Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, Josef Widder
Vienna University of Technology (TU Wien)

***Abstract*—We introduce an automated parameterized verification method for fault-tolerant distributed algorithms (FTDA). FTDAs are parameterized by both the number of processes and the assumed maximum number of faults. At the center of our technique is a parametric interval abstraction (PIA) where the interval boundaries are arithmetic expressions over parameters. Using PIA for both data abstraction and a new form of counter abstraction, we reduce the parameterized problem to finite-state model checking. We demonstrate the practical feasibility of our method by verifying safety and liveness of several fault-tolerant broadcasting algorithms, and finding counter examples in the case where there are more faults than the FTDA was designed for.**

I. Introduction

Fault-tolerant distributed algorithms (FTDA) constitute a core topic of distributed algorithm theory, with a rich body of results [27], [2]. Yet, they have not been systematically studied from a model checking point of view. For FTDAs one typically considers systems of n processes out of which at most t may be faulty. In this paper we consider various faults such as crash faults, omissions, and Byzantine faults. As FTDAs are parameterized in n and t, we require parameterized verification to establish the correctness of an FTDA. The pragmatic approach to verify a system of *fixed* size is not practical, as only very small instances can be verified due to state space explosion [24], [36], [34]. While in classic parameterized model checking the number of processes n is the sole parameter, for FTDAs, t is also a parameter, and is essentially a fraction of n, expressed by a *resilience condition*, e.g., $n > 3t$. Thus, one has to reason about all runs with $n-f$ non-faulty and f faulty processes, where $f \leq t$ and $n > 3t$.

From an operational viewpoint, FTDAs typically consist of multiple processes that communicate by passing messages. As senders can be faulty, a receiver cannot wait for a message from a specific sender process. Thus, most FTDAs use counters to reason about the environment; e.g., if a process receives a certain message from more than t distinct senders, then one of the senders must be non-faulty. A large class of FTDAs expresses these counting arguments using *threshold guards:*

```
if received <m> from t+1 distinct processes
then action(m);
```

Threshold guards generalize existential and universal guards [16], i.e., rules that wait for messages from at least one or all processes, respectively. As can be seen from the above example, and as discussed in [24], existential and universal guards are not sufficient to capture advanced FTDAs: Threshold guards are a basic building block that has been used in various environments (various degrees of synchrony, fault assumptions, etc.) and FTDAs, such as consensus [15], software and hardware clock synchronization [32], [19], approximate agreement [14], and k-set agreement [13]. The ability to efficiently reason about these guards is thus a keystone for automated parameterized verification of such algorithms.

This paper considers parameterized verification of FTDAs with threshold guards and resilience conditions. We introduce a framework based on a new form of control flow automata that captures the semantics of threshold-guarded FTDAs, and propose a novel two-step abstraction technique. It is based on *parametric interval abstraction* (PIA), a generalization of interval abstraction where the interval borders are expressions over *parameters* rather than constants. Using the PIA domain, we obtain a finite-state model checking problem in two steps:

Step 1: PIA data abstraction. We evaluate the threshold guards over the parametric intervals. Thus, we abstract away unbounded variables and parameters from the process code. We obtain a parameterized system where the replicated processes are finite-state and independent of the parameters.

Step 2: PIA counter abstraction. We use a new form of counter abstraction where the process counters are abstracted to PIA. As Step 1 guarantees that we need only finitely many counters, PIA counter abstraction yields a finite-state system.

To evaluate the precision of our abstractions, we implemented our abstraction technique in a tool chain, and conducted experiments on several FTDAs. Our experiments showed the need for abstraction refinement to deal with spurious counterexamples [7] that are due to parameterized abstraction and fairness. This required novel refinement techniques, which we also discuss in this paper. In addition to refinement of PIA counter abstraction, which is automated in a loop using a model checker and an SMT solver, we are also exploiting simple user-provided invariant candidates (as in [28], [35]) to refine the abstraction.

We verify several FTDAs that have been derived from the well-known distributed broadcast algorithm by Srikanth and Toueg [32], [33], and a folklore reliable broadcasting algorithm [2, Sect. 8.2.5.1]. Each of these FTDAs tolerates different faults (e.g., crash, omission, Byzantine), and uses different threshold guards. To the best of our knowledge, we are the first to achieve parameterized automated verification of Byzantine FTDAs.

Supported by the Austrian National Research Network S11403 and S11405 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) grants PROSEED, ICT12-059, and VRG11-005. Details that had to be omitted from this paper can be found in [23].

ISBN 978-0-9835678-3-7/13.

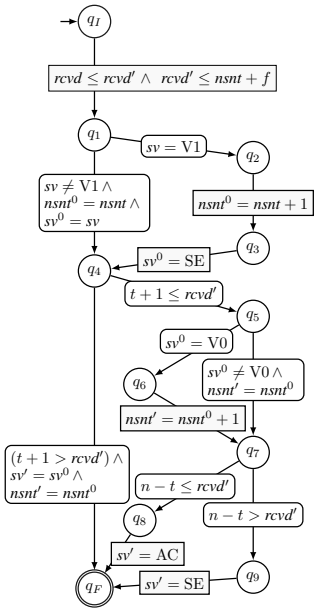


Fig. 1. CFA of our case study for Byzantine faults.

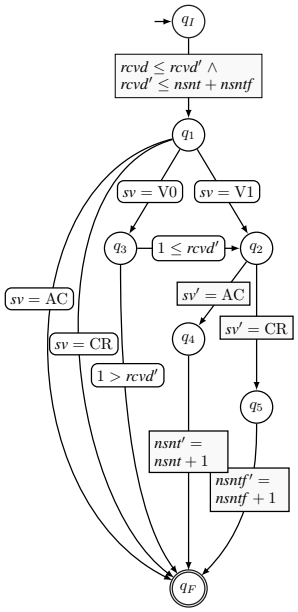


Fig. 2. CFA of FTDA from [18] (if x' is not assigned, then $x' = x$).

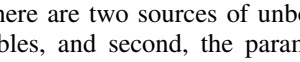
II. Our Approach at a Glance

To give an intuition of our method, we start with the control flow automaton (CFA) given in Figure 1 that formalizes our case study FTDA. The CFA uses the shared integer variable *nsnt* (capturing the number of messages sent by non-faulty processes), the local integer variable *rcvd* (storing the number of messages received by the process so far), and the local status variable *sv*, which ranges over a finite domain (capturing the local progress w.r.t. the FTDA). In [24] we show that this formalization captures the logic of our case study FTDA.

We use the CFA to represent one atomic *step* of the FTDA: Each edge is labeled with a guard. A path from q_I to q_F induces a conjunction of all the guards along it, and imposes constraints on the variables before the step (e.g., *sv*), after the step (*sv*′), and temporary variables (sv^0). If one fixes the variables before the step, different valuations (of the primed variables) that satisfy the constraints capture non-determinism.

A system consists of $n - f$ processes that concurrently execute the code corresponding to the CFA, and communicate via *nsnt*. Thus, there are two sources of unboundedness: first, the integer variables, and second, the parametric number of processes. We deal with these two issues in two steps.

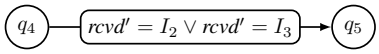
Step 1: PIA data abstraction. We observe that the CFA contains several transitions which are labeled with *threshold guards* that refer to (unbounded) variables and parameters. For instance, the CFA in Figure 1 contains the following transition, which is labeled with a threshold guard:



The CFA also contains a guard $n - t \leq rcvd'$. Actually, the correctness of the FTDA is based on the fact that the values of the thresholds, e.g., $t+1$ and $n-t$, are sufficiently far apart from each other under the resilience condition $n > 3t \wedge f \leq t$; in particular, $(n - t) - f \geq t + 1$. These properties are also used in the manual proofs [33]. We observe that such FTDAs are designed by carefully choosing the thresholds and the resilience condition. Consequently, our abstraction must be sufficiently precise to preserve the relationship between thresholds and the resilience condition.

