Parameterized Distributed Synthesis of Fault-Tolerance Using Counter Abstraction

Hadi Moloodi
School of Electrical and Computer Engineering
College of Engineering, University of Tehran
hadi.moloodi@ut.ac.ir

Fathiyeh Faghih
School of Electrical and Computer Engineering
College of Engineering, University of Tehran
f.faghih@ut.ac.ir

Borzoo Bonakdarpour
Department of Computer Science and Engineering
Michigan State University
borzoo@msu.edu

***Abstract*—In this paper, we propose an automated technique for synthesizing *fault-tolerant distributed* protocols from their fault-intolerant version, where the number of processes is parameterized. A fault-tolerant protocol is one that ensures constant satisfaction of safety and liveness specifications even in the presence of faults. Although the parametrized synthesis problem is undecidable in general, we could propose a sound algorithm for this challenging problem. Our synthesis algorithm utilizes *counter abstraction* to construct a finite representation of the state space. Then, it performs fixpoint calculations to compute and exclude states that violate the safety/liveness specifications in the presence of faults. We demonstrate the effectiveness of our algorithm by synthesizing fault-tolerant distributed protocols for well-known problems, such as reliable broadcast and a simplified version of Byzantine agreement in a matter of seconds.**

I. Introduction

Distributed algorithms are often subject to different types of *faults*, and hence, providing fault-tolerance in distributed settings has been an active area of research for decades. The existence of many impossibility results (e.g., [1]) in very simple fault-prone settings is a witness to the challenges that the occurrence of faults impose on designing distributed algorithms. Verification of these algorithms is an equally tedious task as the explosion of different interleavings makes the problem intractable.

A prominent approach to circumvent the design-verification loop is to automatically *synthesize* fault-tolerant algorithms. Program synthesis is in particular appealing as it guarantees correctness by construction for the resulting protocol, and hence, no verification step is required to check their soundness. While there is a strong line of work on parameterized verification of distributed fault-tolerant algorithms [2], [3], [4], [5], [6], [7], there is relatively little work on parameterized synthesis. Existing approaches for synthesizing distributed fault-tolerant algorithms include techniques that synthesize a fault-tolerant protocol from their specification [8], [9], [10], [11]. Others attempt to repair a distributed protocol with respect to a given set of faults into a fault-tolerant algorithm [12], [13], [14], [15]. However, a major shortcoming of these approaches is that they are topology-specific, and only work for a system with a predefined *fixed* number of processes. Typically, the runtime of these algorithms grows exponentially by the number of processes. A natural solution to this problem is to develop parameterized solutions that can synthesize a fault-tolerant distributed algorithm that works for topologies with *any* number of processes.

Parameterized synthesis and verification are known to be undecidable problems in general [16], [17]. Few approaches have been proposed in the literature to overcome this difficulty. One general approach is to come up with cutoffs [18], [19], [20], such that a distributed algorithm that works for topologies up to the cutoff size is guaranteed to be parameterized. In [21], the authors present an algorithm for parameterized synthesis of distributed fault-tolerant systems based on parameterized verification and CEGIS loop. They model the problem as a threshold automaton where the transition relation is given with a number of holes in its guards. The synthesis solution concertizes parametric guards.

In this paper, we take a different approach for synthesizing parameterized fault-tolerant distributed algorithms. Our technique takes as input a fault-*intolerant* process *template*, and automatically *adds* fault-tolerance to it. By "addition", we mean that the synthesis algorithm preserves all the good behaviors of the input intolerant program and (1) restricts the behvaiors that may violate the specification in the presence of faults, and (2) augments the program with *recovery* computations that ensure eventual satisfaction of all liveness properties after the occurrence of faults. Any system instantiated from the resulting template with any number of processes is guaranteed to be fault-tolerant.

Our algorithm works as follows. Inspired by parameterized model checking of fault-tolerant distributed protocols [2], we first employ *counter abstraction* [22] to build a finite abstraction of the system's state space. This is not a trivial application of counter abstraction, as synthesizing a transition relation for process templates involves reasoning about read/write restrictions of processes in abstract states. Thus, we adapt counter abstraction so that it is aware of partial observability of processes. Next, the algorithm performs two fixpoint computations to guarantee that (1) states from where the safety specification may be violated are not reachable in

We acknowledge Igor Konnov and Josef Wieder for discussions and sharing insights.

the presence of faults, and (2) when a computation diverges from the set of good behaviors due to the occurrence of faults, first they do not deadlock and, furthermore, they can recover back to a good behavior. Recovery can be achieved by adding transitions that do not compromise the existing good behaviors while ensuring eventual reachability of a good behavior.

Our technique is fully implemented and we report experimental results. Our case studies include the (1) Byzantine agreement problem with and without crash faults, (2) reliable broadcast, and (3) uniform broadcast. All of our experiments concluded in less than 20 seconds. The paper is organized as follows. Our computation model is described in Section II before presenting the formal statement of the problem in Section III. We introduce our synthesis algorithm in Section IV, evaluate it in Section V, and conclude in Section VI.

II. Model of Computation

A. Parameterized Distributed Programs

Before defining parameterized distributed programs, we need to introduce a process template. A *process template* is a tuple $\langle V, G\rangle$, where V is a finite set of *control variables* and G is a finite set of *guarded commands* that specify the behavior of a process instantiated from the process template. The set of *control* variables V has two disjoint subsets: (1) a finite set V^ℓ of *local* variables, and (2) a finite set V^s of *shared* variables. Each variable $v \in V$ has a finite domain D_v, and a set of initial values $Init_v \subseteq D_v$. Intuitively, a process created from the process template can read and write any subset of variables in its own set of V in one atomic step; it also can read the shared variables of other processes. Let I be a set of index variables that range over natural numbers. The special index variable $\mathsf{id} \in I$ points to the process evaluating a guard. To construct guards, we define *conditions* as follows:

- *Index conditions* $(j = a)$ and $(j = k)$, where $j, k \in I$ are index variables, and $a \in \mathbb{N}$ is a constant index.
- *Variable conditions* $\mathbf{x}[j] = d$ and $\mathbf{x}[a] = d$, where $x \in V^\ell \cup V^s$ is a control variable, $d \in D_x$ is a value from x's domain, and $j \in I$ and $a \in \mathbb{N}$ are an index variable and a constant index respectively.

Having introduced conditions, we define guards:

- *Basic guard:* is either an index condition, or a variable condition.
- *Quantified guard:* $\exists j_1 \ldots \exists j_m.\phi(j_1 \ldots j_m)$ and $\forall k.\phi(k)$, where ϕ is a basic guard and $j_1, \ldots, j_m, k \in I$ are index variables. This type of guards are used, when we want to express a condition on the number of processes that are in a special state, for example having exactly two failed processes.

We define *commands* as expressions $\mathbf{x}[\mathsf{id}] := d$ and $\mathbf{x}[\mathsf{id}] := \mathbf{y}[a]$, if $x, y \in V$ are control variables with $D_x = D_y$, and $a \in \mathbb{N}$, $d \in D_x$, and y is in V^s of some process. The following grammar shows how guarded commands are constructed:

$$
\begin{array}{rcl}
EXP & ::= & GUARD \rightarrow COMMAND \\
GUARD & ::= & (GUARD \wedge BQG) \mid \\
 & & (GUARD \vee BQG) \mid BQG \\
BQG & ::= & \mathsf{basic} \mid \mathsf{quantified} \mid \neg BQG \\
COMMAND & ::= & COMMAND\ \mathsf{command;} \mid \mathsf{command;}
\end{array}
$$

We call a guard ϕ *normal*, if all its variable conditions access its own variables or the other processes' variables from V^s. In a fault command, the guards may be chosen not to be normal, and it is essentially permitted to formulate assumptions. For example, the upper bound on the number of faults in the system can be formulated by accessing other processes' Byzantine bits, which is not a shared variable.

A process template $P = \langle V, G\rangle$ can be instantiated as a process P_k (k is called the index of the process) by instantiating the set of control variables V_k from V, and the set of guarded commands G_k by replacing the index variable id with the index k. The special index variable id points to the process evaluating a guard. As the guards can refer to the variables of processes different from id by using a constant index, we identify the maximum index K that appears in index and variable conditions as well as in statements. That is, for every $j = a$ and $x[b]$, occurring in the guards and commands, it holds that $a \leq K$ and $b \leq K$. Informally, all but the first K processes behave similarly.

