathbf{rcv}[i,j] = \top$.

Message exchange phase. In this transition, the cell (i,j) of the message channel is assigned the message sent from process j to process i, if i is in the receiver list of j for this round. We define $s \xrightarrow{\textsf{MEP}} s'$ if (i) $s'.\mathbf{Msg}[i,j] = \mathit{snd\_msg}(s.\mathit{control}_j)$ if $\mathbf{rcv}[i,j] = \top$, and (ii) $s'.\mathbf{Msg}[i,j] = \bot$ if $\mathbf{rcv}[i,j] \neq \top$.

Updating process variables. In this transition, the failure flags are updated, and every correct process first applies the process function $\mathit{update}_{n,r}$ to update its control variables, and then updates its neighborhood arrays (except for msg) using the messages it received.

We define $s \xrightarrow{\textsf{PROC}} s'$ as follows. First, the failure flags are updated, i.e., for all $i \in [n]$, $s'.\mathbf{fld}[i] = s.\mathbf{fld}[i] \vee s.\mathbf{cr}[i]$. Second, the control variables are updated as follows: (i) for all $i \in [n]$, $s'.\mathit{control}_i = \mathit{update}_{n,s.r}(s.\mathit{local}_i)$ if $s'.\mathbf{fld}[i] = \bot$; and (ii) $s'.\mathit{control}_i = s.\mathit{control}_i$ otherwise. Third, the neighborhood arrays are updated as follows: for every $i \in [n]$ and every $\mathbf{Y} \in \mathit{nbhd}(\mathcal{V}) \setminus \{\mathbf{Msg}\}$: (i) $s'.\mathbf{Y}[i,j] = h_y(s.\mathbf{Msg}[i,j])$, for all $j \in [n]$, if $s'.\mathbf{fld}[i] = \bot$, and (ii) $s'.\mathbf{Y}[i,j] = s.\mathbf{Y}[i,j]$, for all $j \in [n]$, otherwise. Finally, the the message channel is flushed, i.e., $s'.\mathbf{Msg}[i,j] = \bot$, for every $i, j \in [n]$.

Definition 6 (FTDA instance). *Given process variables V, process functions F, environment variables V^e, and parameter values $n, t, f \in \mathbb{N}$, such that $f \leq t < n$, we define the* FTDA instance $\mathcal{I}(n,t,f)$ *to be the transition system $\langle \mathcal{S}(n,t,f), \mathcal{S}_0(n,t,f), \mathcal{Q}(n,t,f) \rangle$.*

3.3 Specification Language

We use a fragment of indexed linear temporal logic [7, 19] to encode the specifications of distributed algorithms. We define its semantics w.r.t. n processes and a set of Boolean predicates Pred. The state of each process is given by the valuations of a set of variables Vars, where each variable $z \in \mathsf{Vars}$ has an associated domain D_z. A global state ς is given by valuations $\varsigma.\mathbf{z}[i]$ for each variable z and process i and a truth-value assignment $\varsigma.q \in \{\top, \bot\}$ for each $q \in \mathsf{Pred}$. We consider atomic propositions of the form $([z = v], i)$, where $z \in \mathsf{Vars}$, $v \in D_z$ and i is an index, and predicates $q \in \mathsf{Pred}$. We use the following fragment of indexed-LTL:

Definition 7 (The fragment $\mathcal{F}_l$). *For $l \in \mathbb{N}$, we write $\mathcal{F}_l$ for the set of indexed-LTL formulas of the form $\forall i_1 \forall i_2 \cdots \forall i_l . \psi$, where (1) ψ only contains $\exists$-quantifiers, and (2) an $\exists$-quantifier only appears in subformulas of the form $\exists i.([z = v], i)$.*

The semantics of an atomic proposition in a state ς is defined by $\varsigma \models ([z = v], i)$ iff $\varsigma.\mathbf{z}[i] = v$. We define the semantics of a predicate $q \in \mathsf{Pred}$ as follows: $\varsigma \models q$ iff $\varsigma.q = \top$. The semantics of the logical connectives, quantifiers and temporal operators is standard. We will also write $\mathbf{z}[i] = v$ instead of $([z = v], i)$.

The fragments $\mathcal{F}_l$, for $l \in \mathbb{N}$, are rich enough to capture specifications of fault-tolerant agreement. To express specifications of FTDA instances, we define $\mathsf{Vars} = cntl(V) \cup \{cr, fld\}$, where cr, fld are the flags stored in the environment variables $\mathbf{cr}, \mathbf{fld}$. Below, we formalize the specifications stated in Fig. 2b, which are evaluated over global states $s \in \mathcal{S}(n, t, f)$:

- **Validity:** If there is no process with an initial value different from 0, then 0 is the only decision value (there is a symmetric specification for $w = \{1\}$):

$$\forall i.\ (\exists j . \mathbf{w}[j] \neq \{0\}) \vee \mathsf{G}((\mathbf{fld}[i] = \bot \wedge \mathbf{d}[i] \neq un) \rightarrow \mathbf{d}[i] = 0)$$

- **Agreement:** No two correct processes decide differently:

$$\forall i \forall j.\ \mathsf{G}((\mathbf{fld}[i] = \bot \wedge \mathbf{fld}[j] = \bot \wedge \mathbf{d}[i] \neq un \wedge \mathbf{d}[j] \neq un) \rightarrow \\ ((\mathbf{d}[i] = 0 \wedge \mathbf{d}[j] = 0) \vee (\mathbf{d}[i] = 1 \wedge \mathbf{d}[j] = 1)))$$

- **Termination**: Every correct process eventually decides:

$$\forall i.\ \mathsf{F}(\mathbf{fld}[i] = \bot \rightarrow \mathbf{d}[i] \neq un)$$

Note that the above stated formulas do not use Boolean propositions, i.e., $\mathsf{Pred} = \emptyset$ (also the global states of FDTA instances do not contain Boolean variables). In Section 4.1, we will state formulas that use Boolean propositions and define abstract transition systems that have Boolean variables.

3.4 Parameterized Model Checking

The *parameterized model checking question* is to decide, given process variables V, process functions F, environment variables V^e, and a specification $\varphi \in \mathcal{F}_l$, whether φ is satisfied in every FTDA instance $\mathcal{I}(n, t, f)$ such that $f \leq t < n$.

