Eliminating Message Counters in Threshold Automata*

Ilina Stoilkovska[1,2], Igor Konnov[1], Josef Widder[1], and Florian Zuleger[2]

[1] Informal Systems, Vienna, Austria
{igor, josef}@informal.systems
[2] TU Wien, Vienna, Austria
{stoilkov, zuleger}@forsyte.at

Abstract. Threshold automata were introduced to give a formal semantics to distributed algorithms in a way that supports automated verification. While transitions in threshold automata are guarded by conditions over the number of globally sent messages, conditions in the pseudocode descriptions of distributed algorithms are usually formulated over the number of locally received messages. In this work, we provide an automated method to close the gap between these two representations. We propose threshold automata with guards over the number of received messages and present abstractions into guards over the number of sent messages, by eliminating the receive message counters. Our approach allows us for the first time to fully automatically verify models of distributed algorithms that are in one-to-one correspondence with their pseudocode. We prove that our method is sound, and present a criterion for completeness w.r.t. $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties (satisfied by all our benchmarks).

1 Introduction

In distributed algorithms, the actions that a process takes locally depend on the messages it has received from the other processes in the system. To enable an action, a process checks if a quorum has been obtained (e.g., majority, two-thirds, etc.) by counting the received messages. Statements such as "*wait until $n - t$ ECHO messages are received*" or "*if more than $n/2$ messages with the same value are received*", where n is the number of processes and t is the upper bound on the number of faults, are commonly found in the pseudocode of various algorithms.

The root cause that an action becomes enabled is not that enough messages are *received* (which is information local to a process), but that enough processes have *sent* messages (which is information global to the system). This leads to redundancy when producing a formal model: the information about whether an action is enabled is present in the global state of the system, as well as in the

* Partially supported by: Austrian Science Fund (FWF) via NFN RiSE (S11403, S11405) and doctoral college LogiCS W1255; Interchain Foundation, Switzerland. The final publication is available at Springer via https://doi.org/10.1007/978-3-030-59152-6_11

local state of the processes. As [11] shows, this redundancy may lead to spurious counterexamples when applying abstraction-based model checking, which prevents abstraction-based techniques from scaling beyond small examples.

Threshold automata [13] were introduced to model and verify asynchronous fault-tolerant distributed algorithms. They are effective for verification, as they eliminate this redundancy by only allowing expressions over the global variables (i.e., the variables that count the number of sent messages). That is, it suffices to translate the check whether a quorum has been obtained to a check whether enough messages have been sent. For many algorithms, this translation can easily be done manually, as was the case in [12] (and [20], for synchronous algorithms).

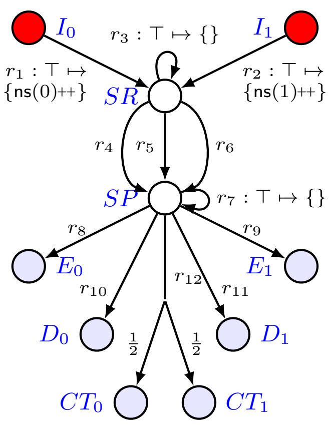
However, different classes of algorithms, such as, e.g., Ben-Or's randomized consensus algorithm [2], have more complex guards, where conditions over receive variables can occur in negated form. To model such algorithms in the threshold automata framework, one needs to translate negated conditions over receive variables to positive conditions over the global variables. Owing to implicit assumptions about the values of the receive and global variables, imposed by the asynchronous computation and faulty environment, eliminating the receive variables by hand becomes increasingly tedious and error-prone.

In this paper, we propose an automated method that translates guard expressions over the local receive variables into guard expressions over the global variables. We explicitly encode the relationship between the receive and global variables using a so-called *environment assumption*. The input is a threshold automaton, whose rules contain conditions over the receive variables, and an environment assumption. The output is a threshold automaton where the receive variables are eliminated. We make the following contributions:

1. We introduce a new variant of threshold automata that allows guards over receive variables, and thus is a formalization which captures the constructs that appear in the pseudocode found in the literature.
2. To eliminate the receive variables, we use quantifier elimination for Presburger arithmetic [16,9,17]. This results in quantifier-free guard expressions over the shared variables, and constitutes a valid input to ByMC [14].
3. We show that this method is sound, i.e., that the resulting system is an overapproximation of the original system. For completeness, we present classes of threshold automata for which eliminating receive message counters preserves linear temporal properties without the next operator ($\mathsf{LTL}_{\text{-}\mathsf{X}}$).
4. In our experiments, we specified several fault-tolerant distributed algorithms with guards over receive variables. We implemented our technique in a prototype, and used it to obtain guards over global variables. When comparing the automatically generated automata to the manually constructed ones, we found flaws, such as missing or redundant rules, or incorrect guards in the manual benchmarks (which were done by some of the authors of this paper).
5. We verified the correctness of the resulting threshold automata using ByMC.

In this way, we establish a fully automated pipeline, that for a given algorithm: starts from a formal model of its pseucocode, produces a formal model suitable for verification, and automatically verifies its correctness.

```
1  bool v := input_value({0, 1});
2  int rnd := 1;
3  while (true) do
4   send (R,rnd,v) to all;
5   wait for n - t messages (R,rnd,*);
6   if received more than (n + t) / 2
7      messages (R,rnd,w) then
8    send (P,rnd,w,D) to all;
9   else send (P,rnd,?) to all;
10  wait for n - t messages (P,rnd,*);
11  if received at least t + 1
12     message (P,rnd,w,D) then {
13    v := w;
14    if received more than (n + t) / 2
15       messages (P,rnd,w,D) then
16     decide w;
17   }
18  else v := random({0, 1});
19  rnd := rnd + 1;
20 od
```



I_0
$r_3 : \top \mapsto \{\}$
I_1
$r_1 : \top \mapsto \{\mathsf{ns}(0)\text{++}\}$
$r_2 : \top \mapsto \{\mathsf{ns}(1)\text{++}\}$
SR
r_4
r_5
r_6
SP
$r_7 : \top \mapsto \{\}$
r_8
r_9
E_0
E_1
r_{10}
r_{12}
r_{11}
D_0
$\frac{1}{2}$
$\frac{1}{2}$
D_1
CT_0
CT_1

$r_4 : \psi_1 \wedge \mathsf{nr}_i(0) > (n+t)/2 \mapsto \{\mathsf{ns}(2)\text{++}\}$ $\quad$ $r_5 : \psi_1 \wedge \mathsf{nr}_i(1) > (n+t)/2 \mapsto \{\mathsf{ns}(3)\text{++}\}$
$r_6 : \psi_1 \wedge \mathsf{nr}_i(0) \leq (n+t)/2 \wedge \mathsf{nr}_i(1) \leq (n+t)/2 \mapsto \{\mathsf{ns}(4)\text{++}\}$
$r_8 : \psi_2 \wedge \mathsf{nr}_i(2) \leq (n+t)/2 \wedge \mathsf{nr}_i(3) \leq (n+t)/2 \wedge \mathsf{nr}_i(2) \geq t+1 \mapsto \{\}$
$r_9 : \psi_2 \wedge \mathsf{nr}_i(2) \leq (n+t)/2 \wedge \mathsf{nr}_i(3) \leq (n+t)/2 \wedge \mathsf{nr}_i(3) \geq t+1 \mapsto \{\}$
$r_{10} : \psi_2 \wedge \mathsf{nr}_i(2) > (n+t)/2 \mapsto \{\}$ $\quad$ $r_{11} : \psi_2 \wedge \mathsf{nr}_i(3) > (n+t)/2 \mapsto \{\}$
$r_{12} : \psi_2 \wedge \mathsf{nr}_i(2) < t+1 \wedge \mathsf{nr}_i(3) < t+1 \mapsto \{\}$

Fig. 1. The pseudocode of the probabilistic Byzantine consensus protocol by Ben-Or [2], with $n > 5t$, and its TA where $\psi_1 \equiv \mathsf{nr}_i(0) + \mathsf{nr}_i(1) \geq n - t$ and $\psi_2 \equiv \mathsf{nr}_i(2) + \mathsf{nr}_i(3) + \mathsf{nr}_i(4) \geq n - t$. We use the notation $r : \varphi \mapsto \{\mathsf{ns}(m)\text{++} \mid m \in M\}$, where φ is the rule guard, and $\{\mathsf{ns}(m)\text{++} \mid m \in M\}$ is the set of increments of send variables.

