On Verifying Fault Tolerance of Distributed Protocols

Dana Fisman[1,2], Orna Kupferman[1], and Yoad Lustig[1]

[1] School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel
[2] IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{danafi,orna,yoadl}@cs.huji.ac.il

Abstract. Distributed systems are composed of processes connected in some network. Distributed systems may suffer from faults: processes may stop, be interrupted, or be maliciously attacked. Fault-tolerant protocols are designed to be resistant to faults. Proving the resistance of protocols to faults is a very challenging problem, as it combines the parameterized setting that distributed systems are based-on, with the need to consider a hostile environment that produces the faults. Considering all the possible fault scenarios for a protocol is very difficult. Thus, reasoning about fault-tolerance protocols utterly needs formal methods.

In this paper we describe a framework for verifying the fault tolerance of (synchronous or asynchronous) distributed protocols. In addition to the description of the protocol and the desired behavior, the user provides the fault type (e.g., fail-stop, Byzantine) and its distribution (e.g., at most half of the processes are faulty). Our framework is based on augmenting the description of the configurations of the system by a mask describing which processes are faulty. We focus on regular model checking and show how it is possible to compile the input for the model-checking problem to one that takes the faults and their distribution into an account, and perform regular model-checking on the compiled input. We demonstrate the effectiveness of our framework and argue for its generality.

1 Introduction

Distributed systems are composed of processes connected in some network [25,30]. In a typical setting, the processes are isomorphic, in the sense that they execute the same protocol up to renaming. Thus, the systems are *parameterized* by the number of processes, and a protocol is correct if it is correct for any number of processes.

With the implementation of distributed protocols, it has been realized that the model of computation that basic protocols assume is often unrealistic. In reality, the processes and the communication between them may suffer from *faults*: messages may be omitted, processes may stop, may be interrupted, and may be maliciously attacked, causing them not to follow their protocol. For example, in the *fail-stop* fault, processes may fail before completing the execution of their code [29]. Even a simple task like broadcasting a message from a sender process to all other processes becomes difficult in the presence of such faults. Indeed, the sender may fail after sending the message to only a subset of the other processes, resulting in disagreement about the content of the message among processes that have not failed.

The realization of faults has led to the development of *fault-tolerant protocols*, which are designated to be resistant to faults. For example, for the broadcasting task, a protocol of n rounds (n is the number of processes) in which the sender broadcasts the message in the first round and every process that has received the message for the first time in the same round broadcasts it to all other processes in the next round, ensures that all processes that have not failed agree on the content of the message [14].

Proving the resistance of protocols to faults is a very challenging problem, as it combines the multi-process setting that distributed protocols are based-on, with the need to consider a hostile environment that produces the faults. Considering all the possible fault scenarios for a protocol is very difficult. This is reflected in the very complicated manual proofs that new protocols are accompanied with, and in the unfortunate fact that it is not rare that errors escape these manual proofs [12,7,19]. Thus, verification of fault-tolerance protocols utterly needs formal methods.

Current applications of model checking to reasoning about fault-tolerant distributed protocols are very elementary [5]. For example, [27] reports an error in the Byzantine self-stabilizing protocol of [13]. The error has been detected using the model checker SMV for the case $n = 4$. Likewise, a corrected version of the protocol has been proven correct in SMV for the case $n = 4$ [26]. While these works clearly demonstrate the necessity and effectiveness of model checking, there is no general methodology for reasoning about fault-tolerant protocols. Moreover, these works ignore the unbounded state-space that the parameterized setting involves; proving that a protocol is correct for the case $n = 4$ does not guarantee the protocol is correct for any number of processes. Thus, formal reasoning about distributed protocols, which is so utterly needed, requires the development and application of parameterized verification methods.

The parameterized setting is, in general, undecidable [4]. There has been extensive research in the last decade on finding settings for which the problem is decidable (c.f., [17]) and on developing methods that are sound but incomplete. Efforts in this direction include induction, network invariants, abstraction, and more [18,24,20].

A direction that has received a lot of attention is that of *regular model checking* [23,3]. In regular model checking, we describe configurations of the system as well as transitions between configurations by regular languages. In more details, in a *regular description of a protocol*, letters in the alphabet Σ describe states of the underlying processes, and a configuration of the system corresponds to a word in Σ^*. The protocol is then given by a regular language $I \subseteq \Sigma^*$ describing the possible initial configurations, and a regular language $R \subseteq (\Sigma \times \Sigma)^*$ describing the transition relation (a letter $[\sigma, \sigma']$ describes the current (σ) and next (σ') state of a certain process). For example, if each process may either have a token or not have it, then a letter in $\Sigma = \{N, T\}$ describes a state of each process, $I = T \cdot N^*$ describes a set of configurations in which only the leftmost process has the token, and $R = ([T, T] + [N, N] + [T, N] \cdot [N, T])^*$ describes a set of transitions in which processes either maintain their state or participate in a pass of a token from a process to its right. In this example all processes make a step simultaneously, thus they represent a synchronous distributed systems. However, we can also represent asynchronous systems using a regular description by coding a transition in which only one process can make a step at each time unit. It is sometimes more convenient to describe the protocol in a monadic second order logic over finite words (FMSO) [23], which is as expressive as regular expressions [11,16]. Then, a formula over Σ

describes the initial configurations, and a formula over $\Sigma \cup \Sigma'$ describes the transitions, with Σ' referring to the letters in the successor state.

A weakness of regular model checking is that not all protocols have a regular description. Moreover, even when a regular description exists, reasoning about it may diverge. The good news is that many interesting protocols do have a regular description. Also, various *acceleration*, *abstraction*, and *symmetry-reduction* techniques are successfully combined with regular model checking and lead the model-checking algorithm into termination [10,1,28,9,8]. Regular model checking has successfully been applied for the verification of a variety of protocols, including ones involving systems with queues, counters, and dynamic linked data structures [10,1]. In particular, termination of regular model checking is guaranteed for systems in which the set of reachable configurations is regular [22].

In this paper we suggest a methodology for reasoning about the fault tolerance of (synchronous or asynchronous) distributed protocols.[1] In addition to the description of the protocol, the user provides the following parameters:

1. *Fault type*: The user can specify the type of faults with which he wishes to challenge the protocol. We support faults like *fail-stop* (processes halt before completing the execution of the code), *Byzantine* (processes do not follow their code, and we also allow variants of Byzantine faults, like *omission* and *timing* faults), and *transient* (the state of the faulty processes is perturbed for a finite duration) faults. The methodology is compositional in the sense that the faults can be combined. For example, the user can check *self-stabilization* (resistance to transient faults) of a protocol in the presence of Byzantine faults.

2. *Fault distribution:* The user can specify the distribution of the faulty processes. The distribution is specified as a bound on the number of the sound/faulty processes, or on their ratio (e.g., a strict minority of the processes are faulty).[2] In fact, as explained shortly, we support all fault distributions that can be specified by a context-free language (CFG).

3. *Desired behavior:* The user specifies the desired property in LTL(FMSO) — an extension of LTL in which the propositional layer is replaced by a second-order layer describing the unbounded configurations [2]. We show how, using LTL(FMSO), the user can specify both properties of the global setting (e.g., all processes eventually agree on the content of the message) or properties that refer to the underlying processes (e.g., every sound process that tries to enter the critical section, eventually enters it).

Our methodology is based on augmenting the description of the configurations of the system by a *mask* describing which processes are faulty. We describe our methodology in the framework of regular model checking. Technically, we replace the alphabet Σ

[1] A different approach to reasoning about fault-tolerant systems is taken in [6]. There, following the classification of faults in [5], faults are modeled by transitions that perturb the state of the system. The problem studied in [6] is that of closed-system *synthesis*. Thus, this work is orthogonal to ours and, in particular, it does not address the parametric setting.