Given a process template P and a number $N \geq K$, one can construct a distributed program P^N by instantiating N processes with indices in $\mathcal{I}_N = \{1, \ldots, N\}$. Each control variable $v \in V$ can then be thought as a vector of size N, represented by $\mathbf{v}$, where $\mathbf{v}[k]$ is the variable instantiated in the process instance with index k. We, thus, have to deal with a parameterized family of programs $\mathcal{D} = \{P^N\}_{N \geq K}$, where each instance has a fixed number of processes.

a) Example.: We use a simplified *Byzantine agreement* problem (denoted $\mathcal{BA}$) as a running example to describe the concepts throughout the paper (see Algorithm 1). This problem consists of a fixed *general* process P_1 and a parameterized set of $N-1$ *lieutenant* processes. The shared vector $\mathbf{d}$ has N control variables, with domain $D_{\mathbf{d}[i]} = \{0, 1, \perp\}$, for all $i \in \mathcal{I}_N$. Each lieutenant process P_i, $2 \leq i \leq N$, is associated with variable $\mathbf{d}[i]$ (i.e., lieutenant i can write into $\mathbf{d}[i]$) and the general is associated with $\mathbf{d}[1]$. Moreover, each process has the following local variables: (1) a Boolean variable f

Algorithm 1 Byzantine agreement

1: Template variables $V^\ell = \{b, f\}, V^s = \{d\}, V = V^\ell \cup V^s$:
b, f : **boolean**; **init** $b = false, f = false$
$d : \{0, 1, \perp\}$; **init** $d = \perp$

2: Template guards G:
$(\mathsf{id} = 1) \wedge \mathbf{d}[\mathsf{id}] = \perp \rightarrow \mathbf{d}[\mathsf{id}] := choose\ from \{0, 1\}$,
3: $\mathbf{f}[\mathsf{id}] := true$;
$(\mathbf{d}[1] \neq \perp) \wedge (\mathbf{d}[\mathsf{id}] = \perp) \wedge \neg\mathbf{f}[\mathsf{id}] \wedge \neg\mathbf{b}[\mathsf{id}] \rightarrow \mathbf{d}[\mathsf{id}] := \mathbf{d}[1]$;
$(\mathbf{d}[1] \neq \perp) \wedge (\mathbf{d}[\mathsf{id}] \neq \perp) \wedge \neg\mathbf{f}[\mathsf{id}] \wedge \neg\mathbf{b}[\mathsf{id}] \rightarrow \mathbf{f}[\mathsf{id}] := true$;

4: Faults F:
$\forall k\,.\,\neg\mathbf{b}[k] \rightarrow \mathbf{b}[\mathsf{id}] := 1$;
$\mathbf{b}[\mathsf{id}] \rightarrow \mathbf{d}[\mathsf{id}] := 0/1/\perp, \mathbf{f}[\mathsf{id}] := true/false$;

which shows whether or not the decision of the process is finalized, and (2) a Boolean variable b which shows whether or not the process is Byzantine (i.e., faulty). As can be seen in Algorithm 1, the general process P_1 changes its decision to 0 or 1 arbitrarily, if its decision is $\perp$ (the first guarded command). As soon as the general process has a decision different from $\perp$, a lieutenant can start execution. If undecided (i.e., its d variable equals $\perp$), and its finalization and Byzantine bits are both false, it may copy the decision of the general (the second guarded command). After copying the general's decision, if the process is not Byzantine, it may change its finalization bit to *true* (the third guarded command). Note that any system instance initializes with $\mathbf{d}[k] = \perp$, $\mathbf{f}[k] = \mathit{false}$, and $\mathbf{b}[k] = \mathit{false}$, for all $k \in \mathcal{I}_N$. Also, note that "Faults" in this algorithm are not part of the things that a process does. They are actions that model the behavior of faults, and will be discussed later.

Definition 1. *Given a number of processes N, an N-state* valuation *is a function $\sigma : V \times \mathcal{I}_N \to \bigcup_{v \in V} D_v$, such that for each $v \in V$ and $j \in \mathcal{I}_N$, $\sigma(v, j) \in D_v$. We also define an* index valuation *as a partial function $\iota : I \to \{1, \ldots, N\}$.*

Definition 2. *Given a process template $P = \langle V, G \rangle$ on a set of atomic propositions* AP *and number of processes $N \in \mathbb{N}$, a system instance P^N is a labeled transition system $(\mathcal{S}_N, \mathcal{S}_N^0, \Delta_N, \lambda_N)$ defined as follows:*

- *$\mathcal{S}_N$ is the set of* global states*, i.e., the set of all N-state valuations;*
- *$\mathcal{S}_N^0 \subseteq \mathcal{S}_N$ is the set of* initial states*, where $\sigma \in \mathcal{S}_N^0$ iff $\forall v \in V.\ \forall j \in \mathcal{I}_N.\ \sigma(v, j) \in \mathit{Init}_v$;*
- *$\Delta_N \subseteq \mathcal{S}_N \times \mathcal{S}_N$ is the* transition relation*, where $(\sigma, \sigma') \in \Delta_N$ iff there is a process index $a \in \mathcal{I}_N$ and a guarded command $\phi \to \mathbf{x_1}[\mathsf{id}] := e_1; \ldots; \mathbf{x_m}[\mathsf{id}] := e_m$ from G such that the guard is satisfied when $\mathsf{id} = a$, and only $\mathbf{x_1}, \ldots, \mathbf{x_m}$ change.*
- *$\lambda_N : \mathcal{S}_N \to 2^{\mathsf{AP}}$ is the labeling function that maps each N-state valuation to a set of atomic propositions.*

A *computation* π of a system instance $P^N = (\mathcal{S}_N, \mathcal{S}_N^0, \Delta_N, \lambda_N)$ is an infinite sequence of states $\pi = s_0 s_1 \ldots$, such that $s_0 \in \mathcal{S}_N^0$, and for all $i \geq 0$, $(s_i, s_{i+1}) \in \Delta_N$.

B. Fault-Tolerance

Fault-tolerance is typically defined in terms of a fault model and satisfaction of two properties: (1) $\varphi_{\mathbf{complete}}$ is a property that should be satisfied in the absence of faults, and (2) $\varphi_{\mathbf{safety}}$ is a weaker *safety* property and need to be satisfied in both the absence and presence of faults.

a) Example.: The safety specification $\varphi_{\mathbf{safety}}$ for $\mathcal{BA}$ is:

- *Validity.* If the general process is non-Byzantine, then the final decision of no non-Byzantine lieutenant process can be different from the decision of the general process.
- *Agreement.* No two non-Byzantine lieutenant processes can finalize to different decisions.
- *Termination.* A finalized decision cannot be overturned.

Formally, in LTL:

$$\varphi_{\mathbf{safety}} =$$
$$\square\Big(\neg\mathbf{b}[1] \ \rightarrow \ (\forall j\,.\,(\neg\mathbf{b}[j] \wedge \mathbf{f}[j]) \rightarrow \mathbf{d}[j] = \mathbf{d}[1])\Big) \ \wedge \text{(validity)}$$
$$\square\Big(\forall i, j.\ (\neg\mathbf{b}[i] \wedge \neg\mathbf{b}[j] \wedge \mathbf{f}[i] \wedge \mathbf{f}[j]) \rightarrow \ \mathbf{d}[j] = \mathbf{d}[i]\Big) \text{(agreement)}$$
$$\square\,\forall j\ .\ \Big((\neg\mathbf{b}[j] \wedge \mathbf{f}[j] \wedge \mathbf{d}[j] = 0) \rightarrow \bigcirc(\mathbf{f}[j] \wedge \mathbf{d}[j] = 0)\Big) \wedge$$
$$\Big((\neg\mathbf{b}[j] \wedge \mathbf{f}[j] \wedge \mathbf{d}[j] = 1) \rightarrow \bigcirc(\mathbf{f}[j] \wedge \mathbf{d}[j] = 1)\Big) \text{(termination)}$$

A *process template with faults* is a tuple $P_F = \langle V, G, F \rangle$, where F is a set of guarded commands that specify the *faults* that may occur in a computation. We trivially extend Definition 2 and obtain $P_F^N = (\mathcal{S}_N, \mathcal{S}_N^0, \Delta_N^F, \lambda_N)$, where Δ_N^F is the union of process and fault transitions.

b) Example: Algorithm 1 is the process template of the $\mathcal{BA}$ example with two faults. If there exists no Byzantine process, one may become Byzantine. A Byzantine process may change its decision and (or) finalization bits arbitrarily to any value in their domains. As can be seen, the first guarded command is not normal, as the Byzantine bits are not part of the shared vector of the system. The reason for checking the Byzantine bit of other processes before going Byzantine is that we assume at each point of system execution, at most one process can be faulty. This is a simplified version of the well-known agreement problem. Note that the synthesis problem with even one fault is challenging in the parameterized setting.

A computation $s_0 s_1 \cdots$ is called *finitely-faulty*, if $\exists n \geq 0\,.\,\forall i \geq n\,.\,(s_i, s_{i+1}) \in \Delta_N \setminus \Delta_N^F$. That is, at some point, faults stop occurring. We assume all computations are finitely-faulty, which is required to ensure *recovery* in the synthesized fault-tolerant system. Note that this is an assumption simialr to fairness, which is not reflected in the description of algorithms.