2 Overview on Our Approach

We discuss our approach using the example in Figure 1. It shows the pseudocode of the probabilistic consensus algorithm by Ben-Or [2], which describes the behavior of one process. A system consists of n processes, f of which are faulty; there is an upper bound $t \geq f$ on the number of faults. We comment on some typical peculiarities of the pseudocode in Figure 1: v in line 4 is a program variable, w in line 7 is an (implicitly) existentially quantified variable whose scope ranges from line 6 to line 8 (similarly in lines 11–13 and lines 14–16). Besides, the tuple notation of messages hides the different types of its components: In line 8, the quadruple (P, rnd, w, D) is sent, where P and D are message tags (from a finite domain), while rnd is an algorithm variable (integer), and w is the mentioned existentially quantified variable. In the other branch, in line 9 the triple $(P, rnd, ?)$ is sent, that is, the "w, D" pair is replaced by the single tag '?'. We highlight these constructs to emphasize the difficulty of understanding the algorithm descriptions given in pseudocode, as well as the need for formal models. Irrespective of this formalization challenge, this algorithm and many other fault-tolerant distributed algorithms typically contain the following constructs:

- setting the values of local variables, e.g., line 13,
- sending a message of type m, for $m \in M$, e.g., line 4,

– waiting until enough messages of some type have been received, e.g., line 5.

Threshold Automata. In this paper, we present a method for obtaining a formal model of a fault-tolerant distributed algorithm starting from its pseudocode. In Section 3, we generalize *threshold automata (TA)* [13], and propose the extension as a formalism to faithfully encode fault-tolerant distributed algorithms. The TA specifies the behavior of one process; a parallel composition of multiple copies of a TA specifies the behavior of a distributed system in a faulty environment.

The TA shown on the right in Figure 1 models one iteration of the while-loop starting in line 3. It resembles a control flow graph, where:

– the *locations* of the TA encode the values of the local variables and the value of the program counter. For example, I_0 encodes that a process sets v to 0 in line 1, while E_0 encodes the same assignment in line 13.
– sending a message $m \in M$ is captured by incrementing the send variable $\mathsf{ns}(m)$. For example, a process with initial value 0 sends $(R, rnd, 0)$ in line 4. We say that the message $(R, rnd, 0)$ is of type 0, and model the sending by a process moving from I_0 to SR, and incrementing the variable $\mathsf{ns}(0)$.
– waiting until enough messages are received is modeled by keeping processes in a so-called *wait location*, defined in Section 4. For example, once a process sends a message in line 4, it moves to line 5, where it waits for $n-t$ messages that can be either $(R, rnd, 0)$ or $(R, rnd, 1)$. In the TA, the wait location SR encodes that the process has sent a message of type either 0 or 1, and that it now waits to receive at least $n - t$ messages of type 0 or 1.

Eliminating Receive Variables. In Section 4, we introduce two types of TA: rcvTA, which have local transitions guarded by expressions over *local* receive variables $\mathsf{nr}_i(m)$, and sndTA, where the guards are over *global* send variables $\mathsf{ns}(m)$, for $m \in M$. We use rcvTA to encode the behavior of a single process, and sndTA for verification purposes. The approaches in [12,13] encoded distributed algorithms using sndTA, and defined techniques for verifying safety and liveness properties of systems of sndTA. Thus, to apply these techniques to systems of rcvTA, our goal is to automatically generate sndTA, given a rcvTA.

In Section 5, we propose an abstraction from rcvTA to sndTA, which translates guards over $\mathsf{nr}_i(m)$ to guards over $\mathsf{ns}(m)$, for $m \in M$, based on *quantifier elimination.* The translation incorporates the relationship between the send and receive variables in asynchronous faulty environments, encoded using an *environment assumption* Env. The environment assumption depends on the fault model; e.g., for Byzantine faults, Env has constraints of the kind: $\mathsf{nr}_i(m) \leq \mathsf{ns}(m) + f$, that is, a process can receive up to f messages more than the ones sent by correct processes, where f is the number of faulty processes. Given a guard φ over the receive variables, to obtain a guard $\widehat{\varphi}$ over the send variables, we apply quantifier elimination to the formula $\varphi' \equiv \exists \mathsf{nr}_i(0) \dots \exists \mathsf{nr}_i(|M|-1)\ (\varphi \wedge \mathsf{Env})$. This produces a quantifier-free formula $\widehat{\varphi}$ over the send variables, equivalent to φ'.