The second important observation is that it is not necessary to keep track of the precise value of variables that are compared against thresholds, e.g., *rcvd*′. Rather, in our case study, it is sufficient to know whether *rcvd*′ lies in the interval $[0, t + 1[$, or $[t + 1, n - t[$, or $[n - t, \infty[$, in order to determine which of the threshold guards of the CFA are satisfied. Our *parametric interval abstraction* PIA exploits this idea. In addition, in Step 2 we will see that we also have to distinguish 0 from other values. Thus, PIA consists of mapping integers to a finite domain of four intervals $I_0 = [0, 1[$ and $I_1 = [1, t + 1[$ and $I_2 = [t + 1, n - t[$ and $I_3 = [n - t; \infty[$.

Then, we replace the guards that refer to unbounded variables and parameters by their existential abstraction. For instance, the above transition with the guard "$t + 1 \leq rcvd'$" means that *rcvd*′ lies in the intervals $[t+1, n-t[$ or $[n-t, \infty[$. As these correspond to the abstract intervals I_2 and I_3, respectively, we can replace the guard by:



The abstraction of the guard "$nsnt^0 = nsnt + 1$" can be expressed similarly, as later discussed in Figure 4. The expression "$rcvd' \leq nsnt + f$", which is also used in a guard, is more complicated as it involves two variables and a parameter. Still, the basic abstraction idea is the same. The corresponding abstract expression has the form $(rcvd' = I_0 \wedge nsnt = I_0) \vee (rcvd' = I_0 \wedge nsnt = I_1) \vee \cdots \vee (rcvd' = I_3 \wedge nsnt = I_3)$.

These abstract guards are Boolean expressions over equalities between variables and abstract values. Therefore, it is sufficient to interpret the variables *nsnt* and *rcvd* over the finite domain. Hence, all variables range over finite domains, and we arrive at finite state processes in this way. Our system, however, is still parameterized, namely, in the number of processes.

Step 2: PIA counter abstraction. We reduce this system to a finite state system using the following two ideas. First, we change to a counter-based representation, i.e., the global state is represented by the (abstract) shared variable *nsnt*, and by one counter for each of the local states. A counter stores how many processes are in the corresponding local state. Second, as processes interact only via the *nsnt* variable, precisely counting processes in certain states may not be necessary; as *nsnt* already ranges over the abstract domain, it is natural to count processes in terms of the same abstract domain.

The local state of a process is determined by the values of *sv* and *rcvd*. Thus, we denote by $\kappa[x, y] = I$ that the number of processes with $sv = x$ and $rcvd = y$ lies in the abstract interval I. Then, in Figure 3, the state s_0 represents the initial states with $t + 1$ to $n - t - 1$ processes having $sv = \mathtt{V0}$ and 1 to t processes having $sv = \mathtt{V1}$. (We omit local states that have the counter value I_0 to facilitate reading.)

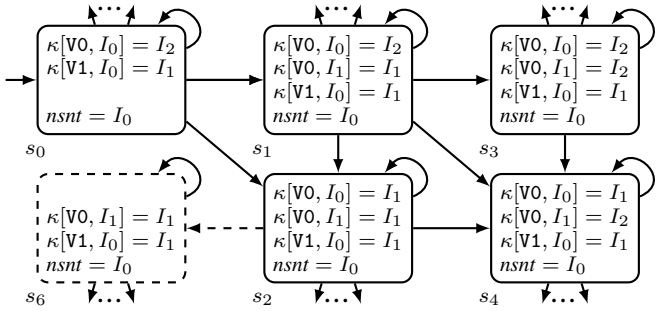


Fig. 3. A small part of the transition system obtained by counter abstraction. As shown by our experimental data in Table I of Section VII, the reachable state space is substantially larger.

Figure 3 gives a small part of the transition system obtained from the counter abstraction starting from initial state s_0. Each transition corresponds to one process taking a step in the concrete system. For instance, in the transition (s_0, s_2) a process with local state $[\texttt{V0}, I_0]$ changes its state to $[\texttt{V0}, I_1]$. Therefore, the counter $\kappa[\texttt{V0}, I_0]$ is decremented and the counter $\kappa[\texttt{V0}, I_1]$ is incremented. However, as we interpret counters over the abstract domain, the operations of incrementing and decrementing a counter are actually non-deterministic. Consequently, the transition (s_0, s_1) captures the same concrete local step as (s_0, s_2). In (s_0, s_1), the non-deterministic decrement of the abstract counter $\kappa[\texttt{V0}, I_0]$ did not change its value.

Typically, the specifications of FTDAs refer to global states where "there is a process in a given local state" or "all processes are in a given local state." To express this via counters, we have to check whether counter values are I_0.

Abstraction refinement. Our abstraction steps result in a system which is an over-approximation of all systems with fixed parameters. For instance, the non-determinism in the counters may "increase" or "decrease" the number of processes in a system, although in all concrete system the number of processes is constant: Consider the transition (s_2, s_6) in Figure 3, and let x, y, z be the non-negative integers that are in s_2 abstracted to $\kappa[\texttt{V0}, I_0]$, $\kappa[\texttt{V0}, I_1]$, and $\kappa[\texttt{V1}, I_0]$, respectively. Similarly y' and z' are abstracted to $\kappa[\texttt{V0}, I_1]$ and $\kappa[\texttt{V1}, I_0]$ in s_6. If the following inequalities do not have a solution under the resilience condition $(n > 3t, t \geq f)$, then there is no concrete system with a transition between two states that are abstracted to s_2 and s_6, respectively.

$$\begin{aligned} 1 \leq x < t+1,\ 1 \leq y\ < t+1,\ 1 \leq z\ < t+1,\\ 1 \leq y' < t+1,\ 1 \leq z' < t+1,\\ x + y + z = y' + z' = n - f. \end{aligned}$$

We use an SMT solver for this, and examine each transition of a counterexample returned by a model checker. If a transition is spurious, then we remove it from the abstract system.

Related abstractions. Interval abstraction [10] is a natural solution to the problem of unboundedness of local variables. However, if we fixed the interval bounds to numeric values, then they would not be aligned to the thresholds, and the abstraction would not be sufficiently precise to do parametric verification. At the same time, we do not have to deal with symbolic ranges over *variables* in the sense of [30], because for FTDAs the interval bounds are *constant* in each run.

Further, we want to produce a single process skeleton that is independent of parameters and captures the behavior of *all* process instances. This can be done by using ideas from existential abstraction [9], [12], [25] and sound abstraction of fairness constraints [25]. We combine these two ideas to arrive at PIA data abstraction.

The PIA counter abstraction is similar to [29], in that counters range over an abstract domain, and increment and decrement is done using existential abstraction. The domain in [29] consists of three values representing 0, 1, or *more*. This domain is sufficient for mutual-exclusion-like problems: It allows to distinguish good from bad states, while it is not possible (and also not necessary) to distinguish two bad states: A bad state is one where at least two processes are in the critical section, which is precisely abstracted in the three-valued domain. However, two bad states where, e.g., 2 and 3 processes are in the critical section, respectively, cannot be distinguished. Verification of threshold-based FTDAs requires more involved counting; e.g., we have to capture whether at least $n - t$ processes or at most t processes incremented *nsnt*. Therefore, we use counters from the PIA domain.

III. System Model with Multiple Parameters

In this section we develop all notions that are required to precisely state the parameterized model checking problem for multiple parameters. As running example, we use the parameters mentioned above, namely, the number of processes n, the upper bound on the number of faults t, and the actual number of faults f. We start to define parameterized processes (that access shared variables) in a way that allows us to modularly compose them into a parameterized system instance.

We apply this modeling to verify FTDAs as follows: as input we take a process description that uses the parameters n and t in the code. From this we construct a system instance parameterized with n, t, and f, which then describes all runs of an algorithm in which exactly f faults occur. The verification problem for a distributed algorithm in the *concrete case* with fixed n and t is the composition of model checking problems that differ in the actual value of $f \leq t$. This modeling also allows us to set $f = t + 1$, which models runs in which more faults occur than expected, and search for counterexamples. For the *parameterized case*, we introduce a resilience condition on these parameters, and require to verify the algorithm for all values of parameters that satisfy the resilience condition.