Definition 3. *Let $P^N = (\mathcal{S}_N, \mathcal{S}_N^0, \Delta_N, \lambda_N)$ be a distributed program, and $\varphi_{\mathbf{complete}}$ be an LTL formula. A Boolean formula LS over the atomic propositions* AP *represents the set of* legitimate states *of P^N from $\varphi_{\mathbf{complete}}$, if (1) LS is closed in P^N, i.e., $\square(LS \rightarrow \square LS)$, (2) $P^N \models (LS \rightarrow \varphi_{\mathbf{complete}})$, and (3) there is no deadlocked computation starting from LS.*

The second condition enforces any computation starting from LS to satisfy $\varphi_{\mathbf{complete}}$, and the third one requires those computations to be infinite. We note the difference between *terminating* and *deadlocked* computations. When a computation $\overline{s}$ *terminates* in state s_t, the transition (s_t, s_t) is a member of the set of transitions Δ, and hence, $\overline{s}$ can be extended by stuttering at s_t. On the contrary, if a computation $\overline{s}$ reaches a state s_d, from where, there is no $s_i \in S$, such that $(s_d, s_i) \in \Delta$, then s_d is a deadlock state, and $\overline{s}$ is a deadlocked computation.

c) Example.: The legitimate states (LS) in the Byzantine agreement problem for its safety specification is:

$$\Big(\neg\mathbf{b}[1] \;\rightarrow (\forall j\,.\,\neg\mathbf{b}[j] \rightarrow (\mathbf{d}[j] = \bot \vee \mathbf{d}[j] = \mathbf{d}[1])) \wedge$$
$$(\forall j\,.\,\neg\mathbf{b}[j] \wedge \mathbf{d}[j] = \bot \rightarrow \neg\mathbf{f}[j])\Big) \wedge$$
$$\Big(\mathbf{b}[1] \;\rightarrow\; (\forall j\,.\,\mathbf{d}[j] = 0) \vee (\forall j\,.\,\mathbf{d}[j] = 1)\Big)$$

Note that the last line corresponds to the case when the general gets Byzantine, which means if the general is Byzantine, other processes agree with each other in LS. We should emphasize that this is not an initial state.

Also, the following specifies the set of terminating states:

$$\Big(\neg\mathbf{b}[1] \;\rightarrow\; (\forall j\,.\,\neg\mathbf{b}[j] \rightarrow (\mathbf{d}[j] = \mathbf{d}[1] \wedge \mathbf{f}[j]))\Big) \wedge$$
$$\Big(\mathbf{b}[1] \;\rightarrow\; (\forall j\,.\,(\mathbf{d}[j] = 0 \wedge \mathbf{f}[j])) \vee (\forall j\,.\,(\mathbf{d}[j] = 1 \wedge \mathbf{f}[j]))\Big)$$

When a fault occurs, the system execution may leave its legitimate states. Intuitively, a *fault-tolerant* distributed program should satisfy its safety specification $\varphi_{\mathbf{safety}}$ in the absence and presence of faults, and, when faults stops occurring, it returns to its set of legitimate states in a finite number of steps, from where it satisfies its complete specification $\varphi_{\mathbf{complete}}$. Since recovery to LS ensures satisfaction of $\varphi_{\mathbf{complete}}$, unlike the safety specification $\varphi_{\mathbf{safety}}$, we do not require that $\varphi_{\mathbf{complete}}$ is explicitly given. Note that we follow the seminal definition of fault-tolerance, adopted in the literature [23], [24], [25], [26], [12].

Definition 4. *A distributed program P^N with legitimate states LS, and safety specification $\varphi_{\mathbf{safety}}$ is F-tolerant to $\varphi_{\mathbf{safety}}$ from LS, if and only if:*

$$P_F^N \models \Box\big(\varphi_{\mathbf{safety}} \wedge (\neg LS \rightarrow \Diamond\, LS)\big)$$

P_F^N is the distributed program P^N subject to faults F. Note that the specifications should hold in reachable states, starting from the initial states and taking protocol actions and faults.

Notice that by Definition 3, in the above definition, eventual reachability of LS automatically implies satisfaction of the complete specification of the program.

III. Problem Statement

Given are a process template with a set of faults $P_F = \langle V, G, F\rangle$, a constant K, a safety specification $\varphi_{\mathbf{safety}}$, and a predicate LS, such that for every $N \geq K$, and $P^N = (\mathcal{S}_N, \mathcal{S}_N^0, \Delta_N, \lambda_N)$, we have

$$P^N \models \Box(LS \rightarrow \Box\, LS).$$

Our goal is to design a synthesis algorithm that generates a process template P', and a predicate LS', such that for every $N \geq K$, where $P'^N = (\mathcal{S}'_N, (\mathcal{S}_N^0)', \Delta'_N, \lambda'_N)$:

1) $\mathcal{S}'_N = \mathcal{S}_N$;
2) $(\mathcal{S}_N^0)' \subseteq (\mathcal{S}_N^0)$;
3) $(\mathcal{S}_N^0)' \neq \emptyset$;
4) $\lambda'_N = \lambda_N$;
5) $LS' \rightarrow LS$;
6) For every LTL formula χ, such that $P^N \models LS \rightarrow \chi$, we have $P'^N \models LS' \rightarrow \chi$, and
7) $P_F'^N$ is F-tolerant to $\varphi_{\mathbf{safety}}$ from LS'.

Note that condition 6 requires that all properties satisfied in LS (already mentioned as $\varphi_{\mathbf{complete}}$) are also satisfied in LS'. Our solution for preserving this condition is to not add any transitions inside the set of legitimate states. We will discuss more details in Section IV-C.

IV. Parameterized Synthesis Algorithm

In this section, we present our solution for synthesizing parameterized fault-tolerant distributed programs. The synthesis problem is known to be NP-complete for the concrete case [27] and undecidable in the parameterized case [17]. We propose a heuristic inspired by the algorithm introduced in [12] for the concrete case, augmented with *counter abstraction* [22] to handle parameterized models. The general idea is to generate a counter abstraction for the given template, and then remove the states that violate the safety specification. After that, deadlock states are resolved by adding recovery transitions to the model.

A. $\{0, ..., l, \infty\}$-Counter Abstraction

Definition 5. *Let $P = \langle V, G\rangle$ be a process template. A* local state valuation *is a function $\rho : V \rightarrow \bigcup_{v\in V} D_v$ such that for all $v \in V$, $\rho(v) \in D_v$. The set of all local state valuations defined with respect to V is denoted by $\mathcal{P}_V$.*

As we consider finite sets of variables over finite domains, the set $\mathcal{P}_V$ is finite. For example, the local states of each process in $\mathcal{BA}$ can be represented by tuple $\langle val_d, val_f, val_b\rangle$, which ranges of 12 possible combinations.

Further, given a system instance with N processes, we define function $\# : \mathcal{S}_N \times \mathcal{P}_V \rightarrow \mathbb{N}_0$; for every $\sigma \in \mathcal{S}_N$ and every $\rho \in \mathcal{P}_V$, where the value $\#(\sigma, \rho)$ equals to the number of processes in σ (with index $\geq K$) that are in local state valuation ρ. To define the abstract states, we introduce two sets of variables:

$$A = \big\{\mathbf{x}[a] \mid x \in V \;\wedge\; 1 \leq a \leq K\big\} \qquad B = \big\{\kappa_\rho \mid \rho \in \mathcal{P}_V\big\},$$

where $D_{\kappa_\rho} = \{0, \ldots, l, l+1\}$. Set A keeps K copies of the process variables, associated to the first K processes. Each variable κ_ρ from B plays the role of an abstract counter. That is, $\kappa_\rho = 0$ indicates zero processes, $\kappa_\rho = 1$ shows exactly one process, $\kappa_\rho = l$ denotes l processes, and value $\kappa_\rho = l + 1$ corresponds to more than l processes. Finally, we define $V_C = A \cup B$ and an *abstract state* as a valuation $\hat{\sigma} : V_C \rightarrow \bigcup_{v\in V_C} D_v$, such that for every $v \in V_C$, $\hat{\sigma}(v) \in D_v$. Each abstract state is thus represented by the valuation of the variables corresponding the fixed processes, and a tuple of size $\mid \mathcal{P}_V \mid$, where each number in this tuple corresponds to the abstract number of template processes in one local state $\rho \in \mathcal{P}_V$. We denote the set of all abstract states with $\mathcal{S}_C$.