[2] Proving the correctness of a system, one is typically interested in an upper bound on the faulty processes and/or a lower bound on the sound processes. Refuting the correctness of a system, one is typically interested in an upper bound on the sound processes and/or a lower bound on the faulty processes.

by the alphabet $\Sigma \times \{\mathrm{S}, \mathrm{F}\}$, in which each letter describes not only the state of the corresponding process but also whether it is sound (S) or faulty (F). We then compile the languages $I \subseteq \Sigma^*$ of the initial configurations into a language $I' \subseteq (\Sigma \times \{\mathrm{S}, \mathrm{F}\})^*$, and compile the language $R \subseteq (\Sigma \times \Sigma)^*$ of transitions into a language $R' \subseteq ((\Sigma \times \{\mathrm{S}, \mathrm{F}\}) \times (\Sigma \times \{\mathrm{S}, \mathrm{F}\}))^*$. The type of the fault is reflected in the way faulty processes behave in I' and R'. The compilation is automatic and is done on top of the FMSO description of the underlying process. We can determine the distribution of the faults by restricting I' to configurations whose projection on $\{\mathrm{S}, \mathrm{F}\}$ belongs to a language that describes the desired distribution. Here, we may use either a regular language (say, for bounding the number of faulty processes by a constant) or a context-free language (say, for bounding the ratio of the faulty processes).[3]

We demonstrate the application of our methodology in a toy example of a token-ring-based mutual-exclusion protocol. Application of our methodology to a real example of the reliable broadcasting protocol of [14] can be found in the full version of the paper.

2 Preliminaries

2.1 Regular Description of a Protocol

A *regular description of a protocol* is a tuple $P = \langle \Sigma, I, R \rangle$, where Σ is an alphabet, each letter of which describes a possible state of one process. Intuitively, each configuration of a system composed of processes that follow the protocol P is a word $w \in \Sigma^*$. The length of w is the number of underlying processes, with the first letter describing the state of the first process, the second letter describing the state of the second process, and so on. Accordingly, $I \subseteq \Sigma^*$ is a regular language describing the initial configuration of the system for any number of processes, and $R \subseteq (\Sigma \times \Sigma)^*$ is a regular language describing its transition relation. Given two words over Σ of same length, say $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_n$ and $w' = \sigma'_1 \cdot \sigma'_2 \cdots \sigma'_n$, we use $[w, w'] \in R$ to abbreviate $[\sigma_1, \sigma'_1] \cdot [\sigma_2, \sigma'_2] \cdots [\sigma_n, \sigma'_n] \in R$. A *computation* of the system is a sequence $w_0, w_1, w_2, \ldots$ of words over Σ such that $w_0 \in I$, and for every $i \geq 0$, we have $[w_i, w_{i+1}] \in R$.

For a regular language $L \subseteq \Sigma^*$, let $pre_R(L) = \{w : \exists w' \in L \text{ such that } [w, w'] \in R\}$ and $post_R(L) = \{w : \exists w' \in L \text{ such that } [w', w] \in R\}$ be the pre- and post-images of L, respectively. We use pre^*_R and $post^*_R$ to denote the transitive closure of pre_R and $post_R$, respectively. Thus, if L describes a set of configurations, then $pre^*_R(L)$ is the set of configurations that can reach a configuration in L, and dually for $post^*_R(L)$.

Example 1. Consider the Token-Ring protocol described below. Each process has a boolean variable *token_is_mine* indicating whether it holds the token. The process may be in one of the three locations ℓ_0, ℓ_1, and ℓ_2. Location ℓ_2 is a critical section.

[3] The restriction of I' can be done after the fixed-point computation that model checking involves is completed. This enables us to proceed with both forward and backward model checking. This is also why we do not sacrifice decidability in the richer context-free setting.

Protocol 1. Token-Ring

```
boolean $token\_is\_mine$
repeat
    $\ell_0$ : if $token\_is\_mine$ then
    pass_token_to_right();
        goto $\{\ell_0, \ell_1\}$;
    $\ell_1$ : await $token\_is\_mine$;
    $\ell_2$ : critical;
until forever ;
```

A process that enters ℓ_2 exits it (and returns to ℓ_0) in the next transition. Location ℓ_1 is a trying section. A process in ℓ_1 waits for the token, and once it has it, it moves to the critical section in the next transition. Location ℓ_0 is the non-critical section. A process in ℓ_0 may either stay in ℓ_0 or proceed to ℓ_1. In addition, if the process has the token, it passes it to the process to its right.

We now present a regular description of the token-ring protocol. Let $\Sigma_{loc} = \{\ell_0, \ell_1, \ell_2\}$ and $\Sigma_{tok} = \{N, T\}$. A state of a process is a letter in $\Sigma = \Sigma_{loc} \times \Sigma_{tok}$, describing both the location of the process and the value of $token\_is\_mine$. For example, the letter $\langle \ell_2, T\rangle$ indicates a process in location ℓ_2 that has the token. For simplicity, we use the letters $0T, 1T, 2T, 0N, 1N$, and $2N$ to describe the letters in Σ.

The initial configuration of a system in which all processes are in location ℓ_0 and the token is owned by one[4] process is given by $0T \cdot 0N^*$. In order to describe the transition relation, we distinguish between three types of actions a process may be involved at during a transition. Each action corresponds to a letter $[\sigma, \sigma'] \in \Sigma \times \Sigma$, where σ describes the current state of the process and σ' describes the next state.

- The process does *not* pass or receive the token. These actions correspond to the letters $S_N = \{[0N, 0N], [0N, 1N], [1N, 1N], [2N, 0N], [1T, 2T], [2T, 0T]\}$.
- The process has the token in location ℓ_0, in which case it *passes* it to the right. These actions correspond to the letters $S_P = \{[0T, 0N], [0T, 1N]\}$.
- The process does not have the token and *receives* it from the left. These actions correspond to the letters $S_R = \{[0N, 0T], [0N, 1T], [1N, 1T], [2N, 0T]\}$.

The transition function R then allows all processes to proceed with actions in S_N and allows adjacent processes to proceed with S_P (the left process) and S_R (the right process) simultaneously. Accordingly,[5] $R = (S_N + S_P \cdot S_R)^* + (S_R \cdot (S_N + S_P \cdot S_R)^* \cdot S_P)$.

Note that R indeed reflects the protocol. In particular, the transition function is defined also for processes that are in the critical section without the token, and for configurations in which there is more than a single token. Indeed, for different initial configurations, such transitions may be taken.[6]

[4] Since we use words to model a ring architecture, we arbitrarily set the token owner to be the leftmost process. Nevertheless, the transitions are defined so that the so called rightmost process has this leftmost process as its neighbor to the right.

[5] The second term corresponds to the rightmost process closing the ring.

[6] We note that the common description of token-passing protocols in the regular model-checking literature allows only a single pass to take place in a transition. In a transition in our model, all processes proceed together and any number of tokens may be passed simultaneously (yet a process cannot receive and pass a token at the same transition).

2.2 FMSO and LTL(FMSO)

In Section 2.1, we used regular expressions in order to specify configurations of the distributed system. In this section we present *finitary monadic second order logic* (FMSO) [11,16,31] – an alternative formalism for specifying regular languages. Describing configurations of a parameterized system by a monadic second order logic is suggested in [23], where the logic used is FS1S. A similar direction was taken in [2], where LTL is augmented with MSO. We choose to work with FMSO, which unifies both approaches.