We define the parameters, local variables of the processes, and shared variables referring to a single *domain* D that is totally ordered and has the operations of addition and subtraction. In this paper we assume that D is the set of nonnegative integers $\mathbb{N}_0$.

We start with some notation. Let Y be a finite set of variables ranging over D. We denote by $D^{|Y|}$, the set of all $|Y|$-tuples of variable values. Given $\mathbf{s} \in D^{|Y|}$, we use the expression $\mathbf{s}.y$, to refer to the value of a variable $y \in Y$ in

vector $\mathbf{s}$. For two vectors $\mathbf{s}$ and $\mathbf{s}'$, by $\mathbf{s} =_X \mathbf{s}'$ we denote the fact that for all $x \in X$, $\mathbf{s}.x = \mathbf{s}'.x$ holds.

Process. The set of variables V is $\{sv\} \cup \Lambda \cup \Gamma \cup \Pi$: The variable sv is the *status variable* that ranges over a finite set *SV* of *status values*. The finite set Λ contains variables that range over the domain D. The variable sv and the variables from Λ are *local variables*. The finite set Γ contains the *shared variables* that range over D. The finite set Π is a set of *parameter variables* that range over D, and the *resilience condition RC* is a predicate over $D^{|\Pi|}$. In our example, $\Pi = \{n, t, f\}$, and the resilience condition $RC(n,t,f)$ is $n > 3t \ \wedge \ f \leq t \ \wedge \ t > 0$. Then, we denote the set of *admissible parameters* by $\mathbf{P}_{RC} = \{\mathbf{p} \in D^{|\Pi|} \mid RC(\mathbf{p})\}$.

A process operates on states from the set $S = SV \times D^{|\Lambda|} \times D^{|\Gamma|} \times D^{|\Pi|}$. Each process starts its computation in an initial state from a set $S^0 \subseteq S$. A relation $R \subseteq S \times S$ defines *transitions* from one state to another, with the restriction that the values of parameters remain unchanged, i.e., for all $(\mathbf{s}, \mathbf{t}) \in R$, $\mathbf{s} =_\Pi \mathbf{t}$. Then, a *parameterized process skeleton* is a tuple $\mathsf{Sk} = (S, S^0, R)$.

We get a process instance by fixing the parameter values $\mathbf{p} \in D^{|\Pi|}$: one can restrict the set of process states to $S|_{\mathbf{p}} = \{\mathbf{s} \in S \mid \mathbf{s} =_\Pi \mathbf{p}\}$ as well as the set of transitions to $R|_{\mathbf{p}} = R \cap (S|_{\mathbf{p}} \times S|_{\mathbf{p}})$. Then, a *process instance* is a process skeleton $\mathsf{Sk}|_{\mathbf{p}} = (S|_{\mathbf{p}}, S^0|_{\mathbf{p}}, R|_{\mathbf{p}})$ where $\mathbf{p}$ is constant.

System Instance. For fixed admissible parameters $\mathbf{p}$, a distributed system is modeled as an asynchronous parallel composition of identical processes $\mathsf{Sk}|_{\mathbf{p}}$. The number of processes depends on the parameters. To formalize this, we define the size of a system (the number of processes) using a function $N\colon \mathbf{P}_{RC} \to \mathbb{N}_0$, for instance, when modeling only correct processes explicitly, we use $n - f$ for $N(n,t,f)$.

Given $\mathbf{p} \in \mathbf{P}_{RC}$, and a process skeleton $\mathsf{Sk} = (S, S^0, R)$, a system instance is defined as an asynchronous parallel composition of $N(\mathbf{p})$ process instances, indexed by $i \in \{1, \ldots, N(\mathbf{p})\}$, with standard interleaving semantics. Let AP be a set of atomic propositions. A *system instance* $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a Kripke structure $(S_I, S_I^0, R_I, \mathrm{AP}, \lambda_I)$ where:

- $S_I = \{(\sigma[1], \ldots, \sigma[N(\mathbf{p})]) \in (S|_{\mathbf{p}})^{N(\mathbf{p})} \mid \forall i, j \in \{1, \ldots, N(\mathbf{p})\}, \sigma[i] =_{\Gamma \cup \Pi} \sigma[j]\}$ is the set of *(global) states*. Informally, a global state σ is a Cartesian product of the state $\sigma[i]$ of each process i, with identical values of parameters and shared variables at each process.
- $S_I^0 = (S^0)^{N(\mathbf{p})} \cap S_I$ is the set of *initial (global) states*, where $(S^0)^{N(\mathbf{p})}$ is the Cartesian product of initial states of individual processes.
- A transition (σ, σ') from a global state $\sigma \in S_I$ to a global state $\sigma' \in S_I$ belongs to R_I iff there is an index i, $1 \leq i \leq N(\mathbf{p})$, such that:
  - (MOVE) The i-th process *moves*: $(\sigma[i], \sigma'[i]) \in R|_{\mathbf{p}}$.
  - (FRAME) The values of the local variables of the other processes are preserved: for every process index $j \neq i$, $1 \leq j \leq N(\mathbf{p})$, it holds that $\sigma[j] =_{\{sv\} \cup \Lambda} \sigma'[j]$.
- $\lambda_I : S_I \to 2^{\mathrm{AP}}$ is a state labeling function.

Remark 1: The set of global states S_I and the transition relation R_I are preserved under every transposition $i \leftrightarrow j$ of process indices i and j in $\{1, \ldots, N(\mathbf{p})\}$. That is, every system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *fully symmetric* by construction.

Atomic Propositions. We define the set of atomic propositions AP to be the disjoint union of AP_{SV} and AP_D: The set AP_{SV} contains propositions that capture comparison against a given status value $Z \in SV$, i.e., $[\forall i.\ sv_i = Z]$ and $[\exists i.\ sv_i = Z]$. Further, the set of atomic propositions AP_D captures comparison of variables x, y, and a linear combination c of parameters from Π; AP_D consists of propositions of the form $[\exists i.\ x_i + c < y_i]$ and $[\forall i.\ x_i + c \geq y_i]$.

The labeling function λ_I of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ maps a state σ to expressions p from AP as follows (the existential case is defined accordingly using disjunctions):

$$[\forall i.\ sv_i = Z] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].sv = Z)$$

$$[\forall i.\ x_i + c \geq y_i] \in \lambda_I(\sigma) \text{ iff } \bigwedge_{i=1}^{N(\mathbf{p})} (\sigma[i].x + c(\mathbf{p}) \geq \sigma[i].y)$$

Temporal Logic. We specify properties using temporal logic $\mathsf{LTL}_{\setminus \mathsf{X}}$ over AP_{SV}. We use the standard definitions of paths and $\mathsf{LTL}_{\setminus \mathsf{X}}$ semantics [6]. A formula of $\mathsf{LTL}_{\setminus \mathsf{X}}$ is defined inductively as: (i) a literal p or $\neg p$, where $p \in \mathrm{AP}_{SV}$, or (ii) $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, $\varphi\,\mathbf{U}\,\psi$, $\varphi \vee \psi$, and $\varphi \wedge \psi$, where φ and ψ are $\mathsf{LTL}_{\setminus \mathsf{X}}$ formulas.

Fairness. We are interested in verifying safety and liveness properties. The latter can be usually proven only in the presence of fairness constraints. As in [25], [29], we consider verification of safety and liveness in systems with *justice* fairness constraints. We define fair paths of a system instance $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ using a set of justice constraints $J \subseteq \mathrm{AP}_D$. A path π of a system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is *J-fair* iff for every $p \in J$ there are infinitely many states σ in π with $p \in \lambda_I(\sigma)$. By $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$ we denote that the formula φ holds on all J-fair paths of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$.