3.5 Symmetry

We observe that the FTDA instance $\mathcal{I}(n,t,f)$ is a symmetric transition system [20]. Due to the symmetry, we can fix a small number m of processes that represent any m processes among the n processes of the FTDA instance. To determine m, we take the maximal number of $\forall$-quantifiers that appear in the specifications expressed in our fragment of indexed-LTL. For example, the validity and termination consensus specifications (cf. Section 3.3) have a single $\forall$-quantifier, while agreement has two. Therefore, for consensus we set $m = 2$.

Once we fix m, for every indexed-LTL formula φ, where the indices range over $[n]$, we define a formula φ^m, where the indices bound by $\forall$-quantifiers range over $[m]$, and the indices bound by $\exists$-quantifiers range over $[n]$. We denote by $\mathcal{F}_l^m$ the set of indexed-LTL formulas $\{\varphi^m \mid \varphi \in \mathcal{F}_l \text{ and } l \leq m\}$.

Proposition 1 (Symmetry). *The indexed-LTL specification φ is satisfied in an FTDA instance $\mathcal{I}(n,t,f)$ if φ^m is satisfied in $\mathcal{I}(n,t,f)$.*

4 Abstracting Synchronous FTDAs

We define the pattern-based predicate abstraction in Section 4.1, and the zero-many data and counter abstraction in Section 4.2. As these definitions are not effective, in Section 4.3 we give an effective method for processes defined by the process functions in Section 3.1. The challenge lies in the fact that we need to abstract a family of systems parameterized by n, t, and f.

4.1 Predicate Abstraction: Eliminating t and f

Recall that the parameter f refers to the actual number of processes that crash during a run of an instance $\mathcal{I}(n,t,f)$, and the parameter t appears in the termination guards. To build a system $\mathcal{J}(n)$ parameterized only in n, we introduce predicates that abstract basic termination guards, and verification conditions that have to be satisfied in every execution of $\mathcal{J}(n)$.

Predicates. We introduce k Boolean predicates $pr_1, \ldots, pr_k$, where k is the number of basic termination guards of the form $r \rhd \phi(n,t)$, appearing in the set G of guarded assignments that define the function $update_{n,r}$. Each predicate pr_j, for $j \in [k]$, is true whenever the basic termination guard it replaces is satisfied. By replacing the basic termination guards with predicates in $\mathcal{J}(n)$, we eliminate the variable variable r from environment, as in $\mathcal{I}(n,t,f)$, the variable r occurs only in the basic termination guards. Thus, the set $\mathcal{V}$ of variables of $\mathcal{J}(n)$ contains:

- control variables $\mathbf{x} \in cntl(\mathcal{V})$, ranging over $(D_x)^n$;
- neighborhood arrays $\mathbf{Y} \in nbhd(\mathcal{V})$, ranging over $(D_y)^{n \cdot n}$;
- environment variables $\mathbf{cr}, \mathbf{fld}, \mathbf{rcv}$.

Additionally, for $\mathcal{J}(n)$, we introduce a set $\mathsf{Pred} = \{pr_1, \ldots, pr_k\}$ of Boolean predicates. The set $\Sigma(n)$ of states of $\mathcal{J}(n)$ is the set of all valuations of $\mathcal{V}$. The following abstraction mapping α maps states of $\mathcal{I}(n,t,f)$ to states from $\mathcal{J}(n)$.

Definition 8 (Abstraction mapping α). *We define the abstraction mapping $\alpha : \mathcal{S}(n,t,f) \to \Sigma(n)$ as: $\sigma.\mathbf{x} = s.\mathbf{x}$, for all $\mathbf{x} \in cntl(\mathcal{V})$, $\sigma.\mathbf{Y} = s.\mathbf{Y}$, for all $\mathbf{Y} \in nbhd(\mathcal{V})$, $\sigma.\mathbf{cr} = s.\mathbf{cr}$, $\sigma.\mathbf{fld} = s.\mathbf{fld}$, $\sigma.\mathbf{rcv} = s.\mathbf{rcv}$ and for every $j \in [k]$, $\sigma.pr_j = \top$, if the basic termination guard $r \rhd_j \phi_j(n,t)$ is satisfied in the state s, and $\sigma.pr_j = \bot$, otherwise.*

Pattern-based verification conditions. We define a set of indexed-LTL formulas $\mathcal{C}$ of *verification conditions* that we impose on $\mathcal{J}(n)$. The formulas in $\mathcal{C}$ introduce restrictions on how the predicates in pr and the crash flags in $\mathbf{cr}$ are assigned values in $\mathcal{J}(n)$ in a way that reflect behaviors of the concrete executions. Note that these verification conditions are imposed on the environment, and can therefore be reused across algorithms that operate under the same environment.

Let *clean* denote the formula $\neg(\exists i\ \mathbf{cr}[i] = \top)$. The formula *clean* is satisfied in a state $\sigma \in \Sigma(n)$, if there is no process that has been flagged as newly crashed in σ. We list the conditions that we identified in multiple benchmarks, together with an explanation of why they hold.

$\mathsf{FG}\,clean$ ensures that from some time on, there are no more crashes. It holds because f is finite in each instance.

$\mathsf{F}\,(pr_j)$ and $\mathsf{FG}\,(pr_j)$ for $j \in [k]$ where $\rhd_j$ is $=$ and $\rhd_j$ is $>$, respectively. For instance, the termination guard of *FloodSet* in Fig. 2 is $r = t+1$, and evaluates to true once (in round $t+1$), while a guard $r > t$ becomes and stays true.

$(\bigwedge_j \neg pr_j)\mathsf{U}clean$ ensures that the basic termination guards become true only after a clean round has occurred. This is typical for consensus algorithms that use a guard $r = t+1$ and are designed for $f \le t$ faults. In this case, in at least one of the $t+1$ rounds no process crashes, i.e., the round is clean.

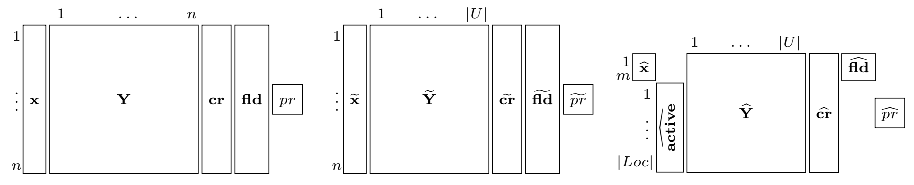
While currently we perform this abstraction step manually, in the future we aim at developing an automatic procedure for such verification conditions.