FMSO formulas are interpreted over finite words over Σ. Formulas are defined with respect to a set $\mathbb{F}$ of first-order variables ranging over positions in the word, and a set $\mathbb{S}$ of second-order variables ranging over sets of positions in the word. Let us emphasize to readers who are familiar with temporal logic that the variables in $\mathbb{F}$ and $\mathbb{S}$ do not point to points in time (or sets of such points) — an FMSO formula describes a configuration of the system at a single time point, and the variables in $\mathbb{F}$ and $\mathbb{S}$ point to positions (or sets of positions) in the configuration, namely to identity of processes in the parameterized system.

In our application, the alphabet Σ describes a state of a process, and a word of length n describes a configuration consisting of n processes. Typically, $\Sigma = \Sigma_1 \times \cdots \times \Sigma_k$ is the product of *underlying alphabets*, each describing a propositional aspect of a state of a process. For example, in Example 1, we had $\Sigma = \Sigma_{loc} \times \Sigma_{tok}$. We refer to the set $\{\Sigma_1, \ldots, \Sigma_k\}$ as the *signature* of Σ and refer to the set $\Sigma_1 \cup \cdots \cup \Sigma_k$ as the set of *underlying letters*. An advantage of FMSO is that it enables convenient reference to the underlying letters. Given a word in Σ^*, a position term $p \in \mathbb{F}$ points to a letter in Σ, and preceding it by an underlying alphabet points to an underlying letter. For example, the letter term $\Sigma_{tok}[p]$ is evaluated to the status of the token of the process in position p. Thus, if p is evaluated to 3, then the letter term $\Sigma_{tok}[p]$, when interpreted over the word $\langle \ell_0, T\rangle, \langle \ell_0, N\rangle, \langle \ell_0, N\rangle, \langle \ell_0, N\rangle$, is evaluated to N – the projection on $\{N, T\}$ of the third letter. We now describe the syntax and the semantics of FMSO formally.

Syntax. Let $\mathbb{F}, \mathbb{S}$ be sets of variables as above, and let $\Sigma = \Sigma_1 \times \cdots \times \Sigma_k$. Terms and formulas of FMSO are defined inductively as follows.

- A *position (first order) term* is of the form $0, i, p \oplus 1$, or $p \ominus 1$, for $i \in \mathbb{F}$ and a position term p.
- A *letter term* is of the form τ or $x[p]$, for $\tau \in \Sigma_1 \cup \cdots \cup \Sigma_k$, a position term p, and x an underlying alphabet in $\{\Sigma_1, \ldots, \Sigma_k\}$.
- A *formula* is of the form $a_1 = a_2, p_1 \leq p_2, I_1 \subseteq I_2, p \in I, \neg\varphi, \varphi \vee \psi, \exists i \varphi$, or $\exists I \varphi$, for letter terms a_1 and a_2, position terms p, p_1 and p_2, formulas φ and ψ, $i \in \mathbb{F}$, and $I_1, I_2, I \in \mathbb{S}$.

Writing formulas, we use the standard abbreviations $=, <, \neq, \wedge, \rightarrow$, and $\forall$. In addition, we use $\Sigma[p] = \langle \sigma_1, \ldots, \sigma_k \rangle$ as an abbreviation for $\Sigma_1[p] = \sigma_1 \wedge \ldots \wedge \Sigma_k[p] = \sigma_k$. An FMSO formula is *closed* if all the occurrences of variables in $\mathbb{F}$ and $\mathbb{S}$ are in a scope of a quantifier.

Semantics. For an integer $n \in \mathbb{N}$, let $\mathbb{Z}_n$ denote the set $\{0, 1, \ldots, n-1\}$. We define the semantics of an FMSO formula with respect to a tuple $\mathcal{I} = \langle n, \mathcal{I}_F, \mathcal{I}_S, \mathcal{I}_\Sigma \rangle$, where $n \in \mathbb{N}$ is the length of the word that $\mathcal{I}$ models, $\mathcal{I}_F : \mathbb{F} \to \mathbb{Z}_n$ assigns the first-order variables with locations in $\mathbb{Z}_n$, $\mathcal{I}_S : \mathbb{S} \to 2^{\mathbb{Z}_n}$ assigns the second-order variable with subsets of $\mathbb{Z}_n$, and $\mathcal{I}_\Sigma : \mathbb{Z}_n \to \Sigma$ is the word that $\mathcal{I}$ models. Given a letter $\sigma \in \Sigma$ and an underlying alphabet $x \in \{\Sigma_1, \ldots, \Sigma_k\}$, we denote the projection of σ on x by $\sigma_{|x}$.

Position terms are evaluated with respect to n and $\mathcal{I}_F$ as follows: $[\![0]\!] = 0$, $[\![i]\!] = \mathcal{I}_F(i)$, $[\![p \oplus 1]\!] = ([\![p]\!] + 1) \bmod n$, and $[\![p \ominus 1]\!] = ([\![p]\!] - 1) \bmod n$. Letter terms are evaluated with respect to $\mathcal{I}_\Sigma$ as follows: $[\![\tau]\!] = \tau$ and $[\![x[p]]\!] = \mathcal{I}_\Sigma(p)_{|x}$. Satisfaction of formulas is defined as follows:

$$\begin{array}{llll}
\mathcal{I} \models a_1 = a_2 & \text{iff } [\![a_1]\!] = [\![a_2]\!] & \mathcal{I} \models \neg\varphi & \text{iff } \mathcal{I} \not\models \varphi \\
\mathcal{I} \models p_1 \leq p_2 & \text{iff } [\![p_1]\!] \leq [\![p_2]\!] & \mathcal{I} \models \varphi \vee \psi & \text{iff } \mathcal{I} \models \varphi \text{ or } \mathcal{I} \models \psi \\
\mathcal{I} \models I_1 \subseteq I_2 & \text{iff } \mathcal{I}_S(I_1) \subseteq \mathcal{I}_S(I_2) & \mathcal{I} \models \exists i\varphi & \text{iff } \exists m \in \mathbb{Z}_n \text{ s.t. } \mathcal{I}[i \mapsto m] \models \varphi \\
\mathcal{I} \models p_1 \in I & \text{iff } [\![p_1]\!] \in \mathcal{I}_S(I) & \mathcal{I} \models \exists I\varphi & \text{iff } \exists S \subseteq \mathbb{Z}_n \text{ s.t. } \mathcal{I}[I \mapsto S] \models \varphi,
\end{array}$$

where $\mathcal{I}[i \mapsto m]$ is obtained from $\mathcal{I}$ by letting $\mathcal{I}_F(i)$ be m, and similarly for $\mathcal{I}[I \mapsto S]$.

LTL(FMSO). The logic LTL is traditionally defined over computations in which each point in time can be characterized by a propositional formula. In the parameterized setting, each point in time is an unbounded configuration, and can be characterized by an FMSO formula. The logic LTL(FMSO) is an extension of LTL in which the propositional layer is replaced by FMSO. Thus, the FMSO formulas are used to describe a configuration of the computation at a given instance of time, and the LTL operators are used to reason about the on-going behavior of the system. The internal FMSO formulas may contain a free variable whose quantification is external to the temporal operators. A regular model-checking procedure for LTL(FMSO) is described in [2]. The syntax and semantics of LTL(FMSO) are given in the full version of the paper. Here, we give some examples.

Example 2. Consider the token-ring protocol given in Example 1. We use LTL(FMSO) in order to specify its desired properties:

– Mutual exclusion (there is always at most one process in the critical section): $\Box(\forall i, j : (i \neq j) \to \neg(\Sigma_{loc}[i] = \ell_2 \wedge \Sigma_{loc}[j] = \ell_2))$.
– Non-starvation (whenever a process tries to enter the critical section, it eventually does): $\Box\forall i : (\Sigma_{loc}[i] = \ell_1 \to \Diamond(\Sigma_{loc}[i] = \ell_2))$.