Definition 6. *Given the number of processes $N \geq K$, we define* state abstraction *α_N as a function $\mathcal{S}_N \rightarrow \mathcal{S}_C$ that for every $\sigma \in \mathcal{S}_N$, returns an abstract state $\hat{\sigma} \in \mathcal{S}_C$ that satisfies the following properties:*

- *For every $x \in V$ and every index $i \in \{1, \ldots, K\}$, it holds $\hat{\sigma}(\mathbf{x}[i]) = \sigma(x, i)$.*

- *For every local state $\rho \in \mathcal{P}_V$, the following holds: if $\#(\sigma,\rho) \leq l$, then $\hat{\sigma}(\kappa_\rho) = \#(\sigma,\rho)$, and $\hat{\sigma}(\kappa_\rho) = l+1$, otherwise.*

Definition 7. *Let $P = \langle V, G\rangle$ be a process template. $C = (\mathcal{S}_C, \mathcal{S}_C^0, \Delta_C, \lambda_C)$ is a* counter abstraction *of the parameterized family $\{P^N\}_{N\geq K}$, if it satisfies the following conditions:*

- *As defined above, $\mathcal{S}_C$ is the set of all abstract states.*
- *The set of initial states $\mathcal{S}_C^0 \subseteq \mathcal{S}_C$ contains an abstract state $\hat{\sigma} \in \mathcal{S}_C$ if and only if for every variable $x \in V$:*
  - *The counter of each local state $\rho \in \mathcal{P}_V$ with $\rho(x) \notin Init_x$ is set to zero, i.e., $\hat{\sigma}(\kappa_\rho) = 0$.*
  - *Each fixed process $i \leq K$ respects the initial values, i.e., $\hat{\sigma}(x[i]) \in Init_x$.*
- *$(\hat{\sigma}, \hat{\sigma}') \in \Delta_C$ if and only if there exists a size $N \geq K$ and global states $\sigma, \sigma' \in \mathcal{S}_N$ such that $\hat{\sigma} = \alpha_N(\sigma)$, $\hat{\sigma}' = \alpha_N(\sigma')$, and $(\sigma,\sigma') \in \Delta_N$.*
- *$p \in \lambda_C(\hat{\sigma})$ if and only if there exists a size $N \geq K$ and a global state $\sigma \in \mathcal{S}_N$ with $p \in \lambda_N(\sigma)$.*

a) Example.: Consider the abstraction level l to be 1 in the counter abstraction model of $\mathcal{BA}$. Each abstract state $\hat{\sigma} \in \mathcal{S}_C$ consists of two parts; the state of the template processes (lieutenants), and the state of the fixed process (general), including the values of $\mathbf{d}[1]$, $\mathbf{b}[1]$, and $\mathbf{f}[1]$. The state of the template processes is represented by a 12-tuple, since there are 12 local states for the template processes ($|D_d| = 3$, $|D_f| = 2$, $|D_b| = 2$, and hence, $|\mathcal{P}_V| = 12$). Each element of the 12-tuple is a number (0,1, or 2), showing the abstract number of processes in the corresponding local state. For example, if the first element of the 12-tuple corresponds to the local state of $\langle d = \perp, f = 0, b = 0\rangle$, and this element is 0, it shows that there are no processes with $d = \perp$, $f = 0$, and $b = 0$. For example, consider the following transition:

$$\big[\langle \mathbf{2},0,0,0,0,0,0,\mathbf{1},0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0, \mathbf{f}[1]=1\big] \rightarrow \big[\langle \mathbf{2},0,0,0,0,0,0,\mathbf{2},0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0, \mathbf{f}[1]=1\big]$$

Let the first element of the 12-tuple represent the local state $\langle d = \perp, f = 0, b = 0\rangle$ and the 8th element represent the local state $\langle d = 1, f = 0, b = 0\rangle$. This transition asserts that the number of lieutenant processes with local state $\langle d = 1, f = 0, b = 0\rangle$ changes from one to two and number of lieutenants with local state $\langle d = \perp, f = 0, b = 0\rangle$ remains more than one.

B. Read/Write Restrictions

Read restrictions in a distributed setting stems from partial observability of processes. Consider the set $\{P_1, \ldots, P_N\}$ of processes. The *read-set* of each process P_i, where $1 \leq i \leq N$ (denoted as R_{P_i}) is the set of all variables that the process can read, i.e., $R_{P_i} = V_{P_i}^l \cup V^s$. The *write-set* of a process P (denoted as W_{P_i}) is the set of variables a process is allowed to write into, i.e., $W_{P_i} = V_{P_i}^l \cup V_{P_i}^s$. Notice that $W_{P_i} \subseteq R_{P_i}$, and hence, a process cannot write into a variable blindly.

a) Example.: In $\mathcal{BA}$, the read-set of each process P_i is $R_{P_i} = \{\mathbf{d}, \mathbf{f}[i], \mathbf{b}[i]\}$, and its write-set is $W_{P_i} = \{\mathbf{d}[i], \mathbf{f}[i]\}$.

Definition 8. *For $P^N = (\Sigma_N, \Delta_N, \mathsf{AP}_N, \lambda_N, Init_N)$, the read restriction is defined as follows:*

$$\forall (s_1, s_1') \in \Delta_N . \exists j \in [1,N] . \exists v \in W_{P_j} . \Big(s_1(v) \neq s_1'(v)\Big) \Rightarrow \Big(\forall s_2, s_2' \in \Sigma_N . \big[(\forall v \notin W_{P_j} : (v(s_1) = v(s_1') \wedge v(s_2) = v(s_2'))) \wedge (\forall v \in R_{P_j} : (v(s_1) = v(s_2) \wedge v(s_1') = v(s_2')))\big] \implies (s_2, s_2') \in \Delta_N\Big) \quad (1)$$

In other words, the action of each process does not depend on the variables it cannot read. If two states are the same, except for the values of variables not in the read-set of a process, and the process has a transition starting from one of these states, it has similar transition starting from the other one, as well. We say that such transitions are in the same *group*.

b) Example.: Consider a $\mathcal{BA}$ instance, P^3 (one general and two lieutenants). The following two transitions

$$t_1 = \Big(\big[\mathbf{d} = \langle 0,\perp,0\rangle, \mathbf{b} = \langle 0,0,1\rangle, \mathbf{f} = \langle 1,0,1\rangle\big], \big[\mathbf{d} = \langle 0,0,0\rangle, \mathbf{b} = \langle 0,0,1\rangle, \mathbf{f} = \langle 1,0,1\rangle\big]\Big)$$
$$t_2 = \Big(\big[\mathbf{d} = \langle 0,\perp,0\rangle, \mathbf{b} = \langle 0,0,1\rangle, \mathbf{f} = \langle 1,0,0\rangle\big], \big[\mathbf{d} = \langle 0,1,0\rangle, \mathbf{b} = \langle 0,0,1\rangle, \mathbf{f} = \langle 1,0,0\rangle\big]\Big)$$

have the same effect as far as P_2 is concerned (since P_2 cannot read $\mathbf{f}[3]$). This implies that if t_1 is included in the set of transitions of a distributed program, then so should t_2. Otherwise, execution of t_1 by P_2 will depend on the value of $\mathbf{f}[3]$, which, of course, P_2 cannot read.

In a model in counter abstraction, the read restriction is a bit different. The reason is that each state is determined based on the abstract number of processes in each local state, and hence, the exact information on the read-set of each process is not available. Before defining read restriction in counter abstraction, we need to have the definition of the abstract number of processes with some variable valuation. For a variable $v \in V$ and some valuation $val \in D_v$, the set of local states, where variable v has the value val, is denoted by $\mathcal{P}_V^{(v=val)}$. More formally, $\mathcal{P}_V^{(v=val)} = \{\rho \in \mathcal{P}_V \mid \rho(v) = val\}$. For a set X with domain $0 \leq i \leq l+1$ (l is abstraction level), the abstract sum of the elements in X is defined as below:

$$\hat{\Sigma}(X) = \begin{cases} \sum\limits_{i\in X} i & \text{if } \sum\limits_{i\in X} \leq l \\ l+1 & \text{otherwise} \end{cases}$$

We define the abstract number of a variable $v \in V$ with value val in an abstract state $\hat{\sigma} \in \mathcal{S}_C$ as follows:

$$num(\hat{\sigma}, v, val) = \hat{\sum_{\rho \in \mathcal{P}_V^{(v=val)}}} \hat{\sigma}(\kappa_\rho)$$

For a process P_i, we say that two abstract states $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are the same (i.e., the part of the global state that the

process can read is the same in both abstract states), denoted as $\hat{\sigma}_1 \overset{P_i}{\approx} \hat{\sigma}_2$, if and only if:

$$\hat{\sigma}_1 \overset{P_i}{\approx} \hat{\sigma}_2 \Leftrightarrow \forall\, v \in R_{P_i} \,.\, \Big((\forall\, i \in \mathcal{I}_K \,.\, \hat{\sigma}_1(v[i]) = \hat{\sigma}_2(v[i])) \wedge num(\hat{\sigma}_1, v, val) = num(\hat{\sigma}_2, v, val)\Big)$$