Given $\mathcal{I}(n,t,f)$ and $\Sigma(n)$, let $\mathcal{J}(n)$ be the *overapproximation* [14] of $\mathcal{I}(n,t,f)$ induced by α. Let $\mathcal{C}$ be a set of constraints that are satisfied in all instances $\mathcal{I}(n,t,f)$, for $f \le t < n$ (e.g., for *FloodSet*, $\mathcal{C} = \{\mathsf{FG}\,clean, \mathsf{F}\,pr, pr\mathsf{U}clean\}$). Let $\chi_{\mathcal{C}}$ be the conjunction of all formulas in $\mathcal{C}$. As there are no $\forall$-quantifiers in $\chi_{\mathcal{C}}$, the formula $\chi_{\mathcal{C}} \to \varphi^m$ is also in $\mathcal{F}_l^m$.

Proposition 2 (Soundness of α). *For every $n \in \mathbb{N}$, if $\mathcal{J}(n) \models \chi_{\mathcal{C}} \to \varphi^m$ then $\mathcal{I}(n,t,f) \models \varphi^m$, for all $t, f \in \mathbb{N}$ s.t. $f \le t < n$.*

4.2 Zero-many Data and Counter Abstraction

The system $\mathcal{J}(n)$ obtained after applying predicate abstraction is still parameterized in n, i.e., the size of its array variables depends on n. To build a finite system independent of n, we fix the size of the array variables. We proceed by fixing m processes and abstracting the remaining $n - m$ processes based on their process location (defined in Section 3.2). That is, for all process locations $\ell \in Loc$, we store whether no process from the $n - m$ processes is in location ℓ (zero), or whether at least one process from the $n - m$ processes is in location ℓ (many).



(a) $\sigma \in \Sigma(n)$ (b) $\widetilde{\sigma} \in \widetilde{\Sigma}(n)$ (c) $\widehat{\sigma} \in \widehat{\Sigma}$

Fig. 3: Two-step zero-many abstraction (for $|cntl(\mathcal{V})| = |nbhd(\mathcal{V})| = 1$, and omitting $\mathbf{rcv}$ for space reasons) with m fixed processes. **(a)** illustrates a state σ of $\mathcal{J}(n)$; **(b)** shows the result of applying $\widetilde{\alpha}_n$, where $U = [m] \cup Loc$. $\widetilde{\mathbf{Y}}[i, v]$ stores the set of values in row i of $\mathbf{Y}$ for all columns j witnessed by v; **(c)** shows the result of applying $\widehat{\alpha}_n$, i.e., $\widehat{\mathbf{x}}$ and $\widehat{\mathbf{fld}}$ store the control variable x and the failure flag of the m processes, $\widehat{\mathbf{active}}[u]$ stores if there are zero or many processes witnessed by $u \in Loc$, $\widehat{\mathbf{Y}}[u, v]$ stores the union of values $\widetilde{\mathbf{Y}}[i, v]$ for processes i witnessed by u, and $\widehat{\mathbf{cr}}[u]$ stores the union of $\widetilde{\mathbf{cr}}[i]$ for processes i witnessed by u.

We apply the zero-many abstractions in two steps (Fig. 3). After the first step, the overapproximation still depends on n, but after the second step the overapproximation is finite and independent of n.

Step one: Zero-many data abstraction. In the first step, we fix the size of the two-dimensional arrays from the set $nbhd(\mathcal{V}) \cup \{\mathbf{rcv}\}$ so that the number of their columns depends on Loc and m, and not on n. As a result, we obtain an abstract system $\widetilde{\mathcal{J}}(n)$, that is still parameterized in n.

To build $\widetilde{\mathcal{J}}(n)$, we need the following notation. First, we define a set $U = [m] \cup Loc$ of *indices*, which will be used as indices of the abstract array variables. We say that an index $u \in U$ *witnesses* a process, if it corresponds to one of the fixed m processes, i.e., if $u \in [m]$, or if $u \in Loc$ and there exists a process whose current location is u. In this step, we use the elements of U as indices for the columns of the two-dimensional arrays: each cell in a column indexed by $u \in U$ is a union of the cells in the column indexed by the processes witnessed by u.

Next, we define a mapping $ids : \Sigma(n) \times U \to 2^{[n]}$, that maps a state $\sigma \in \Sigma(n)$ and an index $u \in U$ to the set of processes witnessed by u, i.e., $ids(\sigma, u) = \{u\}$, if $u \in [m]$, and $ids(\sigma, u) = \{i \in [n] \setminus [m] \mid \sigma.location_i = u\}$, if $u \in Loc$.

Finally, we define the set of variables $\widetilde{\mathcal{V}}$ of $\widetilde{\mathcal{J}}(n)$, that contains (i) the control variables $\widetilde{\mathbf{x}} \in cntl(\widetilde{\mathcal{V}})$, ranging over $(D_x)^n$, (ii) the neighborhood arrays $\widetilde{\mathbf{Y}} \in nbhd(\widetilde{\mathcal{V}})$, ranging over $(2^{D_y})^{n \cdot |U|}$, and (iii) the environment variables $\widetilde{\mathbf{cr}}, \widetilde{\mathbf{fld}}, \widetilde{\mathbf{rcv}}$, ranging over $\{\bot, \top\}^n$,$\{\bot, \top\}^n$, and $(2^{\{\bot, \top\}})^{n \cdot |U|}$ respectively. The set $\widetilde{\Sigma}(n)$ of states of $\widetilde{\mathcal{J}}(n)$ is the set of all valuations of $\widetilde{\mathcal{V}}$.

Using this notation, we introduce the abstraction mapping $\widetilde{\alpha}_n : \Sigma(n) \to \widetilde{\Sigma}(n)$, that maps states of $\mathcal{J}(n)$ (Fig. 3a) to states from $\widetilde{\Sigma}(n)$ (Fig. 3b).