We implemented a prototype that automatically generates guards over the send variables. We used Z3 [10] to automate the quantifier elimination step. We encoded several algorithms from the literature using rcvTA, translated them to sndTA using our prototype, and verified their correctness using ByMC [14], which we report on in Section 8. For instance, for the guard $\varphi_6 \equiv \mathsf{nr}_i(0) + \mathsf{nr}_i(1) \geq n - t \wedge \mathsf{nr}_i(0) \leq (n+t)/2 \wedge \mathsf{nr}_i(1) \leq (n+t)/2$, of rule r_6 in Figure 1, our prototype applies quantifier elimination to the formula $\exists \mathsf{nr}_i(0) \ldots \exists \mathsf{nr}_i(4)\ (\varphi_6 \wedge \mathsf{Env})$ and outputs the guard $\widehat{\varphi}_6 \equiv \mathsf{ns}(0) + \mathsf{ns}(1) + f \geq n - t \wedge \mathsf{ns}(0) + f \geq (n - 3t)/2 \wedge \mathsf{ns}(1) + f \geq (n - 3t)/2 \wedge \widehat{\mathsf{Env}}$, where $\widehat{\mathsf{Env}}$ is what remains of Env after eliminating $\mathsf{nr}_i(0), \ldots, \mathsf{nr}_i(4)$.

Soundness and criteria for completeness. In Section 6, we show that a system of n copies of a generated sndTA is an *overapproximation* of a system of n copies of the original rcvTA, i.e., we show that the translation is sound. This allows us to check the properties of a system of n copies of rcvTA by checking the properties of the system of n copies of sndTA. In general, the translation is not complete. We characterize a class of TA, for which we show that the overapproximation is *precise* w.r.t. $\mathsf{LTL}_{\text{-}\mathsf{X}}$ properties. We call these TA *common*, as they capture common assumptions made by algorithm designers. A TA is common if a process either: (1) does not wait for messages of the same type in different wait locations, or (2) in a given wait location, it waits for *more* messages of the same type than in any of its predecessor wait locations. In Section 7, we propose a formalization of these two assumptions, allowing us to classify the TA of all our benchmarks as common. We present a construction which given an infinite trace of the system of n copies of sndTA, builds an infinite *stutter-equivalent* trace of the system of n copies of a common rcvTA.

3 Threshold Automata

Let M denote the set of types of messages that can be sent and received by the processes. A process i, for $1 \leq i \leq n$, has three kinds of variables:

- *local* variables, x_i, visible only to process i, that store values local to process i, such as, e.g., an initial value or a decision value;
- *receive* variables $\mathsf{nr}_i(m)$, visible only to process i, that accumulate the number of messages of types $m \in M$ that were received by process i;
- *send* variables, $\mathsf{ns}(m)$, shared by all processes, that accumulate the number of messages of types $m \in M$ that were sent by the processes.

A *threshold automaton* TA is a tuple $(\mathcal{L}, \mathcal{I}, \mathcal{R}, \Gamma, \Delta, \Pi, RC, \mathsf{Env})$ whose components are defined below.

Locations $\mathcal{L}$, $\mathcal{I}$. The *locations* $\ell \in \mathcal{L}$ encode the current value of the process local variables x_i, together with information about the program counter. The *initial locations* in $\mathcal{I} \subseteq \mathcal{L}$ encode the initial values of the process local variables.

Variables Γ, Δ. The set Γ of *shared variables* contains send variables $\mathsf{ns}(m)$, for $m \in M$, ranging over $\mathbb{N}$. The set Δ of *receive variables* contains receive variables $\mathsf{nr}_i(m)$, for $m \in M$, ranging over $\mathbb{N}$. Initially, the variables in Γ and Δ are set to 0. As they are used to count messages, their value cannot decrease.