2.3 An FMSO-Based Description of a Protocol

In this section we explain how FMSO can be used to define protocols and the parameterized system they induce. A similar description appears in [23].[7] There, however, formulas describe the parameterized system whereas here formulas describe an underlying process parameterized by its identity. An FMSO description of the parameterized system is then automatically derived from the description of its underlying process. The ability to describe a single process is fundamental to our method since the input to our application carries information on how a fault affects a single process rather than how it affects the parameterized system (In Remark 5, we elaborate on the significance of this ability further).

A *protocol parameterized by* $i \in \mathbb{F}$ is a tuple $P[i] := \langle \Sigma, \Theta[i], \Delta[i] \rangle$, where $\Sigma = \Sigma_1 \times \cdots \times \Sigma_k$ is the alphabet, $\Theta[i]$ is an FMSO formula that specifies the initial state of

[7] The monadic second order used in [23] is FS1S (rather than FMSO).

the process i, and $\Delta[i]$ is an FMSO formula over $\Sigma \cup \Sigma'$ where Σ' is a primed version of the alphabet Σ. The formula $\Delta[i]$ relates the current configuration (over the alphabet Σ) with the successor configuration (over the alphabet Σ'). The only free variable in the formulas $\Theta[i]$ and $\Delta[i]$ is i. Note that the formulas may refer to the current as well as the successor state of other processes, but this reference is either relativized by i (say, to $i \oplus 1$) or is universal or existential.[8]

The *parameterized system induced by* $P[i]$ is given by $P = \langle \Sigma, \Theta, \Delta \rangle$, where the initial configuration is $\Theta = \forall i \Theta[i]$, and the transition relation is $\Delta = \forall i \Delta[i]$. Thus, as expected, each process starts in an initial state, and in each point in time, all processes simultaneously proceed according to the protocol. Note that, as in the regular description of a protocol, this does not prevent us from describing asynchronous systems. Asynchronous systems can be modeled by adding to $\Delta[i]$ a disjunct of the form $\Sigma[i]{=}\Sigma'[i]$ that allows a process to remain in its state.

Example 3. Consider the Token-Ring protocol discussed in Example 1. We can provide an FMSO description of the protocol $P[i] := \langle \Sigma, \Theta[i], \Delta[i] \rangle$ as follows. The alphabet is $\Sigma = \Sigma_{loc} \times \Sigma_{tok}$. The initial state stipulates that the control location is ℓ_0 and the token is owned by the process iff its identity is 1. Thus $\Theta[i] := (\Sigma_{loc}[i]{=}\ell_0) \wedge ((i{=}1 \wedge \Sigma_{tok}[i]{=}T) \vee (i{\neq}1 \wedge \Sigma_{tok}[i]{=}N))$. Following the regular description given in Example 1, we define three transitions $\delta_N[i]$, $\delta_P[i]$, and $\delta_R[i]$, where $\delta_N[i]$ corresponds to the case where the process does not pass or receive the token, $\delta_P[i]$ corresponds to the case where the process passes the token and $\delta_R[i]$ corresponds to the case where the process receives the token. Since a token may pass from a process to its right neighbor, the overall transition relation is then $\Delta[i] := \delta_N[i] \vee (\delta_P[i] \wedge \delta_R[i \oplus 1]) \vee (\delta_R[i] \wedge \delta_P[i \ominus 1])$.

The transitions $\delta_N[i]$, $\delta_P[i]$, and $\delta_R[i]$ are defined as follows:

- $\delta_N[i] := (\Sigma_{tok}[i]{=}\Sigma'_{tok}[i]) \wedge (\Sigma_{loc}[i]{=}\ell_0 \rightarrow (\Sigma'_{loc}[i]{=}\ell_0 \vee \Sigma'_{loc}[i]{=}\ell_1)) \wedge (\Sigma[i]{=}\langle \ell_1, N \rangle \rightarrow \Sigma'[i]{=}\langle \ell_1, N \rangle) \wedge (\Sigma[i]{=}\langle \ell_1, T \rangle \rightarrow \Sigma'[i]{=}\langle \ell_2, T \rangle) \wedge (\Sigma_{loc}[i]{=}\ell_2 \rightarrow \Sigma'_{loc}[i]{=}\ell_0)$.
- $\delta_P[i] := (\Sigma_{tok}[i]{=}T \wedge \Sigma'_{tok}[i]{=}N) \wedge (\Sigma_{loc}[i]{=}\ell_0 \wedge (\Sigma'_{loc}[i]{=}\ell_0 \vee \Sigma'_{loc}[i]{=}\ell_1))$.
- $\delta_R[i] := (\Sigma_{tok}[i]{=}N \wedge \Sigma'_{tok}[i]{=}T) \wedge (\Sigma_{loc}[i]{=}\ell_0 \rightarrow (\Sigma'_{loc}[i]{=}\ell_0 \vee \Sigma'_{loc}[i]{=}\ell_1)) \wedge (\Sigma_{loc}[i]{=}\ell_1 \rightarrow \Sigma'_{loc}[i]{=}\ell_1) \wedge (\Sigma_{loc}[i]{=}\ell_2 \rightarrow \Sigma'_{loc}[i]{=}\ell_0)$.

3 Verifying Resistance to Faults

In this section we describe our methodology for verifying the resistance of distributed protocols to faults. The idea behind our methodology is as follows.

- Recall that each process is defined with respect to a set of underlying alphabets. We add to this set the underlying alphabet $\Sigma_f = \{\text{S}, \text{F}\}$. Doing so, each process i may be either sound ($\Sigma_f[i] = \text{S}$) or faulty ($\Sigma_f[i] = \text{F}$).

[8] The ability of process i to refer to other processes may seem to give it a power to force another process into doing something. However, in the induced parameterized system all processes take a transition simultaneously. Thus, there should be an agreement between what the other process does and what process i stipulates it does.

- Given a protocol $P[i]$ parameterized by $i \in \mathbb{F}$, we automatically modify $P[i]$ to include also transitions that correspond to a faulty behavior. The modification depends on the type of fault, and is described in Section 3.2. A process follows the new transitions iff it is faulty. Transitions may not change the classification to faulty and sound.[9]
- Given the modified protocol, we (automatically, see Section 2.3) generate from it a parameterized system. Note that each of the processes in the parameterized system may be either faulty or sound, and that this classification is indicated in Σ_f. By translating the FMSO formulas to regular expressions, we obtain a regular description $\widetilde{P} = \langle \Sigma \times \Sigma_f, I, R \rangle$ of the system. For some types of faults, we need to exclude from $\widetilde{P}$ computations that do not satisfy some fairness conditions. Rather than augmenting $\widetilde{P}$ with a fairness constraint, we associate with it an LTL(FMSO) formula ψ_{fair} that we later use as an assumption in the specification.[10]
- Given a fault distribution in terms of an upper/lower bound on the faulty/sound processes or an upper/lower bound on the ratio between the faulty and sound processes, we translate it into a CFG language $D \subseteq \Sigma_f^*$. A configuration of $\widetilde{P}$ agrees with the fault distribution if its projection on Σ_f is in D. The translation is automatic (see Section 3.3). The user may also describe D directly. For a language $L \subseteq (\Sigma \times \Sigma_f)^*$ and a language $D \subseteq \Sigma_f^*$, let $agree(L, D)$ denote the subset of L whose projection on Σ_f agrees with D. Formally, $[\sigma_1, \sigma'_1] \cdots [\sigma_n, \sigma'_n] \in agree(L, D)$ iff $[\sigma_1, \sigma'_1] \cdots [\sigma_n, \sigma'_n] \in L$ and $\sigma'_1 \cdots \sigma'_n \in D$.
- It is left to check $\widetilde{P}$ with fault distribution D with respect to the desired LTL(FMSO) property ψ. We proceed with the regular model-checking algorithm of [2], applied to $\psi_{fair} \rightarrow \psi$. Whenever a computation of the algorithm refers to the language I of initial configurations, we refer instead to $agree(I, D)$. It is possible to restrict I to configurations that agree with D at various steps in the model-checking procedure. Also, restricting I can be replaced by restricting fixed-points calculated during the computation. As detailed in Section 3.4, this flexibility has helpful practical implication.