Definition 9 (Abstraction Mapping $\widetilde{\alpha}_n$). *We define the abstraction mapping* $\widetilde{\alpha}_n : \Sigma(n) \to \widetilde{\Sigma}(n)$ *as:* $\widetilde{\sigma}.\widetilde{\mathbf{x}} = \sigma.\mathbf{x}$*, for all* $\widetilde{\mathbf{x}} \in cntl(\widetilde{\mathcal{V}})$*;* $\widetilde{\sigma}.\widetilde{\mathbf{cr}} = \sigma.\mathbf{cr}$*;* $\widetilde{\sigma}.\widetilde{\mathbf{fld}} = \sigma.\mathbf{fld}$*;*

$\widetilde{\sigma}.\widetilde{pr}_j = \sigma.pr_j$, for $j \in [k]$; and for all $\widetilde{\mathbf{Y}} \in nbhd(\widetilde{\mathcal{V}}) \cup \{\widetilde{\mathbf{rcv}}\}$, $i \in [n], v \in U$, $\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i,v] = \bigcup\{\sigma.\mathbf{Y}[i,j] \mid j \in ids(\sigma, v)\}$.

Given the system $\mathcal{J}(n)$ and the set $\widetilde{\Sigma}(n)$ of states, we define $\widetilde{\mathcal{J}}(n)$ as the *overapproximation* of $\mathcal{J}(n)$ induced by $\widetilde{\alpha}_n$.

Proposition 3 (Soundness of $\widetilde{\alpha}_n$). *For every $n \in \mathbb{N}$, and a formula $\psi^m \in \mathcal{F}_l^m$, we have that if $\widetilde{\mathcal{J}}(n) \models \psi^m$, then $\mathcal{J}(n) \models \psi^m$.*

Step two: Zero-many counter abstraction In this step, we store the values of the control variables and the failure flags for the m processes in the variables $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$ and $\widehat{\mathbf{fld}}$ respectively, and for the remaining $n-m$ processes, we keep information whether there exists some process from $[n] \setminus [m]$ in some location $\ell \in Loc$ in a newly introduced variable $\widehat{\mathbf{active}}$. Finally, we use the elements from the set U to index the rows of the two-dimensional arrays from the set $nbhd(\widehat{\mathcal{V}}) \cup \{\widehat{\mathbf{rcv}}\}$ and the one-dimensional array $\widehat{\mathbf{cr}}$. Note that the failure flags of the $n-m$ processes are encoded in the process locations. This results in a finite system $\widehat{\mathcal{J}}$, which is not parameterized.

To build $\widehat{\mathcal{J}}$, we first define the mapping $\widetilde{ids} : \widetilde{\Sigma}(n) \times U \to 2^{[n]}$ analogously to the mapping ids above: $\widetilde{ids}(\widetilde{\sigma}, u) = \{u\}$ if $u \in [m]$, and $\widetilde{ids}(\widetilde{\sigma}, u) = \{i \in [n] \setminus [m] \mid \widetilde{\sigma}.location_i = u\}$ otherwise. We define the set $\widehat{\mathcal{V}}$ of variables of $\widehat{\mathcal{J}}$, that contains:

- control variables $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$ of the m fixed processes, ranging over $(D_x)^m$
- the array $\widehat{\mathbf{active}}$, ranging over $\{0, \mathsf{many}\}^{|Loc|}$, that stores for a location $u \in Loc$, whether there are no processes in location u ($\widehat{\mathbf{active}}[u] = 0$), or if there is at least one process in location u ($\widehat{\mathbf{active}}[u] = \mathsf{many}$);
- neighborhood arrays $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}})$, ranging over $(2^{D_y})^{|U|\cdot|U|}$, and
- environment variables $\widehat{\mathbf{cr}}, \widehat{\mathbf{fld}}, \widehat{\mathbf{rcv}}$, ranging over $(2^{\{\bot,\top\}})^{|U|}$, $\{\bot,\top\}^m$, and $(2^{\{\bot,\top\}})^{|U|\cdot|U|}$ respectively.

Using the notation defined above, we define the abstraction mapping $\widehat{\alpha}_n$ that maps an abstract state $\widetilde{\sigma} \in \widetilde{\Sigma}(n)$ (Fig. 3b) to an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ (Fig. 3c).

Definition 10 (Abstraction mapping $\widehat{\alpha}_n$). *We define the abstraction mapping $\widehat{\alpha}_n : \widetilde{\Sigma}(n) \to \widehat{\Sigma}$ as: for $u \in [m]$, $\widehat{\sigma}.\widehat{\mathbf{x}}[u] = \widetilde{\sigma}.\widetilde{\mathbf{x}}[u]$, for all $\widehat{\mathbf{x}} \in cntl(\widehat{\mathcal{V}})$, and $\widehat{\sigma}.\widehat{\mathbf{fld}}[u] = \widetilde{\sigma}.\widetilde{\mathbf{fld}}[u]$; for $u \in Loc$, $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = 0$ if $\widetilde{ids}(\widetilde{\sigma}, u) = \emptyset$, and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$ otherwise; for $u \in U$, $\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{cr}}[i] \mid i \in \widetilde{ids}(\widetilde{\sigma}, u)\}$; for $j \in [k]$, $\widehat{\sigma}.\widehat{pr}_j = \widetilde{\sigma}.\widetilde{pr}_j$; and for all $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \cup \{\widehat{\mathbf{rcv}}\}$, $u, v \in U$, $\widehat{\sigma}.\widehat{\mathbf{Y}}[u,v]$ is $\bigcup\{\widetilde{\sigma}.\widetilde{\mathbf{Y}}[i,v] \mid i \in \widetilde{ids}(\widetilde{\sigma}, u)\}$.*

Given $\widetilde{\mathcal{J}}(n)$ and $\widehat{\Sigma}$, we define the abstract system $\widehat{\mathcal{J}}$ as the overapproximation induced by the mapping $\widehat{\alpha}_n$.

We now define how to evaluate formulas $\psi^m \in \mathcal{F}_l^m$ in states $\widehat{\sigma} \in \widehat{\Sigma}$. As we have removed the parameter n, when evaluating ψ^m in $\widehat{\sigma}$, the indices bound by the $\exists$-quantifier range over the set U of abstract indices, while the indices bound by the $\forall$-quantifier continue to range over the set $[m]$.