Definition 2: Given a system description containing

- a domain D,
- a parameterized process skeleton $\mathsf{Sk} = (S, S_0, R)$,
- a resilience condition *RC* (generating a set of admissible parameters $\mathbf{P}_{RC}$),
- a system size function N,
- justice requirements J,

and an $\mathsf{LTL}_{\setminus \mathsf{X}}$ formula φ, the *parameterized model checking problem* (PMCP) is to verify $\forall \mathbf{p} \in \mathbf{P}_{RC}.\ \mathsf{Inst}(\mathbf{p}, \mathsf{Sk}) \models_J \varphi$.

IV. Threshold-guarded FTDAs

In [24], we formalized threshold-guarded FTDAs in Promela. In order to introduce our abstraction technique, we propose a language-independent approach that focuses on the control flow and is based on control flow automata (CFA) [21].

A *guarded control flow automaton* (CFA) is an edge-labeled directed acyclic graph $\mathcal{A} = (Q, q_I, q_F, E)$ with a finite set Q of nodes called locations, an initial location $q_I \in Q$, and a final location $q_F \in Q$. A path from q_I to q_F is used to describe *one step* of a distributed algorithm. The edges have the form

$E \subseteq Q \times \texttt{guard} \times Q$, where guard is defined as an expression of one of the following forms where $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$:

- if $Z \in SV$, then $sv = Z$ and $sv \neq Z$ are *status guards*;
- if x is a variable in D and $\lhd \in \{\leq, >\}$, then

$$a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i \ \lhd \ x$$

  is a *threshold guard*;
- if $y, z_1, \ldots, z_k$ are variables in D for $k \geq 1$, and $\lhd \in \{=, \neq, <, \leq, >, \geq\}$, and $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, then

$$y \ \lhd \ z_1 + \cdots + z_k + \big(a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i\big)$$

  is a *comparison guard*;
- a conjunction $g_1 \wedge g_2$ of guards g_1 and g_2 is a guard.

Status guards are used to capture the basic control flow. Threshold guards capture the core primitive of the FTDAs we consider. Finally, comparison guards are used to model send and receive operations. Figure 1 shows an example CFA with $\Gamma = \{nsnt\}$, $\Lambda = \{rcvd\}$, and $\Pi = \{n, t, f\}$.

Obtaining a Skeleton from a CFA. One step of a process skeleton is defined by a path from q_I to q_F in a CFA. Given *SV*, Λ, Γ, Π, *RC*, and a CFA $\mathcal{A}$, we define the process skeleton $\mathsf{Sk}(\mathcal{A}) = (S, S^0, R)$ induced by $\mathcal{A}$ as follows: The set of variables used by the CFA is $W \supseteq \Pi \cup \Lambda \cup \Gamma \cup \{sv\} \cup \{x' \mid x \in \Lambda \cup \Gamma \cup \{sv\}\}$, which may contain also temporary variables. A variable x corresponds to the value before a step, x' to the value after the step, and $x^0, x^1, \ldots$ to intermediate values. A path p from q_I to q_F induces a conjuction of all the guards along it. We call a mapping v from W to the values from the respective domains a *valuation*. We write $v \models p$ to denote that the valuation v satisfies the guards of the path p. We define the mapping between a CFA $\mathcal{A}$ and the transition relation of a process skeleton $\mathsf{Sk}(\mathcal{A})$: If there is a path p and a valuation v with $v \models p$, then v defines a single transition (s, t) of a process skeleton $\mathsf{Sk}(\mathcal{A})$, if for each variable $x \in \Lambda \cup \Gamma \cup \{sv\}$ it holds that $s.x = v(x)$ and $t.x = v(x')$ and for each parameter variable $z \in \Pi$, $s.z = t.z = v(z)$. Finally, the initial states S^0 need to be specified. For the type of algorithms we consider in this paper, all variables of the skeleton that range over D are initialized to 0, and *sv* ranging over *SV* takes an initial value from a fixed subset of *SV*. (For other algorithms, or self-stabilizing systems, one would choose different initializations.)

Remark 3: It might seem restrictive that our guards do not contain, e.g., increment, assignments, non-deterministic choice from a range of values. However, all these statements can be translated in our form using the SSA transformation algorithm from [11]. For instance, Figure 1 has been obtained from the Promela case study in [24], which contains the mentioned statements. Figures 1 and 2 provide two of the algorithms we have used for our experiments in Section VII.

Definition 4 (PMCP for CFA): We define the Parameterized Model Checking Problem for CFA $\mathcal{A}$ by specializing Definition 2 to the parameterized process skeleton $\mathsf{Sk}(\mathcal{A})$.

The problem given in Definition 4 is undecidable even if the CFA contains only status variables [23].

V. Abstraction Scheme

The input to our abstraction method is the infinite parameterized family $\mathcal{F} = \{\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \mid \mathbf{p} \in \mathbf{P}_{RC}\}$ of Kripke structures specified via a CFA $\mathcal{A}$. The family $\mathcal{F}$ has two principal sources of unboundedness: unbounded variables in the process skeleton $\mathsf{Sk}(\mathcal{A})$, and the unbounded number of processes $N(\mathbf{p})$. We deal with these two aspects separately, using two abstraction steps, namely the *PIA data abstraction* and the *PIA counter abstraction*. In both abstraction steps we use the parametric interval abstraction PIA.

Given a CFA $\mathcal{A}$, let $\mathcal{G}_\mathcal{A}$ be the set of all linear combinations $a_0 + \sum_{1 \leq i \leq |\Pi|} a_i \cdot p_i$ in the left-hand sides of $\mathcal{A}$'s threshold guards. Every expression ε of $\mathcal{G}_\mathcal{A}$ defines a function $f_\varepsilon \colon \mathbf{P}_{RC} \to D$. Let $\mathcal{T} = \{0, 1\} \cup \{f_\varepsilon \mid \varepsilon \in \mathcal{G}_\mathcal{A}\}$ be a finite *threshold set*, and $\mu + 1$ its cardinality. For convenience, we name elements of $\mathcal{T}$ as $\theta_0, \theta_1, \ldots, \theta_\mu$ with θ_0 corresponding to the constant function 0, and θ_1 corresponding to the constant 1. E.g., the CFA in Fig. 1 has the threshold set $\{\theta_0, \theta_1, \theta_2, \theta_3\}$, where $\theta_2(n, t, f) = t + 1$ and $\theta_3(n, t, f) = n - t$. Then, we define the domain of parametric intervals as:

$$\widehat{D} = \{I_j \mid 0 \leq j \leq \mu\}$$

Our abstraction rests on an implicit property of many FTDAs, namely, that the resilience condition *RC* induces an order on the thresholds used in the algorithm (e.g., $t+1 < n-t$).

Definition 5: The finite set $\mathcal{T}$ is uniformly ordered if for all $\mathbf{p} \in \mathbf{P}_{RC}$, and all $\theta_j(\mathbf{p})$ and $\theta_k(\mathbf{p})$ in $\mathcal{T}$ with $0 \leq j < k \leq \mu$, it holds that $\theta_j(\mathbf{p}) < \theta_k(\mathbf{p})$.

Assuming such an order does not limit the application of our approach: In cases where only a partial order is induced by *RC*, one can simply enumerate all finitely many total orders. As parameters, and thus thresholds, are kept unchanged in a run, one can verify an algorithm for each threshold order separately, and then combine the results.

Definition 5 allows us to properly define the *parameterized abstraction function* $\alpha_\mathbf{p} \colon D \to \widehat{D}$ and the *parameterized concretization function* $\gamma_\mathbf{p} \colon \widehat{D} \to 2^D$.

$$\alpha_\mathbf{p}(x) = \begin{cases} I_j & \text{if } x \in [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[\text{ for some } 0 \leq j < \mu \\ I_\mu & \text{otherwise.} \end{cases}$$

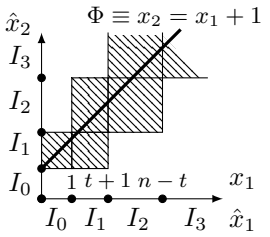
$$\gamma_\mathbf{p}(I_j) = \begin{cases} [\theta_j(\mathbf{p}), \theta_{j+1}(\mathbf{p})[& \text{if } j < \mu \\ [\theta_\mu(\mathbf{p}), \infty[& \text{otherwise.} \end{cases}$$

From $\theta_0(\mathbf{p}) = 0$ and $\theta_1(\mathbf{p}) = 1$, it immediately follows that for all $\mathbf{p} \in \mathbf{P}_{RC}$, we have $\alpha_\mathbf{p}(0) = I_0$, $\alpha_\mathbf{p}(1) = I_1$, and $\gamma_\mathbf{p}(I_0) = \{0\}$. Moreover, from the definitions of α, γ, and Definition 5 one immediately obtains:

Proposition 6: For all $\mathbf{p}$ in $\mathbf{P}_{RC}$, and for all a in D, it holds that $a \in \gamma_\mathbf{p}(\alpha_\mathbf{p}(a))$.