Recall that $\mathcal{I}_K$ is the set of indexes of the fixed processes. In other words, two abstract states are the same for a process P_i, if (1) for each local variable $v \in V_l$, the value of the variable in P_i is the same, and (2) for each shared variable $v \in V_s$ and each value $val \in D_v$, the abstract number of the variables v with value val is the same in both abstract states. Since the system is asynchronous, each transition in Δ_C corresponds to the execution of only one process. A subset of Δ_C that corresponds to the execution of the process P_i is identified by Δ_{P_i}. Having the above definitions, we can define the read restriction constraint in counter abstraction:

$$\forall\, (\hat{\sigma}_1, \hat{\sigma}_1') \in \Delta_{P_i} \,.\, \forall\, \hat{\sigma}_2, \hat{\sigma}_2' \in \mathcal{S}_C \,.\, (\hat{\sigma}_1 \overset{P_i}{\approx} \hat{\sigma}_2 \wedge \hat{\sigma}_1' \overset{P_i}{\approx} \hat{\sigma}_2' \wedge (\forall v \notin R_{P_i} : num(\hat{\sigma}_2, v, val) = num(\hat{\sigma}_2', v, val))) \Rightarrow (\hat{\sigma}_2, \hat{\sigma}_2') \in \Delta_{P_i} \quad (2)$$

c) Example.: Consider the abstraction level $l = 1$ in $\mathcal{BA}$ example. Transitions

$$\begin{aligned} t_1 =& [\langle 2,0,0,0,0,0,0,0,1,0,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0] \rightarrow \\ & [\langle 2,0,0,0,0,0,0,0,2,0,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0], \\ t_2 =& [\langle 2,0,0,0,0,0,1,0,0,0,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0] \rightarrow \\ & [\langle 2,0,0,0,0,0,1,0,1,0,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0] \end{aligned}$$

are in the same group. Note that each state is represented by the state of the general process, along with a 12-tuple, each showing the abstract number of processes in one of the 12 possible local states. The action corresponding to both transitions in the example above corresponds to a lieutenant process in the local state $\langle d = \perp, b = 0, f = 0\rangle$, changing its decision to 1 (from the local state in the first element of the 12-tuple to the local state in the 8th element). The read-set of each lieutenant is its Byzantine and finalization bits, along with the decision bits of all processes. The Byzantine and finalization bits of the executing process are the same in the source and destination of both transitions. Also, the abstract number of lieutenants with decision 0, decision 1, and decision $\perp$ are the same in the source/destination of both transitions (their cardinality is 0,1,2 in the sources and 0,2,2 in the destinations, respectively). We had to skip the details of the local states corresponding to each element of the tuple to save the paper space.

C. Synthesis Algorithm

Our proposed algorithm consists of 7 steps (Algorithm 2). We repeat steps 3-4, 3-6, and 3-7 until a fixpoint is reached, which means no more progress is possible. We represent the fixpoint computation by nested repeat-until loops. In the following, we discuss each step of the algorithm in detail.

Step 1. In the first step, the abstract model $C = (\mathcal{S}_C, \mathcal{S}_C^0, \Delta_C, \lambda_C)$ is generated from the fault-intolerant parameterized system. Then, we put a flag, *remove*, for each state and transition (initially set as *false*) that shows whether or not it has been removed during synthesis. If we decide to remove a state or transition, we simply switch its flag, and if we need to put it back, we switch the flag back. After generating the counter abstraction C, Algorithm 2 is called on C, ψ, LS, and TS.

Algorithm 2 Parameterized Synthesis

Require: A counter abstraction $C = (\mathcal{S}_C, \mathcal{S}_C^0, \Delta_C, \lambda_C)$, a specification $\varphi_{\mathbf{safety}}$, a set LS of legitimate states and a set TS of terminating states.
Ensure: If successful, a counter abstraction $C' = (\mathcal{S}_C', (\mathcal{S}_C^0)', \Delta_C', \lambda_C')$, and legitimate states LS'.

```
1: (Δ_C, S_C) = label_transitions_states(Δ_C, S_C, TS)
2: LS' = LS, S'_C = S_C, Δ'_C = Δ_C
3: repeat
4:     LS'' = LS'
5:     repeat
6:         Δ''_C = Δ'_C
7:         repeat
8:             S''_C = S'_C
9:             S'_C = remove_unreachable (S'_C, Δ'_C)
10:            S'_C = remove_fte (S'_C, Δ'_C, fte)
11:            S'_C = remove_badstate (S'_C, Δ'_C)
12:        until S''_C = S'_C
13:        add_recovery (S'_C, Δ'_C)
14:        (fte, ofd) = remove_deadlock (S'_C, Δ'_C, fte, ofd)
15:    until Δ''_C = Δ'_C
16:    LS' = construct_LS (LS', ofd)
17: until LS'' = LS'
18: (S^0_C)' = {σ | σ ∈ S'_C ∧ σ ∈ S^0_C}
19: return (S'_C, (S^0_C)', Δ'_C, λ'_C)
```

Step 2. In line 1, we label each transition (non-fault ones) with two indexes showing which local states get incremented and decremented by that transition. This labeling is done based on the original algorithm. As an example, in $\mathcal{BA}$, if we consider the transition from the state $[\langle 2,1,0,0,0,0,0,2,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0]$ to the state $[\langle 2,0,0,0,0,0,0,2,0,0,0,0\rangle, \mathbf{d}[1]=1, \mathbf{b}[1]=0]$, we are guaranteed that the local state at the second element is decremented, since its value changes from 1 to 0. However, it is not obvious which local state gets incremented (first or 8th). Assume that the second element corresponds to the abstract number of lieutenants in local state $\langle d=1, f=0, b=0\rangle$. According to $\mathcal{BA}$, the only possible action for a lieutenant in this local state is to finalize on its decision and move to the local state $\langle d=1, f=1, b=0\rangle$. Assuming that the 8th element corresponds to this local state, we can label this transition by 8 as its incrementing index. Besides labeling transitions, we also label each state by a flag that shows whether this state is terminating or not. By this flag our synthesizer can distinguish terminating states from the deadlock ones.

Step 3. In line 9, a function named remove_unreachable is called to remove the unreachable states of the model. An unreachable state is the one with no incoming transitions. To do that, we simply put their *remove* flags to *true*. The outgoing transitions from unreachable states are not removed to keep the maximum reachability in the model (note that removing a transition results in removing all transitions in its group). Initially, this step forms the reachable state space of the protocol starting from the initial states and taking protocol actions and faults. After removing transitions in the next steps

of the algorithm, there will be unreachable states that should be removed by this step.

Step 4. In lines 10 and 11, these sets of states become unreachable: (1) *fte*, (failed to eliminate) states, which are identified in Step 5, and (2) bad states, which are the states violating the safety specification. In order to remove bad states, we make them unreachable. If a bad state is reachable from a set of states only by fault(s), they also become unreachable, since we do not have control over the execution of faults (faults cannot be removed).

Step 5. Deadlock states are not allowed in the synthesized program. For abstract states not satisfying LS, being deadlocked means violation of fault-tolerance, and for abstract states where LS is satisfied, being deadlocked means violating the requirement of satisfying the properties satisfied by the original algorithm. Deadlock states are resolved by either adding recovery paths (this step) or deadlock state elimination (next step). In this step, we identify deadlock states not satisfying LS, and for each, try to resolve it by adding a recovery transition to a state satisfying LS. Before presenting the conditions for adding recovery transitions, we define the increment and decrement relations between two local states.

Definition 9. *An abstract state $\hat{\sigma}_2$ is an* increment *of the abstract state $\hat{\sigma}_1$, considering the local state ρ, and is represented by $inc(\hat{\sigma}_1, \hat{\sigma}_2, \rho)$, if and only if: $\hat{\sigma}_1(\kappa_\rho) = \hat{\sigma}_2(\kappa_\rho) = 2 \ \vee \ \hat{\sigma}_2(\kappa_\rho) = \hat{\sigma}_1(\kappa_\rho) + 1$. An abstract state $\hat{\sigma}_2$ is a* decrement *of the abstract state $\hat{\sigma}_1$, considering the local state ρ, and is represented by $dec(\hat{\sigma}_1, \hat{\sigma}_2, \rho)$, if and only if: $inc(\hat{\sigma}_2, \hat{\sigma}_1, \rho)$.* □

A recovery transition $(\hat{\sigma}_1, \hat{\sigma}_2)$ can be added during the synthesis procedure, if all the following conditions hold:

1) The asynchronous condition:

$$\Big(\exists j \in \mathcal{I}_K \,.\, \exists x \in V \,.\, \hat{\sigma}_1(x,j) \neq \hat{\sigma}_2(x,j) \wedge (\forall i \in \mathcal{I}_K \,.\, \forall x \in V \,.\, i \neq j \rightarrow \hat{\sigma}_1(x,i) = \hat{\sigma}_2(x,i)) \wedge (\forall \rho \in \mathcal{P}_V \ : \hat{\sigma}_1(\kappa_\rho) = \hat{\sigma}_2(\kappa_\rho))\Big) \vee \Big((\forall i \in \mathcal{I}_K \,.\, \forall x \in V \,.\, \hat{\sigma}_1(x,i) = \hat{\sigma}_2(x,i)) \wedge \exists \rho_1, \rho_2 \in \mathcal{P}_V \ : inc(\hat{\sigma}_1, \hat{\sigma}_2, \rho_1) \wedge dec(\hat{\sigma}_1, \hat{\sigma}_2, \rho_2) \wedge (\forall \rho \in \mathcal{P}_V \,.\, \rho \neq \rho_1 \wedge \rho \neq \rho_2 \rightarrow \hat{\sigma}_1(\kappa_\rho) = \hat{\sigma}_1(\kappa_\rho))\Big)$$

   The intuitive meaning of this condition is that in each transition, one and only one process can take action. Either a fixed process changes its local state, and all other processes remain unchanged, or all fixed processes are unchanged, and a template process changes its local state.
2) $(\hat{\sigma}_1, \hat{\sigma}_2) \models \psi$.
3) It does not add a transition to a state satisfying LS to ensure that all properties satisfied by the original algorithm will be satisfied in the synthesized one.
4) It does not create a *bad cycle* in the system.