Recall that to express specifications of agreement algorithms we defined $\mathsf{Vars} = \mathit{cntl}(V) \cup \{cr, \mathit{fld}\}$, and atomic propositions in a formula of the form $([z = v], i)$ for $z \in \mathsf{Vars}$, $v \in D_z$ and index i. We now define the meaning of the indexed atomic propositions in $\widehat{\sigma} \in \widehat{\Sigma}$, by distinguishing the following cases:

$z \neq cr$. We define $\widehat{\sigma} \models ([z = v], i)$ if (a) $i \in [m]$ and $\widehat{\sigma}.\widehat{\mathbf{z}}[i] = v$; or (b) $i \in \mathit{Loc}$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[i] = \mathsf{many} \wedge i.z = v$;
$z = cr$. We define $\widehat{\sigma} \models ([cr = v], i)$ if $v \in \widehat{\sigma}.\widehat{\mathbf{cr}}[i]$.

Proposition 4 (Soundness of $\widehat{\alpha}_n$). *For every $n \in \mathbb{N}$, and a formula $\psi^m \in \mathcal{F}_l^m$ we have that if $\widehat{\mathcal{J}} \models \psi^m$ then $\widetilde{\mathcal{J}}(n) \models \psi^m$.*

The overall soundness of our approach is a consequence of Propositions 1 – 4.

Theorem 1 (Soundness). *Let $\mathcal{I}(n, t, f)$ be an FTDA instance, and $\widehat{\mathcal{J}}$ the abstract system defined as the overapproximation induced by the abstraction mapping $\widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$. If $\widehat{\mathcal{J}} \models \chi_C \to \varphi^m$, then $\mathcal{I}(n, t, f) \models \varphi$.*

4.3 Abstract Transition Relations

In the previous section we have defined $\widehat{\mathcal{J}} = \langle \widehat{\Sigma}, \widehat{\Sigma}_0, \widehat{\Theta} \rangle$ as the overapproximation of the FTDA instances in $\{\mathcal{I}(n, t, f) \mid f \leq t < n\}$ induced by the abstraction mapping $\delta = \widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$, without giving a constructive definition of the abstract transition relation. In this section we show how to efficiently compute abstract versions of the transition relations from Section 3.2: The abstract transitions $\xrightarrow{\widehat{\mathsf{ENV}}}$ and $\xrightarrow{\widehat{\mathsf{MEP}}}$ are straight-forward abstract encodings of updating the environment variables (e.g., crashing processes), and the message exchange phase, respectively. Encoding the abstract process state update $\xrightarrow{\widehat{\mathsf{PROC}}}$ is more involved: due to the counter abstraction, from an index $u \in U$ we have to decode the location that corresponds to that index, and compute the possible successor locations which we store in a relation $\widehat{\mathit{Active}} \subseteq \mathit{Loc} \times \mathit{Loc}$. We use this relation for updating the array $\widehat{\mathbf{active}}$ and the neighborhood arrays $\widehat{\mathbf{Y}} \in \mathit{nbhd}(\widehat{\mathcal{V}})$.

We adapt several notions that we used throughout this paper. First, given an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ and an index $u \in U$, we say that u *witnesses a process* in $\widehat{\sigma}$ if $u \in [m]$ or if $u \in \mathit{Loc}$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$. Next, we adapt the notions *control*, *row* and *local*. For an abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ and $u \in U$, we denote by:

- $\widehat{\sigma}.\mathit{control}_u$ the tuple $\langle \widehat{\sigma}.\widehat{\mathbf{x}}_1[u], \ldots, \widehat{\sigma}.\widehat{\mathbf{x}}_{|\mathit{cntl}(\widehat{\mathcal{V}})|}[u] \rangle$ if $u \in [m]$, and $u.\mathit{control}$ if $u \in \mathit{Loc}$ and $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$;
- $\widehat{\sigma}.\mathit{row}_u^{\widehat{\mathbf{Y}}}$ the tuple $\langle \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_1], \ldots, \widehat{\sigma}.\widehat{\mathbf{Y}}[u, v_{|U|}] \rangle \in (2^{D_y})^{|U|}$;
- $\widehat{\sigma}.\mathit{local}_u$ the tuple $\langle \widehat{\sigma}.\mathit{control}_u, \widehat{\sigma}.\mathit{row}_u^{\widehat{\mathbf{Y}}_1}, \ldots, \widehat{\sigma}.\mathit{row}_u^{\widehat{\mathbf{Y}}_{|\mathit{nbhd}(\widehat{\mathcal{V}})|}} \rangle \in \widehat{P}$, where $\widehat{P} = C \times \prod_{\widehat{\mathbf{Y}} \in \mathit{nbhd}(\widehat{\mathcal{V}})} (2^{D_y})^{|U|}$ is the set of abstract process states;

Abstract environment update. The transition $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{ENV}}} \widehat{\sigma}'$ is defined as follows. First, the predicates from the set Pred are assigned values non-deterministically.

Second, to define the new crashes, for $u \in U$, the value of $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is set to $\{\bot\}$ if u witnesses a failed process, that is, if $u \in [m]$ and $\widehat{\sigma}.\widehat{\mathbf{fld}}[u] = \top$, or if $u \in Loc$ and $u.fld = \top$. Otherwise, if u witnesses a non-failed process, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u]$ is assigned either $\{\bot\}$ or $\{\top\}$ if $u \in [m]$, and one of the values $\{\bot\}$, $\{\top\}$ or $\{\bot, \top\}$ if $u \in Loc$, non-deterministically. If u does not witness a process, $\widehat{\sigma}'.\widehat{\mathbf{cr}}[u] = \emptyset$.

To build the new receiver lists, for every $u, v \in V$ that witness a process, the value of $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v]$ is set to $\{\bot\}$, if v witnesses a failed process. If v witnesses a crashed process, that is, if $\top \in \widehat{\sigma}'.\widehat{\mathbf{cr}}[v]$, then $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v]$ is assigned one of the values $\{\bot\}$, $\{\top\}$ or $\{\bot, \top\}$ non-deterministically. Otherwise, if v witnesses a correct process, $\widehat{\sigma}'.\widehat{\mathbf{rcv}}[u, v] = \{\top\}$. The cells of $\widehat{\sigma}'.\widehat{\mathbf{rcv}}$ indexed by indices from U that do not witness a process are set to $\emptyset$.

Abstract message exchange phase. A transition $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{MEP}}} \widehat{\sigma}'$ is taken if (i) $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v]$ contains $snd\_msg(\widehat{\sigma}.control_v)$, for $u, v \in U$ such that $\top \in \widehat{\sigma}.\widehat{\mathbf{rcv}}[u, v]$, (ii) it contains $\bot$, if $\bot \in \widehat{\sigma}.\widehat{\mathbf{rcv}}[u, v]$, and (iii) $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v] = \emptyset$ in the remaining cases.