Definition 7: We define comparison between parametric intervals I_k and I_ℓ as $I_k \leq I_\ell$ iff $k \leq \ell$.

The PIA domain has similarities to predicate abstraction since the interval borders are naturally expressed as predicates, and computations over PIA are directly reduced to SMT solvers. However, notions such as the order of Definition 7 are not naturally expressed in terms of predicate abstraction.



$$\begin{aligned}\hat{\Phi} \;\equiv\; & \hat{x}_1 = I_0 \wedge \hat{x}_2 = I_1 \\ \vee\; & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_1 \\ \vee\; & \hat{x}_1 = I_1 \wedge \hat{x}_2 = I_2 \\ \vee\; & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_2 \\ \vee\; & \hat{x}_1 = I_2 \wedge \hat{x}_2 = I_3 \\ \vee\; & \hat{x}_1 = I_3 \wedge \hat{x}_2 = I_3\end{aligned}$$

Fig. 4. The shaded area approximates the line $x_2 = x_1 + 1$ along the boundaries of our parametric intervals. Each shaded rectangle corresponds to one conjunctive clause in the formula to the right. Thus, given $\Phi \equiv x_2 = x_1 + 1$, the shaded rectangles correspond to $||\Phi||_\exists$, from which we immediately construct the existential abstraction $\hat{\Phi}$.

A. PIA data abstraction

We now discuss an existential abstraction of a formula Φ that is either a threshold or a comparison guard (we consider other guards later). To this end, we introduce notation for sets of vectors satisfying Φ. According to Section IV, formula Φ has two kinds of free variables: parameter variables from Π and data variables from $\Lambda \cup \Gamma$. Let $\mathbf{x}^p$ be a vector of parameter variables $(x^p_1, \ldots, x^p_{|\Pi|})$ and $\mathbf{x}^v$ be a vector of variables $(x^v_1, \ldots, x^v_k)$ over D^k. Given a k-dimensional vector $\mathbf{d}$ of values from D, by

$$\mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi$$

we denote that Φ is satisfied on concrete values $x^v_1 = d_1, \ldots, x^v_k = d_k$ and parameter values $\mathbf{p}$. Then, we define:

$$\begin{aligned}||\Phi||_\exists = \{\hat{\mathbf{d}} \in \widehat{D}^k \mid \exists \mathbf{p} \in \mathbf{P}_{RC}\, \exists \mathbf{d} = (d_1, \ldots, d_k) \in D^k. \\ \hat{\mathbf{d}} = (\alpha_{\mathbf{p}}(d_1), \ldots, \alpha_{\mathbf{p}}(d_k)) \;\wedge\; \mathbf{x}^p = \mathbf{p}, \mathbf{x}^v = \mathbf{d} \models \Phi\}\end{aligned}$$

Hence, the set $||\Phi||_\exists$ contains all vectors of abstract values that correspond to some concrete values satisfying Φ. Parameters do not appear anymore due to existential quantification. A PIA *existential abstraction* of Φ is defined to be a formula $\hat{\Phi}$ over a vector of variables $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_k)$ over $\widehat{D}^k$ such that $\{\hat{\mathbf{d}} \in \widehat{D}^k \mid \hat{\mathbf{x}} = \hat{\mathbf{d}} \models \hat{\Phi}\} \supseteq ||\Phi||_\exists$.

Computing PIA abstractions. The central property of our abstract domain is that it allows to abstract comparisons against thresholds (i.e., threshold guards) in a precise way. That is, we can abstract formulas of the form $\theta_j(\mathbf{p}) \leq x_1$ by $I_j \leq \hat{x}_1$ and $\theta_j(\mathbf{p}) > x_1$ by $I_j > \hat{x}_1$. In fact, this abstraction is precise in the following sense.

Proposition 8: For all $\mathbf{p} \in \mathbf{P}_{RC}$ and all $a \in D$:
$\theta_j(\mathbf{p}) \leq a$ iff $I_j \leq \alpha_{\mathbf{p}}(a)$, and $\theta_j(\mathbf{p}) > a$ iff $I_j > \alpha_{\mathbf{p}}(a)$.

For comparison guards we use the general form, well-known from the literature, from the following proposition.

Proposition 9: If Φ is a formula over variables $x_1, \ldots, x_k$ over D, then $\bigvee_{(\hat{d}_1,\ldots,\hat{d}_k) \in ||\Phi||_\exists} \hat{x}_1 = \hat{d}_1 \wedge \cdots \wedge \hat{x}_k = \hat{d}_k$ is a PIA existential abstraction.

If the domain $\widehat{D}$ is small (as it is in our case), then one can enumerate all vectors of abstract values in $\widehat{D}^k$ and check which belong to our abstraction $||\Phi||_\exists$, using an SMT solver. As example consider the PIA domain $\{I_0, I_1, I_2, I_3\}$ for the CFA from Fig. 1. Fig. 4 illustrates $||\Phi||_\exists$ of $x_2 = x_1 + 1$ and the use of the formula from Proposition 9.

Transforming CFA. We now describe a general method to abstract `guard` formulas, and thus construct an abstract process skeleton. To this end, we denote by α_E a mapping from a concrete formula Φ to some existential abstraction of Φ (not necessarily constructed as above). By fixing α_E, we can define an abstraction of a `guard` of a CFA:

$$abs(g) = \begin{cases} \alpha_E(g) & \text{if } g \text{ is a threshold guard} \\ \alpha_E(g) & \text{if } g \text{ is a comparison guard} \\ g & \text{if } g \text{ is a status guard} \\ abs(g_1) \wedge abs(g_2) & \text{otherwise, i.e., } g \text{ is } g_1 \wedge g_2 \end{cases}$$

By abusing the notation, for a CFA $\mathcal{A}$ by $abs(\mathcal{A})$ we denote the CFA that is obtained from $\mathcal{A}$ by replacing every guard g with $abs(g)$. Note that $abs(\mathcal{A})$ contains only guards over sv and over abstract variables over $\widehat{D}$. For model checking, we have to reason about the Kripke structures that are built using the skeletons obtained from CFAs. We denote by $\mathsf{Sk}_{abs}(\mathcal{A})$, the process skeleton that is induced by CFA $abs(\mathcal{A})$, and by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$ an instance constructed from $\mathsf{Sk}_{abs}(\mathcal{A})$.

Soundness. It can be shown that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all CFA $\mathcal{A}$, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is simulated by $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$, with respect to AP_{SV}. Moreover, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A}))$ is a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}_{abs}(\mathcal{A}))$.

B. PIA counter abstraction

In this section, we present a counter abstraction inspired by [29], which maps a system instance composed of *identical finite state* process skeletons to a single finite state system. We use the PIA domain $\widehat{D}$ along with abstractions $\alpha_E(\{x' = x + 1\})$ and $\alpha_E(\{x' = x - 1\})$ for the counters.

Let us consider a process skeleton $\mathsf{Sk} = (S, S_0, R)$, where $S = SV \times \tilde{D}^{|\Lambda|} \times \tilde{D}^{|\Gamma|} \times \tilde{D}^{|\Pi|}$ that is defined using an arbitrary finite domain $\tilde{D}$. We present counter abstraction over the abstract domain $\widehat{D}$ in two stages, where the first stage is only a change in representation, but not an abstraction.