Considering condition 1, when two local states are changing, we could find out which action is being taken by a process. If only one local state changes, then the local states with cardinality 2 are the candidates for the other changing local state. For each candidate, we check the corresponding action to see if it satisfies the other conditions. If there exists a cycle outside the set of legitimate states, fault-tolerance is violated (Definition 4). In counter abstraction, not all cycles are bad, as the cycle in an abstract model may not occur in any of the system instances.

Definition 10. *A* bad cycle *in a counter abstraction is a cycle consisting of only abstract states satisfying $\neg LS$, with the condition that there exists a system instance for which the bad cycle is instantiated to a cycle.*

In counter abstraction, not every cycle is a bad cycle. We recognize two categories of spurious cycles. The automaton shown in the bottom of Fig. 1 shows an example of the first category of spurious cycles (ii and di stand for increment index and decrement index, respectively). In this cycle, there are two transitions; one changing the local state of a process from the first element of the tuple to the third element, and the other changing the local state of a process from the second element to the first element. Therefore, in each iteration of this cycle, the number of processes with local state corresponding to the third element is incrementing and the number of processes in the local state corresponding to the second element is decrementing. Since the number of processes in each local state is finite, this cycle cannot be a real one in a concrete model, and hence, is not a bad cycle. Considering the cycle in automaton shown in the top of Fig. 1, we cannot have a similar argument, as it constantly moves a process from local state at the first element to the local state at the third element, and vice versa, and hence, there is a possibility that this cycle is indeed a bad one.

Another category of cycles can be considered as spurious, considering the notion of *weak fairness*. Based on weak fairness, if a transition is continuously enabled, it will be eventually taken. The automaton in the middle of Fig. 1 depicts an example of these cycles. In this cycle, there is one transition that is always enabled, and can break the cycle, if taken. According to weak fairness, in this situation, this transition will be eventually taken, meaning that no concrete system will be stuck in this cycle forever. Note that when a transition is enabled, it is either enabled for one or more than one process. In the case of more than one process, the argument is still valid, since the processes in the same local state behave similarly.

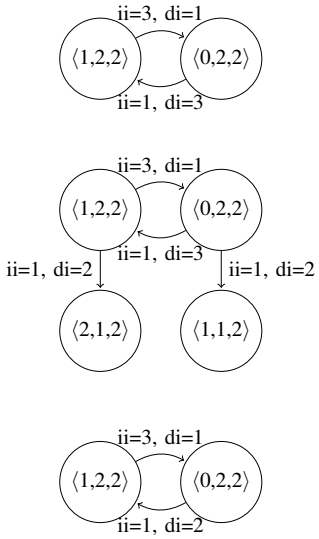

Fig. 1: Top: Bad Cycle, Middle, Bottom: Spurious Cycles

Our synthesizer can easily filter out spurious cycles belong-

ing to one of the two categories mentioned above. Inclusion in the first category can be checked by the following steps: 1- List all the loop transitions, 2- Match each transition's increment index by the decrement index of another transition, 3- If this matching cannot be done for all the indexes, then it concludes that this cycle is spurious and belongs to the first category.

For checking the inclusion in the second category, our synthesizer does the following steps: 1- List all the loop nodes, 2- Check if there is at least one transition (with similar increment and decrement indexes) is in the set of outgoing transitions of all the nodes. 3- If such a transition exists, our synthesizer concludes that this loop is spurious and belongs to the second category.

Step 6. The deadlock states that cannot be resolved using the previous step are removed by making the deadlock states unreachable (removing their incoming transitions). There are two types of transitions reaching a state; program transitions, and faults. If the state is reachable by a fault, we should backtrack, and make the source of the fault unreachable, as well. The reason is that we cannot prevent the occurrence of a fault. Such states are added to the list of "failed to eliminate" states (fte). If the source of the fault satisfies LS, it is added to the ofd list (to be removed in the next step), and if not, it is added to the set of deadlock states to be removed in the next rounds of this step. To remove a state from the set of deadlock states, all incoming transitions to each deadlock state, along with the transitions in its group, are removed from the counter abstraction. If this removal makes a new deadlock state (called nds) in the counter abstraction, we put all the transitions back to the model, and add nds to the set of deadlock states to be removed in the next rounds of this step.

Step 7. In this step, the states in the ofd list are made unreachable. If this removal leads to creating new deadlock states satisfying LS, we make them unreachable as well to prevent any deadlocks satisfying LS. Note that the terminal states are recognized from the deadlock ones. If all initial states get removed in this step, the algorithm will terminate with failure.

Theorem 1. *Algorithm 2 is sound.*

We prove this theorem by showing that all conditions reuqired by the problem statement are satisfied in the solution.

1) The first condition requires that the set of states are the same in the original and the synthesized program. It holds as the state space is not changed during the synthesis algorithm.
2) The second condition requires that the set of initial states in the synthesized protocol is a subset of the initial states in the original protocol. It trivially holds, since we do not change the "initial" label of states.
3) The third condition requires that the set of initial states is not empty. This is guaranteed, as the initial states are in LS, and legitimate states can be removed in Step 7 of the algorithm. In this step, we check for the emptiness of the set of initial states, and if that happens, the algorithm returns with failure.
4) The fourth condition trivially holds, as the labeling is not changed during the synthesis algorithm.
5) The fifth condition needs that in every state that satisfies LS', LS is also satisfied. This condition is guaranteed as the set of states satisfying LS' is a subset of states satisfying LS in the original algorithm.
6) The sixth condition requires that all LTL formulas satisfied by the states satisfying LS in the original system are also satisfied by the states satisfying LS' in the synthesized system. This condition holds by our algorithm, since no transition is added to the set of states satisfying LS, during our synthesis.
7) The last condition is that the protocol is F-tolerant. We prove that by showing that each condition in the definition of fault-tolerance is respected:
   - LS' in the resulting system is closed due to the fact that no behavior is added to LS, and the original algorithm satisfies the closure of LS. Also, in the last step, we do not let any deadlock states in LS.
   - Every computation in the resulting protocol satisfies the safety specification ψ. This is correct since every state violating the safety specification becomes unreachable during the synthesis. Also, when adding each recovery transition, the corresponding action is checked with respect to ψ. We should prove that if no prohibited transition is added in the abstract level, then no prohibited transition is added in any of the concrete models as well. We can prove that by proof by contradiction. Assume that there exists a prohibited transition in a concrete model. Then, there should exist at least one corresponding prohibited transition in the abstract model, and we know that no prohibited transition is added in the abstract level. Hence, the method is sound.
   - The last condition requires convergence to LS. This is also satisfied, since there are finite number of states in the state space of the program, and none of the states satisfying $\neg LS$ are deadlocked. Since there is no cycle in states satisfying $\neg LS$ either, all computations starting from a state satisfying $\neg LS$ eventually get to a state satisfying LS.

V. Case Studies and Experimental Results

We have fully implemented our algorithm. To construct the abstract transition relation Δ_C and the labeling function λ_C, we used the tool ByMC [28]. Then, we fed the abstract transition system and other specifications as input to our synthesizer, which gives a fault-tolerant algorithm as output. We ran the all experiments on a laptop equipped with Intel Core i7-6700HQ 2.60 GHz and Java heap size of 256MB. Table I summarizes our synthesis results. There are cases that the synthesis failed for an abstraction level, and hence, we used the next abstraction levels to synthesize the solution. We have checked all the synthesized protocols using a parameterized verifier [2] to make sure about their correctness.

A. Case study 1 & 2: Byzantine agreement

The first case study is our $\mathcal{BA}$ running example throughout the paper. Our synthesizer takes as input an abstract transition

system with abstraction level set to 1, along with $\varphi_{\mathbf{safety}}$, LS, and TS specifications. The synthesizer took 2 seconds to synthesize the solution.

We also considered $\mathcal{BA}$ with *crash faults*. That is, a lieutenant process may stop doing any actions and never recovers. Variables and behavior of this problem are very similar to the $\mathcal{BA}$ problem. We employ an additional variable C that shows whether or not a process has crashed. The behavior of this fault-intolerant algorithm is presented in Pseudo code 3.