Abstract process variable update. To define how the control states are updated in the abstract system $\widehat{\mathcal{J}}$, we define abstract guarded assignments. The *abstract guarded assignments* are of the form $\widehat{g} \to \widehat{asg}$, where $\widehat{g}$ is a Boolean combination of *abstract basic guards*, and $\widehat{asg}$ is a partial function, defined in the same way as in the concrete case. We have the following *abstract basic guards*:

guard	notation		evaluation
empty	$g^{\mathbf{true}}$		**true**
control	$g^{x,v}$	where $x \in cntl(V)$ and $v \in D_x$	$control_u.x = v$
termination	$g^{\widehat{pr}}$	where $\widehat{pr}$ abstracts $r \rhd \phi(n, t)$	$\widehat{pr}$
neighborhood	g^{Ξ}	where Ξ is a set of triples $(\widehat{\mathbf{Y}}, \Box, v_y)$ s.t. $\widehat{\mathbf{Y}} \in nbhd(V), \Box \in \{\in, \notin\}$, and $v_y \in D_y$	$\exists v \in U$ $\bigwedge_{\Xi}(v_y \Box \widehat{\mathbf{Y}}[u, v])$

The abstract guards are evaluated over $local_u$, for $u \in U$. We write $local_u \models \widehat{g}$ if the abstract guard $\widehat{g}$ is satisfied in the abstract process state $local_u$.

The control state update of the fixed m processes is analogous to the concrete case: a set $\widehat{G}_m$ of abstract guarded assignments with pairwise mutually exclusive guards defines a function $update_m : \widehat{P} \to C$.

To update the control states of processes witnessed by $u \in Loc$, we define a set $\widehat{G}_{Loc}$ of guarded assignments, where the guards are not pairwise mutually exclusive. The set $\widehat{G}_{Loc}$ defines a function $update_{Loc}$, which returns a set of control states. Intuitively, processes that are witnessed by the same location may update to different control states in the concrete system, depending on the neighborhood arrays and the environment. This is why, in the set $\widehat{G}_{Loc}$, there can be multiple guarded assignments with the same guard, but different assignments.

Definition 11 **(*$update_{Loc}$*).** *The function $update_{Loc} : \widehat{P} \to 2^C$ maps abstract local states to subsets of the set C of control states. For $u \in Loc$, we define $update_{Loc}(local_u) = \{c \mid \exists \widehat{g} \to \widehat{asg} \in \widehat{G}_{Loc}$ s.t. $local_u \models \widehat{g}$ and $\widehat{asg}$ results in $c\}$.*

To update the array $\widehat{\mathbf{active}}$, we define the following relation.

Definition 12 ($\widehat{Active}$)**.** *A pair (u, u') of locations from Loc are in relation $\widehat{Active} \subseteq Loc \times Loc$ if $\widehat{\sigma}.\widehat{\mathbf{active}}[u] = \mathsf{many}$ and either: (i) $u.failed = \top$ and $u = u'$, or (ii) $u.failed = \bot$ and $\top \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$ and $u' = \langle \widehat{\sigma}.control_u, \top\rangle$, or (iii) $u.failed = \bot$ and $\bot \in \widehat{\sigma}.\widehat{\mathbf{cr}}[u]$ and $u' \in update_{Loc}(\widehat{\sigma}.local_u)$.*

The relation $\widehat{Active}$ is used to update the neighborhood arrays $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \setminus \{\widehat{\mathbf{Msg}}\}$, as the update of the locations implies update in the indices of the neighborhood arrays. When updating $\widehat{\mathbf{Y}}$, different cases based on whether u, v are in $[m]$ of in *Loc* are distinguished. For example, if $u \in [m]$ and $v \in Loc$ then $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v]$ is the union of the sets $\{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u, v_o]\}$ where h_y is the process function for updating $\mathbf{Y}$, $v_o \in Loc$ is the old location that updated to the new location v, and $(v_o, v) \in \widehat{Active}$.

Finally, for two states $\widehat{\sigma}, \widehat{\sigma}' \in \widehat{\Sigma}$, it holds that $\widehat{\sigma} \xrightarrow{\widehat{\mathsf{PROC}}} \widehat{\sigma}'$ if:

1. for $u \in [m]$, we have $\widehat{\sigma}'.\widehat{\mathbf{fld}}[u] = \widehat{\sigma}.\widehat{\mathbf{fld}}[u] \vee (\widehat{\sigma}.\widehat{\mathbf{cr}}[u] = \{\top\})$;
2. for $u \in [m]$, we have $\widehat{\sigma}'.control_u = update_m(\widehat{\sigma}.local_u)$ if $\widehat{\sigma}'.\widehat{\mathbf{fld}}[u] = \bot$, and $\widehat{\sigma}'.control_u = \widehat{\sigma}.control_u$ otherwise;
3. for $u \in Loc$, we have $\widehat{\sigma}'.\widehat{\mathbf{active}}[u] = 0$ if $\forall v \in Loc\ (v, u) \notin \widehat{Active}$, and $\widehat{\sigma}'.\widehat{\mathbf{active}}[u] = \mathsf{many}$ otherwise;
4. for $u, v \in U$ and $\widehat{\mathbf{Y}} \in nbhd(\widehat{\mathcal{V}}) \setminus \{\widehat{\mathbf{Msg}}\}$, we have:
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \{h_y(d) \mid d \in \widehat{\mathbf{Msg}}[u, v]\}$, if $u, v \in [m]$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{(v_o,v)\in\widehat{Active}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u, v_o]\}$, if $u \in [m], v \in Loc$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{(u_o,u)\in\widehat{Active}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u_o, v]\}$, if $u \in Loc, v \in [m]$;
   - $\widehat{\sigma}'.\widehat{\mathbf{Y}}[u, v] = \bigcup_{\substack{(u_o,u)\in\widehat{Active}\\(v_o,v)\in\widehat{Active}}} \{h_y(d) \mid d \in \widehat{\sigma}.\widehat{\mathbf{Msg}}[u_o, v_o]\}$, if $u, v \in Loc$.
5. for $u, v \in U$, we have $\widehat{\sigma}'.\widehat{\mathbf{Msg}}[u, v] = \{\bot\}$, if u, v witness a process in $\widehat{\sigma}'$, and $\widehat{\sigma}'.\widehat{\mathbf{Msg}} = \emptyset$ otherwise.