Stage 1: Vector Addition System with States (VASS). Let $L = \{\ell \in SV \times \tilde{D}^{|\Lambda|} \mid \exists s \in S.\ \ell =_{\{sv\} \cup \Lambda} s\}$ be the set of *local states* of a process skeleton. As the domain $\tilde{D}$ and the set of local variables Λ are finite, L is finite. We write the elements of L as $\ell_1, \ldots, \ell_{|L|}$. We define the counting function $K \colon S_I \times L \to D$ such that $K[\sigma, \ell]$ is the number of processes i whose local state is ℓ in global state $\sigma \in S_I$, i.e., $\sigma[i] =_{\{sv\} \cup \Lambda} \ell$. Thus, we represent the system state σ as a tuple $(g_1, \ldots, g_k, K[\sigma, \ell_1], \ldots, K[\sigma, \ell_{|L|}])$, i.e., by the shared global state and by the counters for the local states. If a process moves from local state ℓ_i to local state ℓ_j, the counters of ℓ_i and ℓ_j will decrement and increment, respectively.

Stage 2: Abstraction of VASS. We abstract the counters K of the VASS representation using the PIA domain to obtain a finite state Kripke structure $\mathsf{C}(\mathsf{Sk})$. To compute $\mathsf{C}(\mathsf{Sk}) = (S_\mathsf{C}, S^0_\mathsf{C}, R_\mathsf{C}, \mathrm{AP}, \lambda_\mathsf{C})$ we proceed as follows:

A state $w \in S_\mathsf{C}$ is given by values of shared variables from the set Γ, ranging over $\tilde{D}^{|\Gamma|}$, and by a vector

$(\kappa[\ell_1], \ldots, \kappa[\ell_{|L|}])$ over the abstract domain $\widehat{D}$ from Section V. More concisely, $S_{\mathsf{C}} = \widehat{D}^{|L|} \times \tilde{D}^{|\Gamma|}$.

Definition 10: The parameterized abstraction mapping $\bar{h}^{cnt}_{\mathbf{p}}$ maps a global state σ of the system $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ to a state w of the abstraction $\mathsf{C}(\mathsf{Sk})$ such that: For all $\ell \in L$ it holds that $w.\kappa[\ell] = \alpha_{\mathbf{p}}(K[\sigma, \ell])$, and $w =_\Gamma \sigma$.

From the definition, one can see how to construct the initial states. Informally, we require (1) that the initial shared states of $\mathsf{C}(\mathsf{Sk})$ correspond to initial shared states of Sk, (2) that there are actually $N(\mathbf{p})$ processes in the system, and (3) that initially all processes are in an initial state.

The intuition for the construction of the transition relation is as follows: Like in VASS, a step that brings a process from local state ℓ_i to ℓ_j can be modeled by decrementing the (non-zero) counter of ℓ_i and incrementing the counter of ℓ_j using the existential abstraction $\alpha_E(\{\kappa'[\ell_i] = \kappa[\ell_i] - 1\})$ and $\alpha_E(\{\kappa'[\ell_j] = \kappa[\ell_j] + 1\})$.

Soundness. We show that for all $\mathbf{p} \in \mathbf{P}_{RC}$, and for all finite state process skeletons Sk, $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is simulated by $\mathsf{C}(\mathsf{Sk})$, w.r.t. AP_{SV}. Further, the abstraction of a J-fair path of $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk})$ is a J-fair path of $\mathsf{C}(\mathsf{Sk})$.

Theorem 11 (Soundness of data & counter abstraction): For all CFA $\mathcal{A}$, and for all formulas φ from $\mathsf{LTL}_{\setminus \mathsf{X}}$ over AP_{SV} and justice constraints $J \subseteq \mathrm{AP}_D$: if $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A})) \models_J \varphi$, then for all $\mathbf{p} \in \mathbf{P}_{RC}$ it holds $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) \models_J \varphi$.

VI. Abstraction Refinement

The states of the abstract system are determined by variables over $\widehat{D}$. Proposition 8 shows that we precisely abstract the relevant properties of our variables, i.e., comparisons to thresholds. Hence, the classic CEGAR approach [7], which consists of refining the state space, does not appear suitable. However, the non-determinism due to our existential abstraction leads to *spurious transitions* that one can eliminate.

We encountered two sources of spurious transitions: As discussed in Section II, transitions can "lose processes," i.e., any concretization of the abstract number of processes is less than the number of processes we started with. This is not within the assumption of FTDAs, and thus spurious. Second, in our case study (cf. Figure 1) processes increase the global variable *nsnt* by one, when they transfer to a state where the value of the status variable is in $\{\mathrm{SE}, \mathrm{AC}\}$. Hence, in concrete system instances, *nsnt* should always be equal to the number of processes whose status variable value is in $\{\mathrm{SE}, \mathrm{AC}\}$, while due to phenomena similar to those discussed above, we can "lose messages" in the abstract system.

The experiments show that in our case studies neither losing processes nor losing messages has influence on the verification of safety specifications. However, these behaviors pose challenges for liveness as they lead to spurious counterexamples: Message passing FTDAs typically require that a process receives messages from (nearly) all correct processes, which is problematic if processes (i.e., potential senders) or messages are lost.

Besides, in Figure 1 we model message receptions by an update of the variable *rcvd*, more precisely, $\mathit{rcvd} \leq \mathit{rcvd}' \wedge \mathit{rcvd}' \leq \mathit{nsnt} + f$. One may observe that this alone does not require that the value of *rcvd* actually increases. Hence, we add justice requirements, e.g., $J = \{[\forall i.\ \mathit{rcvd}_i \geq \mathit{nsnt}]\}$ in our case study. As observed by [29], counter abstraction may lead to justice suppression. Given a counter-example in the form of a lasso, we detect whether its loop contains only unjust states. If this is the case, similar to an idea from [29], we refine $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by adding a justice requirement, which is consistent with existing requirements in all concrete instances.

Below, we give a general framework for a sound refinement of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$. (In [23], we provide a more detailed discussion on the practical refinement techniques that we use in our experiments.) To simplify presentation, we define a *monster system* as a (possibly infinite) Kripke structure $\mathsf{Sys}_\omega = (S_\omega, S^0_\omega, R_\omega, \mathrm{AP}, \lambda_\omega)$, whose state space and transition relation are disjoint unions of state spaces and transition relations of system instances $\mathsf{Inst}(\mathbf{p}, \mathsf{Sk}(\mathcal{A})) = (S_{\mathbf{p}}, S^0_{\mathbf{p}}, R_{\mathbf{p}}, \mathrm{AP}, \lambda_{\mathbf{p}})$ over all admissible parameters:

$$S_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S_{\mathbf{p}}, \qquad S^0_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} S^0_{\mathbf{p}}, \qquad R_\omega = \bigcup_{\mathbf{p} \in \mathbf{P}_{RC}} R_{\mathbf{p}}$$

$$\lambda_\omega : S_\omega \to 2^{\mathrm{AP}} \text{ and } \forall \mathbf{p} \in \mathbf{P}_{RC}, \forall s \in S_{\mathbf{p}}.\ \lambda_\omega(s) = \lambda_{\mathbf{p}}(s)$$

Let $h\colon S_\omega \to S_{\mathsf{C}}$ be an abstraction mapping, e.g., a combination of the abstraction mappings from Section V.

Definition 12: A sequence $T = \{\sigma_i\}_{i \geq 1}$ is a *concretization* of path $\hat{T} = \{w_i\}_{i \geq 1}$ from $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ if and only if $\sigma_1 \in S^0_\omega$ and for all $i \geq 1$ it holds $h(\sigma_i) = w_i$.

Definition 13: A path $\hat{T}$ of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ is a *spurious path* iff every concretization T of $\hat{T}$ is not a path in Sys_ω.