Parameters Π, Resilience Condition RC. The set Π of *parameters* contains at least the parameter n, denoting the number of processes. The *resilience condition* RC is a linear arithmetic expression over the parameters from Π. Let $\boldsymbol{\pi}$ be a $|\Pi|$-dimensional *parameter vector*, and let $\mathbf{p} \in \mathbb{N}^{|\Pi|}$ be its valuation. If $\mathbf{p}$ satisfies RC, we call it an *admissible valuation* of $\boldsymbol{\pi}$, and define the set $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}^{|\Pi|} \mid \mathbf{p} \models RC\}$ of admissible valuations. The mapping $N : \mathbf{P}_{RC} \to \mathbb{N}$ maps $\mathbf{p} \in \mathbf{P}_{RC}$ to the number $N(\mathbf{p})$ of processes that participate in the algorithm. For each process i, we assume $1 \leq i \leq N(\mathbf{p})$.

The algorithm in Figure 1 has three parameters: $n, t, f \in \Pi$, and $\boldsymbol{\pi} = \langle n, t, f\rangle$. The resilience condition is $n > 5t \wedge t \geq f$. An admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ is $\langle 6, 1, 1\rangle$, as $\mathbf{p}[n] > 5\mathbf{p}[t] \wedge \mathbf{p}[t] \geq \mathbf{p}[f]$.

Rules $\mathcal{R}$. The set $\mathcal{R}$ of *rules* defines how processes move from one location to another. A *rule* $r \in \mathcal{R}$ is a tuple $(\mathit{from}, \mathit{to}, \varphi, \mathbf{u})$, where $\mathit{from}, \mathit{to} \in \mathcal{L}$ are locations, φ is a *guard*, and $\mathbf{u}$ is a $|\Gamma|$-dimensional *update vector* of values from the set $\{0, 1\}$. The guard φ checks if the rule can be executed, and will be defined below. The update vector $\mathbf{u}$ captures the increment of the shared variables.

For example, in Figure 1, executing the rule $r_1 = (I_0, SR, \top, \mathbf{u})$ moves a process i from location I_0 to location SR, by incrementing the value of $\mathsf{ns}(0)$. That is, $r_1.\mathbf{u}[\mathsf{ns}(0)] = 1$, and for every other shared variable $g \in \Gamma$, with $g \neq \mathsf{ns}(0)$, we have $r_1.\mathbf{u}[g] = 0$. Observe that the guard of r_1 is $r_1.\varphi = \top$, which means that r_1 can be executed whenever process i is in location I_0.

Propositions. Let $\boldsymbol{\gamma}$ denote the $|\Gamma|$-dimensional *shared variables vector*, and $\boldsymbol{\delta}$ the $|\Delta|$-dimensional *receive variables vector*. To express guards and temporal properties, we consider the following propositions:

- *ℓ-propositions*, $\mathrm{p}(\ell)$, for $\ell \in \mathcal{L}$, (which will be used in Section 6),
- *r-propositions*, $\mathbf{a} \cdot \boldsymbol{\delta} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$, such that $\mathbf{a} \in \mathbb{Z}^{|\Delta|}, \mathbf{b} \in \mathbb{Z}^{|\Pi|}, c \in \mathbb{Z}$,
- *s-propositions*, $\mathbf{a} \cdot \boldsymbol{\gamma} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$, such that $\mathbf{a} \in \mathbb{Z}^{|\Gamma|}, \mathbf{b} \in \mathbb{Z}^{|\Pi|}, c \in \mathbb{Z}$.