Theorem 2 (**Simulation**)**.** *Let $\widehat{\mathcal{I}}$ be the overapproximation of $\mathcal{I}(n, t, f)$ induced by the abstraction mapping $\delta = \widehat{\alpha}_n \circ \widetilde{\alpha}_n \circ \alpha$. Suppose $(s, s''') \in \mathcal{Q}(n, t, f)$, such that there exist $s', s'' \in \mathcal{S}(n, t, f)$ with $s \xrightarrow{\mathsf{ENV}} s' \xrightarrow{\mathsf{MEP}} s'' \xrightarrow{\mathsf{PROC}} s'''$. Then it holds that $\delta(s) \xrightarrow{\widehat{\mathsf{ENV}}} \delta(s') \xrightarrow{\widehat{\mathsf{MEP}}} \delta(s'') \xrightarrow{\widehat{\mathsf{PROC}}} \delta(s''')$.*

5 Benchmarks and Experiments

We encoded several synchronous FTDAs from the literature in TLA+ [33] and used the model checker TLC [50]. The experiments were run on a machine with two 12-core Intel(R) Xeon(R) E5-2650 v4 CPUs and 256 GB RAM.

Our benchmarks contain algorithms that solve different variants of the consensus problem, the k-set agreement problem, and the atomic commitment problem;

Table 1: Experimental results for parameterized model checking

algorithm	problem	reference	m	$\mathcal{I}(n,t,f)$ with $t \leq n-m$		m'	$\mathcal{I}(n,t,f)$ with $n-m < t < n$	
				states	time		states	time
FloodSet	consensus	[37, p.103]	2	210 583	2min 28s	1	17 911	11s
FC	fair consensus	[47, p. 17]	2	160 523	3min	1	26 967	18s
EDAC	early deciding consensus	[11]	2	416 120	4h 35min	1	35 027	2min 28s
ESC	early stopping consensus	[47, p. 38]	2	163 772	44min 30s	1	12 784	1min 19s
NBAC	non-blocking atomic commit	[47, p. 82]	2	69 845	40s	1	4 981	5s
FloodMin	k-set agreement, for $k = 2$	[37, p.163]	3	10 116 820	10d 16h	2	512 861	1h 39min
						1	43 601	2min 2s

Table 2: Experimental results for the concrete instances of our benchmarks

algorithm	fixed size instance obtained by assigning values to n, t, and f									
	$\mathcal{I}(3,2,1)$		$\mathcal{I}(4,3,2)$		$\mathcal{I}(4,3,3)$		$\mathcal{I}(5,4,2)$		$\mathcal{I}(5,4,3)$	
	states	time	states	time	states	time	states	time	states	time
FloodSet	6 937	4s	99 783	10s	1 220 227	1min 18s	1 024 866	1min 7s	34 724 276	1h 18min
FC	9 118	4s	138 160	11s	1 685 892	1min 45s	1 591 687	1min 39s	53 816 397	1h 43min
EDAC	26 962	5s	242 605	16s	5 703 025	5min 44s	1 940 929	2min 29s	124 183 639	4h 1min
ESC	10 543	4s	170 088	12s	2 954 288	2min 16s	1 577 742	1min 34s	71 913 792	1h 57min
NBAC	256	1s	16 120	7s	16 120	7s	286 670	46s	3 335 753	10min 33s
FloodMin	13 215	6s	287 001	1min 1s	3 311 397	14min 10s	5 297 856	23min 41s	out of memory in 3d 11h	

see the references given in Table 1 for details. As we focus on synchronous algorithms, we have a different set of benchmarks compared to the work in [17,38] that focuses on the partially synchronous algorithms from [12]. The only exception is that [17] considers *FloodMin* in the specific consensus setting ($k = 1$) which boils down to our *FC* consensus benchmark. They check 5 user-provided verification conditions, such as invariants or ranking functions, in less than a second. In our model-checking approach, the user does not have to provide an invariant, thus we have a higher degree of automation.

Table 1 summarizes the experiments for parameterized model checking. In our experiments we assume that the fixed m processes are correct, which implies $f \leq t \leq n - m$. To capture the corner cases $n - m < t < n$ required by the resilience condition $f \leq t < n$, we also do experiments with $m' < m$ concrete processes. In Table 1 we distinguish the cases when at most $m' < m$ are correct (right), from the one where this is not the case (left). We see that most of the verification time is spent on the case of at least m correct processes.

For comparison, Table 2 summarizes the experiments for small instances of up to $n = 5$ processes, where t is set to $n - 1$. We observe that parameterized verification outperforms model checking of fixed size systems already in the case of $n = 5$, $t = 4$, and $f = 3$. In the case of $n = 5$, $t = 4$, and $f = 4$, we were only able to verify the simplest benchmark, *NBAC*. For the remaining ones we reached the limitations of the model checker, as TLC was not able to enumerate all possible successor states due to the immense branching.

By far, *FloodMin* is the most challenging benchmark: its specifications are more complicated, and we therefore have to fix 3 processes (in contrast to 2 in the other benchmarks). In the concrete instance $\mathcal{I}(5, 4, 3)$, i.e., where $n = 5$,

$t = 4$, and $f = 3$, the model checker terminated after three days with an out of memory error.

6 Discussion

While synchronous distributed algorithms are considered "simpler" to design than asynchronous ones, encoding and model checking synchronous algorithms is a challenge: All processes take steps simultaneously, and each process can transfer into several successor states depending on the received messages, which are subject to non-determinism by the environment. We noticed in our experiments that synchronously selecting a successor state for each process combined with the non-determinism results in a huge branching factor. In conjunction with the additional non-determinism introduced through abstraction, this poses serious challenges to the explicit state model checker TLC. In future work, we will consider other model checking back-ends, and different encodings. Our predicate abstraction currently requires some domain knowledge to capture the interplay of the number of faults and round numbers. As future work we consider automatic generation of this abstraction by means of static analysis on the environment. All other abstractions can be done automatically. Finally, more complex resilience conditions that appear in the literature, such as $n > 2t$, would require a finer abstraction than the one we present here, a topic that we reserve for future work.