A prerequisite to abstraction refinement is to check whether a counter-example provided by the model checker is spurious. While for finite state systems there are methods to detect whether a path is spurious [7], we are not aware of a method to detect whether a path $\hat{T}$ in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ corresponds to a path in the (concrete) infinite monster system Sys_ω. Therefore, we limit ourselves to detecting and refining uniformly spurious transitions and unjust states. We first consider spurious transitions.

Definition 14: An abstract transition $(w, w') \in R_{\mathsf{C}}$ is *uniformly spurious* iff there is no transition $(\sigma, \sigma') \in R_\omega$ with $w = h(\sigma)$ and $w' = h(\sigma')$.

The following theorem provides us with a general criterion that ensures that removing uniformly spurious transitions does not affect the property of transition preservation.

Theorem 15: Let $T \subseteq R_{\mathsf{C}}$ be a set of spurious transitions. Then for every transition $(\sigma, \sigma') \in R_\omega$ there is a transition $(h(\sigma), h(\sigma'))$ in $R_{\mathsf{C}} \setminus T$.

It follows that the system $(S_{\mathsf{C}}, S^0_{\mathsf{C}}, R_{\mathsf{C}} \setminus T, \mathrm{AP}, \lambda_{\mathsf{C}})$ still simulates Sys_ω. After considering spurious transitions, we now consider justice suppression.

Definition 16: An abstract state $w \in S_{\mathsf{C}}$ is *unjust under* $q \in \mathrm{AP}_D$ iff there is no concrete state $\sigma \in S_\omega$ with $w = h(\sigma)$ and $q \in \lambda_\omega(\sigma)$.

Consider infinite counterexamples of $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$, which have a form of lassos $w_1 \ldots w_k (w_{k+1} \ldots w_m)^\omega$. For such a counterexample $\hat{T}$ we denote the set of states in the lasso's loop by U. We then check, whether all states of U are unjust

under some justice constraint $q \in J$. If this is the case, then $\hat{T}$ is a spurious counterexample, because the justice constraint q is violated. Note that it is sound to only consider infinite paths, where states outside of U appear infinitely often; in fact, this is a justice requirement. To refine C's unjust behavior we add a corresponding justice requirement. Formally, we augment J (and AP_D) with a propositional symbol $[\mathit{off}\ U]$. Further, we augment the labelling function λ_{C} such that every $w \in S_{\mathsf{C}}$ is labelled with $[\mathit{off}\ U]$ if and only if $w \notin U$.

Theorem 17: Let $J \subseteq \mathrm{AP}_D$ be a set of justice requirements, $q \in J$, and $U \subseteq S_{\mathsf{C}}$ be a set of unjust states under q. Let $\pi = \{\sigma_i\}_{i\geq 1}$ be an arbitrary fair path of Sys_ω under J. The path $\hat{\pi} = \{\hat{h}(\sigma_i)\}_{i\geq 1}$ is fair in $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ under $J \cup \{[\mathit{off}\ U]\}$.

From this we derive that loops containing only unjust states can be eliminated, and thus $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ be refined.

We encountered cases where several non-uniform spurious transitions resulted in a uniformly spurious path (i.e., a counterexample). We refine such spurious behavior by invariants. These invariants are provided by the user as invariant candidates, and are then automatically checked to actually be invariants using an SMT solver. In our example the invariant is simply "the number of processes that sent a message equals the number of sent messages."

VII. Experimental Evaluation

To show feasibility of our abstractions, we have implemented the PIA abstractions and the refinement loop in OCaml as a prototype tool ByMC. We evaluated it on different broadcasting algorithms. They deal with different fault models and resilience conditions; the algorithms are: (Byz), which is the algorithm from Figure 1, for t Byzantine faults if $n > 3t$, (symm) for t symmetric (identical Byzantine [2]) faults if $n > 2t$, (omit) for t send omission faults if $n > 2t$, and (clean) for t clean crash faults [37] if $n > t$. In addition, we verified the RBC algorithm — formalized also in [18] — whose CFA is given in Figure 2. In this paper we verify the following safety and liveness specifications:

$$[\forall i.\ sv_i \neq \mathrm{V1}] \rightarrow \mathbf{G}\ [\forall j.\ sv_j \neq \mathrm{AC}] \quad \text{(U)}$$

$$[\forall i.\ sv_i = \mathrm{V1}] \rightarrow \mathbf{F}\ [\exists j.\ sv_j = \mathrm{AC}] \quad \text{(C)}$$

$$\mathbf{G}\,(\neg\,[\exists i.\ sv_i = \mathrm{AC}]) \vee\ \mathbf{F}\ [\forall j.\ sv_j = \mathrm{AC}] \quad \text{(R)}$$

In addition, in [18] a specification A for RBC was introduced, which we verify for RBC. In contrast to [18], we actually implemented our verification method and give experimental data.

From the literature we know that we cannot expect to verify these FTDAs without restricting the environment, e.g., with communication fairness, namely, every message sent is eventually received. To capture this, we use justice requirements, e.g., $J = \{[\forall i.\ \mathit{rcvd}_i \geq \mathit{nsnt}]\}$ in the Byzantine case.

We extended Promela [22] with constructs to express Π, AP, *RC*, and N [24]. ByMC receives a description of a CFA $\mathcal{A}$ in this extended Promela, and then syntactically extracts the thresholds. The tool chain uses the Yices SMT solver for existential abstraction, and generates the counter abstraction $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ in standard Promela, such that we can use Spin to do finite state model checking. Finally, ByMC also implements the refinements introduced in Section VI and refines the Promela code for $\mathsf{C}(\mathsf{Sk}_{abs}(\mathcal{A}))$ by introducing predicates capturing spurious transitions and unjust states.

TABLE I. Summary of experiments

$M \models \varphi$?	RC	Spin Time	Spin Memory	Spin States	Spin Depth	$\lvert\widehat{D}\rvert$	#R	Total Time
Byz $\models U$	(A)	2.3 s	82 MB	483k	9154	4	0	4 s
Byz $\models C$	(A)	3.5 s	104 MB	970k	20626	4	10	32 s
Byz $\models R$	(A)	6.3 s	107 MB	1327k	20844	4	10	24 s
Sym $\models U$	(A)	0.1 s	67 MB	19k	897	3	0	1 s
Sym $\models C$	(A)	0.1 s	67 MB	19k	1113	3	2	3 s
Sym $\models R$	(A)	0.3 s	69 MB	87k	2047	3	12	16 s
Omt $\models U$	(A)	0.1 s	66 MB	4k	487	3	0	1 s
Omt $\models C$	(A)	0.1 s	66 MB	7k	747	3	5	6 s
Omt $\models R$	(A)	0.1 s	66 MB	8k	704	3	5	10 s
Cln $\models U$	(A)	0.3 s	67 MB	30k	1371	3	0	2 s
Cln $\models C$	(A)	0.4 s	67 MB	35k	1707	3	4	8 s
Cln $\models R$	(A)	1.1 s	67 MB	51k	2162	3	13	31 s
RBC $\models U$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
RBC $\models A$	—	0.1 s	66 MB	1.7k	333	2	0	1 s
RBC $\models R$	—	0.1 s	66 MB	1.2k	259	2	0	1 s
RBC $\not\models C$	—	0.1 s	66 MB	0.8k	232	2	0	1 s
Byz $\not\models U$	(B)	5.2 s	101 MB	1093k	17685	4	9	56 s
Byz $\not\models C$	(B)	3.7 s	102 MB	980k	19772	4	11	52 s
Byz $\not\models R$	(B)	0.4 s	67 MB	59k	6194	4	10	17 s
Byz $\models U$	(C)	3.4 s	87 MB	655k	10385	4	0	5 s
Byz $\models C$	(C)	3.9 s	101 MB	963k	20651	4	9	32 s
Byz $\not\models R$	(C)	2.1 s	91 MB	797k	14172	4	30	78 s
Sym $\not\models U$	(B)	0.1 s	67 MB	19k	947	3	0	2 s
Sym $\not\models C$	(B)	0.1 s	67 MB	18k	1175	3	2	4 s
Sym $\models R$	(B)	0.2 s	67 MB	42k	1681	3	8	12 s
Omt $\models U$	(D)	0.1 s	66 MB	5k	487	3	0	1 s
Omt $\not\models C$	(D)	0.1 s	66 MB	5k	487	3	0	2 s
Omt $\not\models R$	(D)	0.1 s	66 MB	0.1k	401	3	0	2 s