The set of legitimate states can be formalized as follows:

$$\Big(\neg\mathbf{b}[1] \rightarrow \forall j\,.\,(\mathbf{b}[j] \vee \mathbf{c}[j] \vee \mathbf{d}[j] = \mathbf{d}[1] \vee (\mathbf{d}[j] = \bot \wedge \neg\mathbf{f}[j]))\Big) \wedge \Big(\mathbf{b}[1] \rightarrow (\forall j\,.\,(\mathbf{c}[j] \vee \mathbf{d}[j] = 0) \vee \forall j\,.\,(\mathbf{c}[j] \vee \mathbf{d}[j] = 1))\Big)$$

and the set of terminating states are as follows:

$$\Big(\neg\mathbf{b}[1] \rightarrow \forall j\,.\,(\mathbf{b}[j] \vee \mathbf{c}[j] \vee (\mathbf{d}[j] = \mathbf{d}[1] \wedge \mathbf{f}[j]))\Big) \wedge \Big(\mathbf{b}[1] \rightarrow \forall j\,.\,(\mathbf{c}[j] \vee (\mathbf{d}[j] = 0 \wedge \mathbf{f}[j])) \vee \forall j\,.\,(\mathbf{c}[j] \vee (\mathbf{d}[j] = 1 \wedge \mathbf{f}[j]))\Big)$$

The required specifications of this problem are the following:

- *Validity*, which means that if the general process is non-Byzantine, then the final decision of any non-Byzantine lieutenant process that has not crashed should be the same as the decision of the general process.
- *Agreement*, meaning that any two non-Byzantine alive lieutenants should finalize to the same decision.

The above requirements can be formalized as follows:

$$\Box\Big((\neg\mathbf{b}[1] \wedge \mathbf{d}[1] \neq \bot) \rightarrow \forall j\,.\,(\neg\mathbf{b}[j] \wedge \neg\mathbf{c}[j] \wedge \mathbf{f}[j]) \rightarrow \mathbf{d}[j] = \mathbf{d}[1]\Big) \quad (\varphi_{\mathbf{safety}})$$

$$\Box\Big(\forall i, j\,.\,(\neg\mathbf{b}[i] \wedge \neg\mathbf{b}[j] \wedge \neg\mathbf{c}[i] \wedge \neg\mathbf{c}[j] \wedge \mathbf{f}[i] \wedge \mathbf{f}[j]) \rightarrow \mathbf{d}[j] = \mathbf{d}[i]\Big) \quad (\varphi'_{\mathbf{safety}})$$

There is also a transition-based specification $\varphi''_{\mathbf{safety}}$ that prevents a lieutenant process with a finalized decision from changing its decision value or the finalization bit later.

$$\Box\Big((\mathbf{f}[j] \wedge \mathbf{d}[j] = 1) \rightarrow \bigcirc(\mathbf{f}[j] \wedge \mathbf{d}[j] = 1)\Big) \wedge \Big((\mathbf{f}[j] \wedge \mathbf{d}[j] = 0) \rightarrow \bigcirc(\mathbf{f}[j] \wedge \mathbf{d}[j] = 0)\Big) \quad (\varphi''_{\mathbf{safety}})$$

Setting the abstraction level to 1, we created the abstract transition system and fed it to our synthesizer, along with the legitimate states LS, safety specifications, and terminating states TS. Our synthesizer generated the fault-tolerant algorithm in less than 12 seconds.

Algorithm 3 Byzantine Agreement with Fail-Stop

1: Template variables

$$V^{\ell} = \{b, f, c\},$$
$$V^{s} = \{d\}, V = V^{\ell} \cup V^{s}:$$

b, f, c : **boolean**; **init** $b = false, f = false, c = false$
$d : \{0, 1, \bot\}$; **init** $d = \bot$

2: Template guards G:

$$(\mathsf{id} = 1) \wedge \mathbf{d}[\mathsf{id}] = \bot \rightarrow \mathbf{d}[\mathsf{id}] := choose\ from\{0, 1\}, \mathbf{f}[1] := true;$$

$$(\mathbf{d}[1] \neq \bot) \wedge (\mathbf{d}[\mathsf{id}] = \bot) \wedge \neg\mathbf{f}[\mathsf{id}] \wedge \neg\mathbf{b}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \rightarrow \mathbf{d}[\mathsf{id}] := \mathbf{d}[1];$$
$$(\mathbf{d}[1] \neq \bot) \wedge (\mathbf{d}[\mathsf{id}] \neq \bot) \wedge \neg\mathbf{f}[\mathsf{id}] \wedge \neg\mathbf{b}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \rightarrow \mathbf{f}[\mathsf{id}] := true;$$

3: Faults F:

$$\forall k\,.\,\neg\mathbf{b}[k] \wedge \neg\mathbf{c}[\mathsf{id}] \rightarrow \mathbf{b}[\mathsf{id}] := 1;$$
$$\mathbf{b}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \rightarrow \mathbf{d}[\mathsf{id}] := 0/1/\bot, \mathbf{f}[\mathsf{id}] := true/false;$$
$$\forall k\,.\,\neg\mathbf{c}[k] \rightarrow \mathbf{c}[\mathsf{id}] := 1;$$

B. Byzantine reliable broadcast with fail-stop

In a broadcast protocol, one process (sender) intends to send a message to all other processes (receivers). The protocol should satisfy (1) *validity*, i.e., if the sender is correct then every correct receiver eventually delivers the message, and (2) *agreement*, i.e., if a correct receiver delivers a message, then every correct receiver eventually delivers the message. The sender and at most one receiver may crash. One receiver may become Byzantine. A Fault-intolerant broadcast algorithm is presented in Pseudo code 4.

a) Variables: There is a shared Boolean vector r, where each receiver process P_i, $2 \leq i \leq N$, is associated with the variable $\mathbf{r}[i]$ (showing that the receiver has received the message), and the sender is associated with $\mathbf{r}[1]$ (showing that the sender has something to send). Moreover, each process has the following local variables: (1) a Boolean variable d, which shows whether or not the received message is delivered, (2) a Boolean variable b which shows whether or not the process is Byzantine, and (3) a Boolean variable c which shows whether or not the process has crashed.

b) Behavior: As can be seen in Pseudo code 4, we simulate message passing by shared memory. When the sender process (P_1) decides to broadcast a message, it sets its ready flag to 1 ($\mathbf{r}[1] = 1$). When other processes see this flag, they receive the message by setting their received flag to 1 ($\mathbf{r}[k] = 1$), and then deliver it by setting their delivered flag to 1 ($\mathbf{d}[k] = 1$). If the sender crashes at any time, it sets its ready flag to 0, meaning that the sender does not broadcast a message anymore, and also changes its crash flag to 1 ($\mathbf{c}[1] = 1$). When a receiver process crashes, it similarly sets its received flag to 0, and its crash flag to 1. Additionally, receivers may become Byzantine, when they can set and unset their received and delivered flags randomly.

c) Specifications: Our goal is to synthesize a special type of fault-tolerant broadcast, called reliable broadcast, with the following requirements:

$$\Box\Big((\mathbf{r}[j] \wedge \neg\mathbf{b}[j]) \rightarrow \bigcirc(\mathbf{r}[j] \vee \mathbf{c}[j])\Big) \wedge \Big((\mathbf{d}[j] \wedge \neg\mathbf{b}[j]) \rightarrow \bigcirc\mathbf{d}[j]\Big) \quad (3)\ (\varphi_{\mathbf{safety}})$$

This transition-based specification makes sure that when a correct process receives a message, it cannot change its received flag unless it crashes. It also requires that a message delivered by a process, cannot be undelivered.

The set of terminating states is formalized as follows:

$$\Big(\mathbf{c}[1] \ \rightarrow\ \exists j\,.\ (\mathbf{d}[j] \wedge \neg\mathbf{c}[j] \wedge \neg\mathbf{b}[j]) \rightarrow \forall i\,.\,(\mathbf{c}[i] \vee \mathbf{d}[i] \vee \mathbf{b}[i])\Big) \wedge \Big(\neg\mathbf{c}[1] \rightarrow \forall j \neq 1\,.\,(\mathbf{c}[j] \vee \mathbf{d}[j] \vee \mathbf{b}[j])\Big) \quad (TS)$$

Note that this formalization of terminating states, along with the lack of cycles in legitimate states, ensures that in reliable broadcast, either all or none of the correct processes deliver the message. The set of legitimate states is formalized as follows:

$$\mathbf{c}[1] \ \rightarrow\ \forall j\,.\,(\mathbf{c}[j] \vee \mathbf{b}[j] \vee \mathbf{d}[j]) \ \vee\ \forall j\,.\,(\mathbf{c}[j] \vee \mathbf{b}[j] \vee \neg\mathbf{d}[j]) \quad (LS)$$

We tried to synthesize a reliable broadcast by setting the abstraction level to 1, and creating the abstract transition system. However, the synthesizer was not able to generate a fault-tolerant algorithm. Note that our synthesis algorithm is not complete, due to the abstraction. Setting the abstraction level to 2, we got the same result. Finally, we set the abstraction level to 3, and the synthesizer could generate a fault-tolerant version of this broadcast algorithm in less than 9 seconds.