Remark 1. While the methodology is presented for the general parametric setting, its idea can be applied also for a bounded finite number of processes. In particular, it is easy to adapt existing BDD based model checkers to apply for this case. Needless to say, some simple technical updates must be made, such as replacing FMSO with the model-checker language, and providing the fault distribution in a way suitable for BDD. Note also that when the number of processes is bounded but big, the parametric setting may still be advantageous.

We now provide the details of our methodology, starting by reviewing types of faults.

3.1 Types of Faults

The theory of fault-tolerant distributed systems studies a large variety of types of faults. We consider here the most common types. As we explain in Section 3.2, our method is versatile and one should be able to apply it to more types.

[9] As we show in Section 3.2 this does not prohibit us from modeling *fail-stop* and *transient* failures.

[10] One could also consider protocols with fairness constraints [28]. We found the description via LTL(FMSO) simpler.

• **Fail-stop.** A process that suffers from a *fail-stop* failure halts before the termination of its protocol. Such a process has a well-defined failure-mode operating characteristics, and indeed the idea behind fail-stop faults is to minimize the effect of failures – the faulty process halts in response to a failure, and it often does so in a detectable manner and before the effect of the failure becomes visible [29].

• **Byzantine.** In general, a *Byzantine* process is not committed to the protocol. Thus, it can take arbitrary transitions, changing its state and the values of variables it shares. The fact the process is Byzantine is undetectable. Byzantine faults are the most general type of faults and model a wide variety of problems ranging from hardware failures (causing unexpected system behavior) to malicious attack of hackers on the system. One often consider variants of Byzantine faults, like *timing faults* (the process does follow the protocol, but there are arbitrary delays between the execution of successive statements) and *omission faults* (the messages sent by and/or to the process do not get to their destination. The process might or might not be aware of the fact that the transmission went wrong).

• **Transient.** A *transient* fault occurs when a process suffers from a temporal failure, say part of its memory is corrupted. Technically, transient faults are similar to Byzantine faults, only that the duration of the Byzantine behavior is bounded. In addition, the fault may be restricted to specific elements of the process (memory, clock, etc.). Protocols that tolerant transient faults are often termed *self-stabilizing*, as they recover from faults in the prefix of the computation.

3.2 Generating the Faulty Protocol

Let $P[i] := \langle \Sigma, \Theta[i], \Delta[i] \rangle$ be a protocol parameterized by position variable i. For each type of fault discussed in section 3.1, we show how to construct a process $\widetilde{P}[i] := \langle \widetilde{\Sigma}, \widetilde{\Theta}[i], \widetilde{\Delta}[i] \rangle$ in which the process may be either sound or faulty. In the latter case, the process may exhibit a faulty behavior of the corresponding type.

For all types of faults, the signature of $\widetilde{\Sigma}$ consists of the signature of Σ and contains, in addition, the underlying alphabet $\Sigma_f = \{\textsc{s}, \textsc{f}\}$ (and possibly more underlying alphabets, according to the specific fault). Recall that the classification of processes to sound and faulty may not change. Accordingly, for all types of faults, the transition formula $\widetilde{\Delta}[i]$ is of the form $\Delta_f[i] \wedge (\Sigma_f[i] = \Sigma'_f[i])$, where $\Delta_f[i]$ is a modification of $\Delta[i]$ that depends on the specific fault. Below we describe the compilation of $\Delta[i]$ into $\Delta_f[i]$ for the various faults. Also, for some faults, we also generate an LTL(FMSO) formula ψ_{fair} that serves as a fairness condition for the faulty system. Unless we state differently, $\psi_{fair} =$ **true**, thus no fairness is required.

Fail-stop faults. Recall that Σ_f classifies the processes to sound and faulty. In the fail-stop fault, the faulty processes start their execution as sound processes, but may halt before the completion of the protocol. In order to model fail-stop faults, we add to the signature the underlying alphabet $\Sigma_t = \{\textsc{a}, \textsc{h}\}$, which indicates whether a faulty process is still alive (A) or has already halted (H). Sound processes and faulty, yet alive, processes should satisfy the original initial formula, thus $\widetilde{\Theta}[i] := (\Sigma_f[i] = \textsc{s} \vee \Sigma_t[i] = \textsc{a}) \rightarrow \Theta[i]$. The transition formula $\Delta_f[i]$ makes sure that (1) only a faulty process may halt (2) once a process halts, it cannot become alive, (3) the state of a process that halts does not change, and (4) processes that are sound or alive respect $\Delta[i]$. Formally, $\Delta_f[i] = [(\Sigma_t[i]{=}\textsc{h}) \rightarrow (\Sigma_f[i]{=}\textsc{f} \wedge \Sigma_t'[i]{=}\textsc{h})] \wedge [(\Sigma_t[i]{=}\textsc{h}) \rightarrow \Sigma[i]{=}\Sigma'[i]] \wedge [(\Sigma_t[i]{=}\textsc{a}) \rightarrow \Delta[i]]$.

Remark 2. Recall that fail-stop faults are detectable. Detectability can be modeled by making Σ_t observable to the other processes (either by putting it in a shared memory, or by letting the failing processes broadcast a failure notification before they halt). Thus, the original protocol, which is likely to be designed towards fail-stop faults, already has Σ_t in its signature, and the transitions in $\Delta[i]$ may refer to it.

Byzantine faults. Under a Byzantine failure, no assumption is made on the behavior of a faulty processes. A Byzantine process may start in an arbitrary configuration (which may or may not be valid for a sound process) and in each time unit it can transit to any other (valid/invalid) configuration . Accordingly, in $\widetilde{P}[i]$, the requirement to respect $\Theta[i]$ and $\Delta[i]$ is restricted to sound processes. Formally, $\widetilde{\Theta}[i] := (\Sigma_f[i]{=}\textsc{s}) \rightarrow \Theta[i]$ and $\Delta_f[i] := (\Sigma_f[i]{=}\textsc{s}) \rightarrow \Delta[i]$. In the full version of the paper we expand on **timing** and **omission faults**.

Transient faults. A process i affected by a transient fault need not respect $\Theta[i]$ and $\Delta[i]$. Unlike a Byzantine fault, however, the duration of the fault is finite. Thus, at some point, the process recovers and proceeds (from the arbitrary state it has reached in its perturbed behavior) according to Δ_i. In order to model transient faults, we add to the signature the underlying alphabet $\Sigma_t = \{\textsc{p}, \textsc{r}\}$, which indicates whether a faulty process is still perturbing (P) or has already recovered (R). Only faulty processes may perturb, and perturbed processes need not satisfy the initial formula.[11] Thus, $\widetilde{\Theta}[i] := (\Sigma_t[i]{=}\textsc{p} \rightarrow \Sigma_f[i]{=}\textsc{f}) \wedge (\Sigma_f[i]{=}\textsc{s} \rightarrow \Sigma_t[i]{=}\textsc{r}) \wedge (\Sigma_t[i]{=}\textsc{r} \rightarrow \Theta[i])$. In addition, perturbed processes need not satisfy the transition formula, and a recovered process cannot perturb again. Thus, $\Delta_f[i] := (\Sigma_t[i]{=}\textsc{r}) \rightarrow (\Delta[i] \wedge \Sigma_t'[i]{=}\textsc{r})$. Finally, to ensure that a process can perturb only during a finite prefix of the computation, we add the assumption formula $\psi_{fair} = \forall i(\Diamond\Box\, \Sigma_t[i]{=}\textsc{r})$.