Parameterized model checking is undecidable in general [3, 4, 6, 19, 49]. Still, there are techniques for specific classes of systems. A popular technique is abstraction. Different domain-specific abstractions have been used for mutual exclusion [15, 16, 46], cache coherence [13, 32, 40, 43], dynamic scheduling [39], and recently to asynchronous FTDAs [2, 27, 28, 29]. Most of these parameterized model checking techniques consider asynchronous systems. The work most closely related to ours are the cutoff results of [38], as (i) it targets at completely automated verification, and (ii) while we have simulation to abstract systems, the authors of [38] prove simulation to small systems. To achieve this, the authors had to restrict the fragment to which the cutoff theorem applies: First, the cutoff only applies to consensus algorithms, that is, to three specific LTL specifications. As noted in [38], generalizing this to other specifications, e.g., k-set agreement, non-blocking atomic commit, or even a more complete logic fragment would require more theoretical work. Our case studies discussed in Section 5 include other algorithms than just consensus. Second, the guarded command language introduced in [38] can express only threshold guards containing predicates on the number of messages received by a process in the current round. However, there are several round-based distributed algorithms, in particular synchronous ones, that contain other guards; for instance, *termination guards* that check whether a given round number is reached, or guards that check whether messages from the same set of processes are received in two consecutive rounds. Our guarded commands contain such guards. Still, we currently cannot express all distributed algorithms, and extending our verification methods to other syntactic constructs is future work.

References

[1] C. Aiswarya, Benedikt Bollig, and Paul Gastin. An automata-theoretic approach to the verification of distributed algorithms. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 340–353, 2015.

[2] Francesco Alberti, Silvio Ghilardi, Andrea Orsini, and Elena Pagani. Counter Abstractions in Model Checking of Distributed Broadcast Algorithms: Some Case Studies. In *Proceedings of the 31st Italian Conference on Computational Logic, Milano, Italy, June 20-22, 2016.*, pages 102–117, 2016.

[3] Benjamin Aminof, Tomer Kotek, Sasha Rubin, Francesco Spegni, and Helmut Veith. Parameterized Model Checking of Rendezvous Systems. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, pages 109–124, 2014.

[4] Krzysztof R. Apt and Dexter Kozen. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.

[5] Hagit Attiya and Jennifer Welch. *Distributed Computing.* Wiley, 2nd edition, 2004.

[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] Michael C. Browne, Edmund M. Clarke, and Orna Grumberg. Reasoning about Networks with Many Identical Finite State Processes. *Inf. Comput.*, 81(1):13–31, 1989.

[8] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[9] Armando Castañeda, Yoram Moses, Michel Raynal, and Matthieu Roy. Early Decision and Stopping in Synchronous Consensus: A Predicate-Based Guided Tour. In *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, pages 206–221, 2017.

[10] Mouna Chaouch-Saad, Bernadette Charron-Bost, and Stephan Merz. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. In *Reachability Problems, 3rd International Workshop, RP 2009, Palaiseau, France, September 23-25, 2009. Proceedings*, pages 93–106, 2009.

[11] Bernadette Charron-Bost and André Schiper. Uniform Consensus is Harder than Consensus. *J. Algorithms*, 51(1):15–37, 2004.

[12] Bernadette Charron-Bost and André Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Faults. *Distributed Computing*, 22(1):49–71, 2009.

[13] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A Simple Method for Parameterized Verification of Cache Coherence Protocols. In *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, pages 382–398. 2004.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[15] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment Abstraction for Parameterized Verification. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, pages 126–141, 2006.

[16] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 33–47, 2008.

[17] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 161–181, 2014.

[18] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 400–415, 2016.

[19] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about Rings. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 85–94, 1995.

[20] E. Allen Emerson and A. Prasad Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[21] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[22] Dana Fisman, Orna Kupferman, and Yoad Lustig. On Verifying Fault Tolerance of Distributed Protocols. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 315–331, 2008.

[23] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83, 1997.

[24] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 201–209, 2013.

[25] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms. In *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, pages 209–226, 2013.

[26] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188, 2007.

[27] Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. *Formal Methods in System Design*, 51(2):270–307, 2017.

[28] Igor V. Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant

Distributed Algorithms. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 719–734, 2017.

[29] Igor V. Konnov, Helmut Veith, and Josef Widder. On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. *Inf. Comput.*, 252:95–109, 2017.

[30] Hermann Kopetz and Günter Grünsteidl. TTP - A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, 1994.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Michael Dahlin, Allen Clement, and Edmund L. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009.

[32] Sava Krstić. Parametrized System Verification with Guard Strengthening and Parameter Abstraction. In *AVIS*, 2005.

[33] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[34] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[35] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified Causally Consistent Distributed Key-value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 357–370, 2016.

[36] Patrick Lincoln and John M. Rushby. A Formally Verified Algorithm for Interactive Consistency Under a Hybrid Fault Model. In *Digest of Papers: FTCS-23, The Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993*, pages 402–411, 1993.

[37] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[38] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff Bounds for Consensus Algorithms. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 217–237, 2017.

[39] Kenneth L. McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, pages 219–234, 1999.

[40] Kenneth L. McMillan. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking. In *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK, September 4-7, 2001, Proceedings*, pages 179–195, 2001.

[41] Andre Medeiros. ZooKeeper's atomic broadcast protocol: Theory and practice. Technical report, 2012.

[42] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372, 2013.

[43] John W. O'Leary, Murali Talupur, and Mark R. Tuttle. Protocol Verification Using Flows: An Industrial Experience. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 172–179, 2009.

[44] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.

[45] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making Fast Consensus Generally Faster. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pages 156–167, 2016.

[46] Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, pages 107–122, 2002.

[47] Michel Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[48] Wilfried Steiner, John M. Rushby, Maria Sorea, and Holger Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 189–198, 2004.

[49] Ichiro Suzuki. Proving Properties of a Ring of Finite-State Machines. *Inf. Process. Lett.*, 28(4):213–214, 1988.

[50] TLA+ Toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.

[51] Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

[52] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and Universal Quantifiers for Verifying Parameterized Systems. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 599–613, 2016.

[53] Josef Widder, Günther Gridling, Bettina Weiss, and Jean-Paul Blanquart. Synchronous Consensus with Mortal Byzantines. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25-28 June 2007, Edinburgh, UK, Proceedings*, pages 102–112, 2007.

[54] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for Change in a Formal Verification of the RAFT Consensus Protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 154–165, 2016.

[55] Apache ZooKeeper. Web page. `http://zookeeper.apache.org/`.