C. Byzantine uniform reliable broadcast with fail-stop

Here, we strengthen the specification of reliable broadcast as follows. If a process p delivers the message, then every process should deliver it as well, whether or not p is crashed after delivery of the message. Variables and behavior of this algorithm are the same as the previous case study (Pseudo code 4). The predicates for terminating and legitimate states are as follow:

$$\Big(\mathbf{c}[1] \ \rightarrow\ \exists j\,.\ (\mathbf{d}[j] \wedge \neg\mathbf{b}[j]) \rightarrow \forall i\,.\,(\mathbf{c}[i] \vee \mathbf{d}[i] \vee \mathbf{b}[i])\Big) \wedge \Big(\neg\mathbf{c}[1] \rightarrow \forall j \neq 1\,.\,(\mathbf{c}[j] \vee \mathbf{d}[j] \vee \mathbf{b}[j])\Big) \quad (TS)$$

$$\mathbf{c}[1] \rightarrow \forall j\,.\,(\mathbf{r}[j] \vee \mathbf{c}[j] \vee \mathbf{b}[j]) \quad (LS)$$

There is one state-based requirement for this problem:

$$\Box(\mathbf{c}[1] \wedge \exists i\,.\ (\mathbf{d}[i] \wedge \mathbf{c}[i] \wedge \neg\mathbf{b}[i])) \rightarrow \exists j\,.\ (\mathbf{r}[j] \wedge \neg\mathbf{b}[j]) \quad (\varphi_{\mathbf{safety}})$$

Algorithm 4 Byzantine Broadcast with Fail-Stop

1: Template variables

$$V^{\ell} = \{b, d, c\},$$
$$V^{s} = \{r\}, V = V^{\ell} \cup V^{s}:$$

b, d, c, r : **boolean**;
init $b = false, d = false, c = false, r = false$

2: Template guards G:

$$(\mathsf{id} = 1) \wedge \neg\mathbf{r}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \ \rightarrow \mathbf{r}[\mathsf{id}] := true;$$
$$\mathbf{r}[1] \ \wedge\ \neg\mathbf{b}[\mathsf{id}] \ \wedge\ \neg\mathbf{c}[\mathsf{id}] \wedge \neg\mathbf{r}[\mathsf{id}] \ \rightarrow \mathbf{r}[\mathsf{id}] := true;$$
$$\mathbf{r}[\mathsf{id}] \wedge \neg\mathbf{b}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \wedge \neg\mathbf{d}[\mathsf{id}] \ \rightarrow \mathbf{d}[\mathsf{id}] := true;$$

3: Faults F:

$$(\mathsf{id} = 1) \wedge \mathbf{r}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \ \rightarrow \mathbf{r}[\mathsf{id}] := false; \mathbf{c}[\mathsf{id}] = true;$$
$$(\forall k\,.\,\neg\mathbf{b}[k]) \wedge \neg\mathbf{c}[\mathsf{id}] \wedge (\mathsf{id} \neq 1) \ \rightarrow \mathbf{b}[\mathsf{id}] := true;$$
$$\mathbf{b}[\mathsf{id}] \wedge \neg\mathbf{c}[\mathsf{id}] \ \rightarrow \mathbf{d}[\mathsf{id}] := false/true; \mathbf{r}[\mathsf{id}] := false/true;$$
$$\forall k \neq 1\,.\,\neg\mathbf{c}[k] \ \rightarrow \mathbf{c}[\mathsf{id}] := true; \mathbf{r}[\mathsf{id}] := false;$$

Case Study	Byz. Faults	Crash Faults	Abstraction Level (l)	Number of States	Synthesis time (s)
(1,2) *Agreement*	1	0	1	852	1
	1	1	1	4746	11
(3) *Reliable broadcast*	1	2	1	924	—
	1	2	2	2106	—
	1	2	3	4192	8
(4) *Uniform broadcast*	1	2	1	924	—
	1	2	2	2106	—
	1	2	3	4192	15

TABLE I: Synthesis results

This specification corresponds to *uniform agreement* that states if a process (alive or crashed) has delivered the message, then every correct process should also deliver the message. Based on this safety specfication, if a process that has delivered the message crashes, and the sender has also crashed, then there should be at least one process that has received the message. There is also one transition-based specification for this problem similar to the previous case study (Eq. 3).

Our synthesizer was not successful in synthesis with abstraction levels 1 and 2. Setting the abstraction level to 3, we could synthesize a uniform reliable broadcast in 15 seconds.

VI. Conclusion

In this paper, we proposed an automated technique for parameterized synthesis of fault-tolerant distributed systems. Our idea is to use counter abstraction to make the state space finite and apply fixedpoint computations for synthesis. Our algorithm takes as input a parameterized fault-intolerant algorithm and a set of faults, and automatically synthesizes a fault-tolerant version of that algorithm. Using our algorithm, we could successfully synthesize fault-tolerant algorithms for well-known problems in distributed computing in less than 16 seconds. As for future work, we plan to combine our technique with the work in [21] and build a closed-loop synthesis method for parameterized guarded commands of distributed algorithms.

REFERENCES

[1] M. J. Fischer, N. A. Lynch, and M. S. Peterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 373–382, 1985.

[2] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder, "Parameterized model checking of fault-tolerant distributed algorithms by abstraction," in *FMCAD*, 2013, pp. 201–209.

[3] B. Aminof, S. Rubin, I. Stoilkovska, J. Widder, and F. Zuleger, "Parameterized model checking of synchronous distributed algorithms by abstraction," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 1–24.

[4] I. Konnov, H. Veith, and J. Widder, "Smt and por beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 85–102.

[5] ——, "On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability," *Information and Computation*, vol. 252, pp. 95–109, 2017.

[6] I. Konnov, M. Lazić, H. Veith, and J. Widder, "A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms," in *SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 719–734.

[7] I. Stoilkovska, I. Konnov, J. Widder, and F. Zuleger, "Verifying safety of synchronous fault-tolerant algorithms by bounded model checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 357–374.

[8] F. Faghih, B. Bonakdarpour, S. Tixuil, and S. Kulkarni, "Specification-based synthesis of distributed self-stabilizing protocols," *Logical Methods in Computer Science (LMCS)*, vol. 14, 2018.

[9] P. C. Attie, A. Arora, and E. A. Emerson, "Synthesis of fault-tolerant concurrent programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 1, pp. 125–185, 2004.

[10] S. Kulkarni, S. Tixeuil, B. Bonakdarpour, and F. Faghih, "Automated synthesis of distributed self-stabilizing protocols," *Logical Methods in Computer Science*, vol. 14, 2018.

[11] F. Faghih and B. Bonakdarpour, "Assess: A tool for automated synthesis of distributed self-stabilizing algorithms," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2017, pp. 219–233.

[12] B. Bonakdarpour, S. S. Kulkarni, and F. Abujarad, "Symbolic synthesis of masking fault-tolerant programs," *Springer Journal on Distributed Computing (DC)*, vol. 25, no. 1, pp. 83–108, March 2012.

[13] A. Gascón and A. Tiwari, "A synthesized algorithm for interactive consistency," in *NASA Formal Methods Symposium*. Springer, 2014, pp. 270–284.

[14] A. Girault and É. Rutten, "Automating the addition of fault tolerance with discrete controller synthesis," *Formal Methods in System Design (FMSD)*, vol. 35, no. 2, pp. 190–225, 2009.

[15] F. Faghih and B. Bonakdarpour, "Symbolic synthesis of timed models with strict 2-phase fault recovery," *IEEE Transactions on Dependable and Secure Systems (TDSC)*, vol. 15, no. 3, pp. 526–541, 2018.

[16] S. Jacobs and R. Bloem, "Parameterized synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 362–376.

[17] A. P. Klinkhamer and A. Ebnenasir, "Synthesizing parameterized self-stabilizing rings with constant-space processes," in *International Conference on Fundamentals of Software Engineering*. Springer, 2017, pp. 100–115.

[18] S. Außerlechner, S. Jacobs, and A. Khalimov, "Tight cutoffs for guarded protocols with fairness," in *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2016, pp. 476–494.

[19] N. Mirzaei, F. Faghih, S. Jacobs, and B. Bonakdarpour, "Parameterized synthesis of silent self-stabilizing protocols in symmetric rings," *Acta Informatica*, vol. 57, no. 1, pp. 271–304, 2020.

[20] N. Mirzaie, F. Faghih, S. Jacobs, and B. Bonakdarpour, "Parameterized synthesis of self-stabilizing protocols in symmetric rings," in *On Principles of Distributed Systems (OPODIS)*, 2018, pp. 1–17.

[21] M. Lazic, I. Konnov, J. Widder, and R. Bloem, "Synthesis of distributed algorithms with parameterized threshold guards," in *On Principles of Distributed Systems (OPODIS)*, 2017.

[22] A. Pnueli, J. Xu, and L. Zuck, "Liveness with (0, 1,∞)-counter abstraction," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 107–122.

[23] A. Arora and M. G. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, pp. 1015–1027, 1993.

[24] S. S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 2000, pp. 82–93.

[25] S. S. Kulkarni and A. Ebnenasir, "Enhancing the fault-tolerance of non-masking programs," *International Conference on Distributed Computing Systems*, 2003.

[26] S. S. Kulkarni, B. Bonakdarpour, and A. Ebnenasir, "Mechanical verification of automatic synthesis of fault-tolerant programs," in *International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 2004, pp. 36–52.

[27] B. Bonakdarpour and S. S. Kulkarni, "Revising distributed UNITY programs is NP-complete," in *OPODIS*, 2008, pp. 408–427.

[28] 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 preprint arXiv:1210.3846*, 2012.