Remark 3. Transient faults are often associated with specific components of the process. For example, it may be known that certain areas in the memory of the protocol may be temporarily corrupted. Accordingly, rather than letting the affected processes ignore $\Theta[i]$ and $\Delta[i]$, we let them satisfy the projection of $\Theta[i]$ and $\Delta[i]$ on the underlying alphabets in the signature that have not been affected.

Remark 4. An approach that is taken in the distributed-algorithm community is to reason about the self-stabilization of a protocol by reasoning about the protocol when starting from an arbitrary initial configuration (or, per Remark 3, from a set of allowed initial configurations that extends the original set). Such a reasoning can be easily done in our model by leaving $P[i]$ as is, except for $\Theta[i]$.

[11] We could give up Σ_t and model a recovery by modifying the F indication in Σ_f to S. The reason we do use Σ_t is practical: as we explain in Section 3.4, by keeping F and S fixed, we can sample the fault distribution at any time in the computation, which enables us to proceed with both forward and backward model checking.

Remark 5. It is easy to see that the computation of faulty processes need not respect the original protocol. Note, however, that sound processes may also follow computations that were not possible for them in the original protocol although they are obeying the protocol. For example, if the transition of process i is of the form $\alpha \vee (\Sigma[i \oplus 1]{=}\sigma \wedge \alpha')$, for some formulas α and α', and process $i+1$, when respecting the protocol, never satisfies $\Sigma[i \oplus 1]{=}\sigma$, then process i always proceeds with α. In a faulty system, however, process $i+1$ may satisfy $\Sigma[i \oplus 1] = \sigma$, letting process i, which is sound, to proceed with either α or α'. By compiling $P[i]$ rather than the parameterized system P we make sure that such scenarios do not escape the resulting faulty parameterized system. Another reason to compile underlying processes is practical: one of the heuristics that are applied to regular-model checking is symmetry reduction. Keeping the protocol of all (either sound or faulty) processes identical, reasoning about the compiled system can apply these reductions.

3.3 Handling Fault Distributions

The specification of fault-tolerance includes assumptions about the distribution of faults (e.g., a strict minority of the processes are faulty). We model a fault distribution by a language over $\{\text{S}, \text{F}\}$. To ease the work of the specification engineer, we suggest a simple and readable formalism in which common distributions can be specified. Formally, a *distribution bound* is a word $\gamma \in \{U, L\} \times \{\text{S}, \text{F}\} \times (\mathbb{N} \cup (\mathbb{N} \times \mathbb{N}))$. The first letter indicates whether γ imposes an upper (U) or lower (L) bound, the second letter indicates whether the bound refers to the sound or faulty processes, and the third indicates whether it is a constant bound $k \in \mathbb{N}$ or a ratio bound $\frac{k_1}{k_2}$, for $k_1, k_2 \in \mathbb{N}$. For example, $\gamma = \langle U, \text{F}, k \rangle$ (resp. $\langle L, \text{S}, k \rangle$) checks tolerance for a parameterized system with at most k faulty (at least k sound) processes, and $\gamma = \langle U, \text{F}, k_1, k_2 \rangle$ checks tolerance for a parameterized system in which at most $\frac{k_1}{k_2}$ of the processes are faulty. Given a distribution bound, we generate a language over $\{\text{S}, \text{F}\}$ that describes it. For a word w over $\{\text{S}, \text{F}\}$ and a letter $\sigma \in \{\text{S}, \text{F}\}$, let $\#(\sigma, w)$ denote the number of occurrences of σ in w. Then (other bounds are isomorphic to the ones below),

- $\langle U, \text{F}, k \rangle$ induces the regular language $\text{S}^* \cdot ((\text{F} + \text{S}) \cdot \text{S}^*)^k$
- $\langle L, \text{S}, k \rangle$ induces the regular language $\text{F}^* \cdot (\text{S} \cdot \text{F}^*)^k \cdot (\text{F} + \text{S})^*$
- $\langle U, \text{F}, k_1, k_2 \rangle$ induces the context-free language $\{w \mid k_1\#(\text{F}, w) \leq (k_2 - k_1)\#(\text{S}, w)\}$.

The user may also provide the distribution language directly, thus describing richer types of distributions, like $(\text{S} + \text{F})^* \cdot \text{F}^k \cdot (\text{S} + \text{F})^*$ (there exists a neighborhood of k faulty processes), $(\text{S}^{k-1} \cdot (\text{S} + \text{F}))^*$ (only every other k processes may be faulty), etc. Such distribution languages are particularly appropriate in architectures like rings, where the position of the process in the word describing the configuration is important. Also, a conjunction of bounds may be obtained by intersecting the corresponding (at most one context-free) languages.

We are now ready to model check the system with the faults according to the fault distribution in D. We first formalize properties of the compilation that are useful in the model-checking procedure. For simplicity, we assume that the alphabet of the compiled system is $\Sigma \times \Sigma_f$ (when its alphabet contains additional underlying alphabets, we project them in an existential manner). Theorem 1 below states that it is possible to augment the description of a protocol by a mask describing the faulty systems, and that reasoning about the augmented description can be done in both a backward and forward manner. The correctness of the theorem follows from the fact that the Σ_f component of the augmented protocol, which describes the mask, is fixed throughout the execution of the protocol. Let $w = \sigma_1 \cdot \sigma_2 \cdots \sigma_n$ and $w' = \sigma'_1 \cdot \sigma'_2 \cdots \sigma'_n$ be two words of the same length over the alphabets Σ and Σ', respectively. We use $w \otimes w'$ to denote the word over $\Sigma \times \Sigma'$ obtained by merging the letters of w and w'. That is, $w \otimes w' = (\sigma_1, \sigma'_1) \cdot (\sigma_2, \sigma'_2) \cdots (\sigma_n, \sigma'_n)$.

Theorem 1. *Consider an* FMSO *description $P[i]$ over Σ of a protocol parameterized by position variable i, a type β of a fault, and a distribution language $D \subseteq \Sigma_f^*$. Let P be the parameterized system induced by $P[i]$ and let $\widetilde{P}$ and ψ_{fair} be the parameterized system over $\Sigma \times \Sigma_f$ to which $P[i]$ is compiled.*

1. *$w_0, w_1, w_2 \ldots$ is a computation of the system P when suffering from a fault of type β with fault distribution D iff there is a word $d \in D$ such that $w_0 \otimes d, w_1 \otimes d, w_2 \otimes d \ldots$ is a computation of $\widetilde{P}$ that satisfies ψ_{fair}.*
2. *Let $\widetilde{R}$ be the transition relation of $\widetilde{P}$. For a set $S \subseteq (\Sigma \times \Sigma_f)^*$, we have $pre^*_{\widetilde{R}}(agree(S, D)) = agree(pre^*_{\widetilde{R}}(S), D)$ and $post^*_{\widetilde{R}}(agree(S, D)) = agree(post^*_{\widetilde{R}}(S), D)$.*

3.4 Model Checking the Faulty System

We now describe how we adjust the LTL(FMSO) regular model-checking algorithm of [2] to consider the distribution language. Let us first review the procedure in [2].