Table I summarizes our experiments run on 3.3GHz Intel® Core™ 4GB. In the cases (A) we used resilience conditions as provided by the literature, and verified the specification. The model RBC is the reliable broadcast algorithm also considered in [18] under the resilience condition $n \geq t \geq f$. In the bottom part of Table I we used different resilience conditions under which we expected the algorithms to fail. The cases (B) capture the case where more faults occur than expected by the algorithm designer ($f \leq t+1$ instead of $f \leq t$), while the cases (C) and (D) capture the cases where the algorithms were designed by assuming wrong resilience conditions (e.g., $n \geq 3t$ instead of $n > 3t$ in the Byzantine case). We omit (clean) as the only sensible case $n = t = f$ (all processes are faulty) results into a trivial abstract domain of one interval $[0, \infty)$. The column "#R" gives the numbers of refinement steps. In the cases where it is greater than zero, refinement was necessary, and "Spin Time" refers to the Spin running time after the last refinement step. Finally, column $|\widehat{D}|$ indicates the size of the abstract domain.

VIII. Related work

Traditionally, correctness of FTDAs is shown by handwritten proofs [27], [2], and, in some cases, by proof assistants [26], [31], [5]. Completely automated model checking or synthesis are usually not parameterized [24], [36], [34], [3]. Our work stands in the tradition of parameterized model checking for protocols [4], [20], [17], [29], [8], e.g., mutual exclusion and cache coherence. In particular, counter abstraction and justice preservation by Pnueli et al. [29] are keystones of our work.

To the best of our knowledge there are two papers on parameterized model checking of FTDAs [18], [1]. The authors of [18] use regular model checking to make interesting theoretical progress, but did not do any implementation. Their models are limited to processes whose local state space and transition relation are *finite and independent of parameters*. This was sufficient to formalize a reliable broadcast algorithm that tolerates crash faults, and where every process stores whether it has received at least one message. Such models are *not sufficient* to capture FTDAs that contain threshold guards as in our case. Moreover, the presence of a resilience condition such as $n > 3t$ would require them to intersect the regular languages, which describe sets of states, with context-free languages that enforce the resilience condition.

In [1], the safety of synchronous broadcasting algorithms that tolerate crash or send omission faults has been verified. These FTDAs have similar restrictions as the ones considered in [18]: Alberti et al. [1] mention that they did not consider FTDAs that feature "substantial arithmetic reasoning", i.e., threshold guards and resilience conditions, as they would require novel suitable techniques. Our abstractions address this arithmetic reasoning.

To the best of our knowledge, the current paper is thus the first in which *safety* and *liveness* of an FTDA that tolerates Byzantine faults has been *automatically* verified for *all* system sizes and *all* admissible numbers of faulty processes.

IX. Conclusions

We extended the standard setting of parameterized model checking to processes that use threshold guards, and are parameterized with a resilience condition. As a case study we have chosen the core of several broadcasting algorithms under different failure models, including one [33] that tolerates Byzantine faults. These algorithms are widely applied in the literature: typically, multiple (possibly an unbounded number of) instances are used in combination. As future work, we plan to use compositional model checking techniques [28] for parameterized verification of such algorithms. Another open issue is to capture additional fault assumptions such as communication faults [5], [37].

References

[1] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise, and G. P. Rossi, "Universal guards, relativization of quantifiers, and failure models in model checking modulo theories," *JSAT*, vol. 8, no. 1/2, pp. 29–61, 2012.

[2] H. Attiya and J. Welch, *Distributed Computing*, 2nd ed. Wiley, 2004.

[3] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant distributed programs," *Distributed Computing*, vol. 25, no. 1, pp. 83–108, 2012.

[4] M. C. Browne, E. M. Clarke, and O. Grumberg, "Reasoning about networks with many identical finite state processes," *Inf. Comput.*, vol. 81, pp. 13–31, 1989.

[5] B. Charron-Bost and S. Merz, "Formal verification of a consensus algorithm in the heard-of model," *IJSI*, vol. 3, no. 2–3, pp. 273–303, 2009.

[6] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[8] E. Clarke, M. Talupur, and H. Veith, "Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems," in *TACAS'08/ETAPS'08*. Springer, 2008, pp. 33–47.

[9] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM TOPLAS*, vol. 16, no. 5, pp. 1512–1542, 1994.

[10] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, vol. 13, no. 4, pp. 451–490, 1991.

[12] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM TOPLAS*, vol. 19, no. 2, pp. 253–291, 1997.

[13] R. De Prisco, D. Malkhi, and M. K. Reiter, "On k-set consensus problems in asynchronous systems," *TPDS*, vol. 12, no. 1, pp. 7–21, 2001.

[14] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *J. ACM*, vol. 33, no. 3, pp. 499–516, 1986.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J.ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[16] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *CADE*, ser. LNCS, 2000, vol. 1831, pp. 236–254.

[17] ——, "Exact and efficient verification of parameterized cache coherence protocols," in *CHARME*, ser. LNCS, vol. 2860, 2003, pp. 247–262.

[18] D. Fisman, O. Kupferman, and Y. Lustig, "On verifying fault tolerance of distributed protocols," in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 315–331.

[19] M. Függer and U. Schmid, "Reconciling fault-tolerant distributed computing and systems-on-chip," *Dist. Comp.*, vol. 24, no. 6, pp. 323–355, 2012.

[20] S. M. German and A. P. Sistla, "Reasoning about systems with many processes," *J. ACM*, vol. 39, pp. 675–735, 1992.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[22] G. Holzmann, *The SPIN Model Checker*. Addison-Wesley, 2003.

[23] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Counter attack on Byzantine generals: Parameterized model checking of fault-tolerant distributed algorithms," *arXiv CoRR*, vol. abs/1210.3846, 2012.

[24] ——, "Towards modeling and model checking fault-tolerant distributed algorithms," in *SPIN*, ser. LNCS, vol. 7976, 2013, pp. 209–226.

[25] Y. Kesten and A. Pnueli, "Control and data abstraction: the cornerstones of practical formal verification," *STTT*, vol. 2, pp. 328–342, 2000.

[26] P. Lincoln and J. Rushby, "A formally verified algorithm for interactive consistency under a hybrid fault model," in *FTCS*, 1993, pp. 402–411.

[27] N. Lynch, *Distributed Algorithms*. Morgan Kaufman, 1996.

[28] K. L. McMillan, "Parameterized verification of the flash cache coherence protocol by compositional model checking," in *CHARME*, ser. LNCS, vol. 2144, 2001, pp. 179–195.

[29] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0,1,∞)- counter abstraction," in *CAV*, ser. LNCS. Springer, 2002, vol. 2404, pp. 93–111.

[30] S. Sankaranarayanan, F. Ivancic, and A. Gupta, "Program analysis using symbolic ranges," in *SAS*, ser. LNCS, vol. 4634, 2007, pp. 366–383.

[31] U. Schmid, B. Weiss, and J. Rushby, "Formally verified Byzantine agreement in presence of link faults," in *ICDCS*, 2002, pp. 608–616.

[32] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, 1987.

[33] T. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Dist. Comp.*, vol. 2, pp. 80–94, 1987.

[34] W. Steiner, J. M. Rushby, M. Sorea, and H. Pfeifer, "Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation," in *DSN*, 2004, pp. 189–198.

[35] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *FMCAD*, 2008, pp. 1–8.

[36] T. Tsuchiya and A. Schiper, "Verification of consensus algorithms using satisfiability solving," *Dist. Comp.*, vol. 23, no. 5–6, pp. 341–358, 2011.

[37] J. Widder and U. Schmid, "Booting clock synchronization in partially synchronous systems with hybrid process and link failures," *Dist. Comp.*, vol. 20, no. 2, pp. 115–140, 2007.