Given an LTL(FMSO) specification ψ, it is possible to extend the translation of LTL formulas to Büchi automata [32] and generate from ψ a Büchi transducer $\mathcal{A}_\psi$ (automaton over $\Sigma \times \Sigma$, in the terminology of [2]) that accepts exactly all the models of ψ. The transducer consists of three regular languages: initial configurations $I_\psi \subseteq \Sigma^*$, transitions $R_\psi \subseteq (\Sigma \times \Sigma)^*$, and acceptance condition $\alpha \subseteq \Sigma^*$. Let $\mathcal{A}_{\neg\psi} = \langle \Sigma, I_{\neg\psi}, R_{\neg\psi}, \alpha_{\neg\psi} \rangle$ be the Büchi transducer for an LTL(FMSO) formula $\neg\psi$. A parameterized system $P = \langle \Sigma, I, R \rangle$ then violates ψ if the product of P with $\mathcal{A}_{\neg\psi}$, namely the Büchi transducer $\langle \Sigma, I \cap I_{\neg\psi}, R \cap R_{\neg\psi}, \alpha_{\neg\psi} \rangle$, has a fair computation. It is shown in [28,2] how bad-cycle detection algorithms can be lifted to the regular setting.

Let $\mathcal{S} = \langle \Sigma \times \Sigma_f, I, R, \alpha \rangle$ be the product of the regular description of the faulty protocol with a Büchi transducer $\mathcal{A}_{\neg(\psi_{fair} \to \psi)}$ for $\neg(\psi_{fair} \to \psi)$, and let $D \subseteq \Sigma_f^*$ be the distribution language with respect to which the protocol is checked. By Theorem 1 (1), the system P tolerates the fault with distribution D iff $\mathcal{S}$ does not contain a computation that visits α infinitely often and whose projection on Σ_f is in D.

Thus, searching for bad cycles in $\mathcal{S}$, we should restrict attention to computations whose projection on Σ_f is in D. By Theorem 1 (2), we can sample the projection of the computation on Σ_f at any point, and can also do it after the computations of fixed-points converge. Accordingly, for forward model checking, we can start with I and restrict to computations that agree with D after the calculations of $post^*$. Likewise, for backward model checking, we can start with α and restrict to computations that agree with D

after the calculations of pre^*. Note that, as observed in [21], it is possible to conduct backward/forward model checking in which the last step uses a context-free rather than a regular language. This is possible due to the fact that context free languages are closed under intersection with regular languages and the emptiness of a context free language is decidable. Therefore, we can use a context-free language for ratio bounds.

Remark 6. The compilation of $P[i]$ to $\widetilde{P}[i]$ is easy to describe when $P[i]$ is given in FMSO. The model-checking algorithm, however, requires the translation of the FMSO formulas to automata or regular expressions. In general, the translation is non-elementary. The blow-up, however, is caused by nested negations, which are not typical in our setting, and is quite rare in practice [15]. For many faults, we can do the compilation on top of the regular description of the protocol and circumvent FMSO. In some cases, however, such as the one described in Remark 5, going through FMSO is much simpler.

Remark 7. It is sometimes desirable to check a protocol with respect to a combination of faults (for example, whether it is self-stabilized in the presence of a Byzantine fault). It is easy to see that our method is compositional, in the sense that the compilations described in Section 3.2 can be applied on top of each other, each with its distribution of faults (at most one distribution, however, can induce a context-free language). Also, by relating the underlying alphabets that specify the mask for the faulty processes in each type fault, it is possible to relate the faulty processes in the different faults (for example, the processes that suffer from the transient faults are necessarily different from these that suffer from the Byzantine fault).

Remark 8. The user may use the methodology in a *query mode*, in which the bound in the fault distribution is not specified. Thus, the bound is of the form $\{L, U\} \times \{\text{S}, \text{F}\} \times \{?\}$, and the user asks for the maximal/minimal number m of faulty/sound processes with which the property holds/is violated. Since the language D plays a role only in the last step of the model-checking procedure, an algorithm that does a binary search for m can reuse the result of the fixed-point computation that the steps that are independent of D have calculated, and only project it iteratively on different distribution languages. A similar approach can be used for ratio bounds.

We now demonstrate the application of our methodology for the verification of the token-ring protocol presented in Example 1. In the full version of the paper we consider a more impressive application, to the reliable broadcasting protocol of [14].

Example 4. Consider the token-ring protocol given in Example 1. We would like to verify whether it satisfies the mutual-exclusion and non-starvation properties specified in Example 2, in the presence of fail-stop, and Byzantine faults.

We start with fail-stop faults. We first follow the compilation described in Section 3.2 and compile the protocol $P[i]$ that is described in Example 3 to a protocol $\widetilde{P}[i]$ that takes the fail-stop faults into an account. Recall that $P[i]$ is over $\Sigma = \{\ell_0, \ell_1, \ell_2\} \times \{N, T\}$ and $\widetilde{P}[i]$ is over $\widetilde{\Sigma} = \Sigma \times \{\text{A}, \text{H}\} \times \{\text{S}, \text{F}\}$. For $S_{loc} \subseteq \{\ell_0, \ell_1, \ell_2\}$, $S_{tok} \subseteq \{N, T\}$, $S_t \subseteq \{\text{A}, \text{H}\}$, and $S_f \subseteq \{\text{S}, \text{F}\}$, we use $[S_{loc}, S_{tok}, S_l, S_f]$ to abbreviate the regular expression that is the union of all letters $\langle s_{loc}, s_{toc}, s_t, s_f \rangle \in S_{loc} \times S_{toc} \times S_t \times S_f$.

When $S_{loc} = \Sigma_{loc}$, we replace S_{loc} by _. When S_{loc} is a singleton $\{s_{loc}\}$, we simply write s_{loc}, and similarly for the other underlying alphabets.

Since mutual exclusion is a safety property, the model-checking procedure is simple, and we only have to check whether the set of reachable states (per a given distribution language) intersects the language L_{bad} of configurations that violate the property (the language is generated automatically from the Büchi transducer for the negation of the formula, and it is extended to the alphabet $\widetilde{\Sigma}$). Formally, $L_{bad} := \widetilde{\Sigma}^* \cdot [\ell_2, \_ , \_ , \_] \cdot \widetilde{\Sigma}^* \cdot [\ell_2, \_ , \_ , \_] \cdot \widetilde{\Sigma}^*$.

Using acceleration methods such as in [28,22], we can calculate the set L_{reach} of reachable configurations of $\tilde{P}$. $L_{reach} = [\{\ell_0, \ell_1\}, N, \_ , \_]^* \cdot [\_ , T, \_ , \_] \cdot [\{\ell_0, \ell_1\}, N, \_ , \_]^*$.

It is easy to see that the intersection of L_{bad} and L_{reach} is empty, regardless of the distribution bound. Thus, even if all processes are faulty, mutual exclusion holds.

As for non-starvation, the product of $\widetilde{P}$ with the Büchi transducer for the negation of the property is not empty even for the distribution bound $\langle U, \text{F}, 1\rangle$. Indeed, the computation that begins in the configuration $\langle \ell_0, T, \text{H}, \text{F}\rangle \cdot \langle \ell_0, N, \text{A}, \text{S}\rangle^*$, and then loops forever in the configuration $\langle \ell_0, T, \text{H}, \text{F}\rangle \cdot \langle \ell_1, N, \text{A}, \text{S}\rangle^*$, is a computation of $\widetilde{P}$ that is accepted by the Büchi transducer. Also, all the configurations along it belong to the distribution language $\text{S}^* \cdot (\text{S} + \text{F}) \cdot \text{S}^*$ that is induced by $\langle U, \text{F}, 1\rangle$. Note that the computation corresponds to the case the first process fail-stops as soon as the execution of the protocol begins, causing the token not to be passed at all.

Let us now move to Byzantine faults. Since a Byzantine process can produce or destroy tokens as he sees fit, the protocol is not resistant to Byzantine faults, even with a single Byzantine process. Recall that when modelling Byzantine faults, we have $\widetilde{\Sigma} = \Sigma_{loc} \times \Sigma_{tok} \times \{\text{S}, \text{F}\}$. The protocol $\widetilde{P}$ contains the computation that begins with the configuration $\langle \ell_0, T, \text{F}\rangle \cdot \langle \ell_0, N, \text{S}\rangle^*$, then moves to $\langle \ell_1, T, \text{F}\rangle \cdot \langle \ell_1, T, \text{S}\rangle \cdot \langle \ell_0, N, \text{S}\rangle^*$, and then moves to $\langle \ell_2, T, \text{F}\rangle \cdot \langle \ell_2, T, \text{S}\rangle \cdot \langle \ell_0, N, \text{S}\rangle^*$. Clearly, the third configuration intersects the language of bad configurations. Thus, mutual exclusion does not hold. Intuitively, the computation corresponds to the faulty process passing the token to the right but keeping a copy of the token to itself. The protocol also contains the computation that begins with the configuration $\langle \ell_0, T, \text{F}\rangle \cdot \langle \ell_0, N, \text{S}\rangle^*$, and then loops forever in the configuration $\langle \ell_0, N, \text{F}\rangle \cdot \langle \ell_1, N, \text{S}\rangle^*$. This computation is accepted by the Büchi transducer for the negation of the non-starvation property. Intuitively, it corresponds to a scenario where the faulty process destroys the token. Also, both computations are consistent with the fault distribution $\langle U, \text{F}, 1\rangle$. Thus, the properties do not hold in the presence of even a single Byzantine process.

Finally, in timing faults, the set of reachable states is similar to the set of reachable states in the original protocol (lifted by the new underlying alphabets), thus mutual exclusion holds. Non-starvation holds too, as the computations in the product that do violate the property are not fair.

4 Discussion

State-of-the-art work on verifying fault tolerance of distributed systems is restricted to the non-parametric setting. This is true both in work studying specific protocols, as [27,26], which model-check protocols for the case of four processes; as well as in work describing a general methodology, as [6], which studies synthesis of distributed systems with a bounded number of processes.

The idea behind our methodology is simple: it is possible to augment the description of a process by an indication of whether the process is faulty or sound, it is possible to let the process proceed with both indications (in case it is sounds it follows the original protocol, in case it is faulty it follows a modified protocol that is automatically generated according to the type of fault), and it is possible to use a symbolic description of the fault distribution in order to control the number or the fraction of the faulty processes. The methodology is generic in the sense that it can be applied to both synchronous and asynchronous systems, considering a large variety of faults, and taking into account a large variety of distribution faults (any distribution that can be specified by a context-free language).

We demonstrated our methodology in the framework of regular model checking (augmented to support a context-free language describing the fault-distribution). We are optimistic about the application of the methodology in other approaches that address the parameterized setting. In particular, in the approach of *network invariants* [33], one tries to find a system I such that (1) I abstracts P or $P \| P$ and (2) I abstracts $I \| P$. Our initial results in this front show that assuming the composition is symmetric, it is possible to replace P in the above conditions by finite compositions of P with faulty versions of it. For example, in order to prove resistance to c faults, one can replace P by a composition of P with c faulty versions of it; likewise in order to prove the resistance to a $\frac{k_1}{k_2}$ fraction of faulty processes, one can replace P by a combination of k_1 instances of P with $k_2 - k_1$ instances of a faulty version of P. Our future research aims at generalizing these ideas further.

Acknowledgment. We thank Danny Dolev and Ezra Hoch for many helpful discussions on distributed systems and fault-tolerant protocols.

References

1. Abdulla, P.A., d'Orso, J., Jonsson, B., Nilsson, M.: Algorithmic improvements in regular model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 236–248. Springer, Heidelberg (2003)
2. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: Regular model checking for LTL(MSO). In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 348–360. Springer, Heidelberg (2004)
3. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
4. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307–309 (1986)
5. Arora, A., Gouda, M.G.: Closure and convergence: A foundation of fault-tolerant computing. Software Engineering 19(11), 1015–1027 (1993)
6. Attie, P.C., Arora, A., Emerson, E.A.: Synthesis of fault-tolerant concurrent programs. ACM TOPLAS 26, 128–185 (2004)
7. Awerbuch, B.: Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In: Proc. 19th STOC, pp. 230–240 (1987)

8. Baier, C., Bertrand, N., Schnoebelen, P.: On computing fixpoints in well-structured regular model checking, with applications to lossy channel systems. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 347–361. Springer, Heidelberg (2006)
9. Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat acceleration in symbolic model checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)
10. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
11. Büchi, J.R.: Weak second-order arithmetic and finite automata. Zeit. Math. Logik und Grundl. Math. 6, 66–92 (1960)
12. Keneddey Space Center. NASA space shuttle launch archive, mission STS-1 (1981) `http://science.ksc.nasa.gov/shuttle/missions/sts-1/mission-sts-1.html`
13. Daliot, A., Dolev, D., Parnas, H.: Linear time byzantine self-stabilizing clock synchronization. In: Proc. of 7th PODC, pp. 7–19 (2003)
14. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM Journal on Computing 12, 656–666 (1983)
15. Elgaard, J., Klarlund, N., Möller, A.: Mona 1.x: new techniques for WS1S and WS2S. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 516–520. Springer, Heidelberg (1998)
16. Elgot, C.: Decision problems of finite-automata design and related arithmetics. Trans. Amer. Math. Soc. 98, 21–51 (1961)
17. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: Proc. 17th CAD, pp. 236–255 (2000)
18. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. IJFCS 14(4), 527–550 (2003)
19. Faloutsos, M., Molle, M.: Optimal distributed algorithm for minimum spanning trees revisited. In: Proc. 14th PODC, pp. 231–237 (1995)
20. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. STTT 8(3), 261–279 (2004)
21. Fisman, D., Pnueli, A.: Beyond regular model checking. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, Springer, Heidelberg (2001)
22. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. ENTCS 138(3), 21–36 (2005)
23. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. TCS 256, 93–112 (2001)
24. Lesens, D., Halbwachs, N., Raymond, P.: Automatic verification of parameterized linear networks of processes. In: Proc. 24th POPL, pp. 346–357 (1997)
25. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1996)
26. Malekpour, M.R.: A byzantine fault-tolerant self-stabilization synchronization protocol for distributed clock synchronization systems. TR NASA/TM-2006-214322, NASA STI (2006)
27. Malekpour, M.R., Sinimiceanu, R.: Comments on the byzantine self-stabilization synchronization protocol: counterexamples. TR NASA/TM-2006-213951, NASA STI, (2006)
28. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In: Proc. 12th CAV, pp. 328–343 (2000)
29. Schlichting, R.D., Schneider, F.B.: Fail-stop processors: An approach to designing fault-tolerant computing systems. Computer Systems 1(3), 222–238 (1983)
30. Tanenbaum, A., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice Hall, Englewood Cliffs (2007)
31. Thomas, W.: Automata on infinite objects. Handbook of Theoretical Computer Science, 133–191 (1990)
32. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. I&C 115(1), 1–37 (1994)
33. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Proc. Automatic verification methods for finite state systems, pp. 68–80 (1990)