Guards. A *guard* φ is a Boolean combination of r-propositions and s-propositions. We denote by $\mathsf{Vars}_\Delta(\varphi) = \{\mathsf{nr}_i(m) \in \Delta \mid \mathsf{nr}_i(m) \text{ occurs in } \varphi\}$ the set of receive variables that occur in the guard φ. A guard φ is evaluated over tuples $(\mathbf{d}, \mathbf{g}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\Delta|}, \mathbf{g} \in \mathbb{N}^{|\Gamma|}, \mathbf{p} \in \mathbf{P}_{RC}$ are valuations of the vectors $\boldsymbol{\delta}$ of receive variables, $\boldsymbol{\gamma}$ of shared variables, and $\boldsymbol{\pi}$ of parameters. We define the semantics of r-propositions and s-propositions, the semantics of the Boolean connectives is standard. An r-proposition holds in $(\mathbf{d}, \mathbf{g}, \mathbf{p})$, i.e., $(\mathbf{d}, \mathbf{g}, \mathbf{p}) \models \mathbf{a} \cdot \boldsymbol{\delta} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$ iff $(\mathbf{d}, \mathbf{p}) \models \mathbf{a} \cdot \boldsymbol{\delta} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$ iff $\mathbf{a} \cdot \mathbf{d} \geq \mathbf{b} \cdot \mathbf{p} + c$. Similarly for s-propositions, we have $(\mathbf{d}, \mathbf{g}, \mathbf{p}) \models \mathbf{a} \cdot \boldsymbol{\gamma} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$ iff $(\mathbf{g}, \mathbf{p}) \models \mathbf{a} \cdot \boldsymbol{\gamma} \geq \mathbf{b} \cdot \boldsymbol{\pi} + c$ iff $\mathbf{a} \cdot \mathbf{g} \geq \mathbf{b} \cdot \mathbf{p} + c$.

The guard $r_4.\varphi$ of rule r_4 in the TA in Figure 1 is a conjunction of two r-propositions, as $\mathsf{nr}_i(0) > (n + t)/2$ is equivalent to $2\mathsf{nr}_i(0) \geq n + t + 1$. We have $\mathsf{Vars}_\Delta(r_4.\varphi) = \{\mathsf{nr}_i(0), \mathsf{nr}_i(1)\}$ and $\mathsf{Vars}_\Gamma(r_4.\varphi) = \emptyset$.

Environment assumption Env*.* The environment assumption Env is a conjunction of linear arithmetic constraints on the values of the receive, shared variables, and parameters. It is used to faithfully model the assumptions imposed by the fault model and the message communication. For example, for Byzantine faults,

$$\mathsf{Env} \equiv \bigwedge_{r\in\mathcal{R}} \mathsf{Env}(r.\varphi), \text{ for } \mathsf{Env}(r.\varphi) \equiv \bigwedge_{M'\subseteq M(r.\varphi)} \sum_{m\in M'} \mathsf{nr}_i(m) \leq \sum_{m\in M'} \mathsf{ns}(m) + f$$

where $M(r.\varphi)$ are the message types of the receive variables that occur in the guard $r.\varphi$, i.e., $m \in M(r.\varphi)$ iff $\mathsf{nr}_i(m) \in \mathsf{Vars}_\Delta(r.\varphi)$. The constraint $\mathsf{Env}(r.\varphi)$ states that the number of received messages of types in $M' \subseteq M(r.\varphi)$, is bounded by the number of sent messages of types in M' and the number f of faults.

4 Modeling Distributed Algorithms with TA

The definition of TA presented in Section 3 is very general. To faithfully model the sending and receiving of messages in fault-tolerant distributed algorithms, we introduce *elementary TA*, by restricting the locations and the guards.

We first define wait locations. A location $\ell \in \mathcal{L}$ is a *wait location* iff (W1) there exists exactly one $r \in \mathcal{R}$ with $r = (\ell, \ell, \top, \mathbf{0})$, and (W2) there exists at least one $r \in \mathcal{R}$ with $r = (\ell, \ell', \varphi, \mathbf{u})$, with $\ell \neq \ell'$, where $r.\varphi \neq \top$. A process in a wait location $\ell \in \mathcal{L}$ uses the self-loop rule (W1) to stay in ℓ while it awaits to receive enough messages, until some guard of a rule (W2) is satisfied. The process uses the rules (W2) to move to a new location once the number of messages passes some threshold. The self-loop rule is unguarded and updates no shared variables, that is, its guard is $\top$ and its update vector is $\mathbf{0}$. The outgoing rules that are not self-loops are guarded and can contain updates of shared variables.

In Figure 1, SR is a wait location, as it has a self-loop rule $r_3 = (SR, SR, \top, \mathbf{0})$ as well as three guarded outgoing rules, r_4, r_5, and r_6, that are not self-loops.

Definition 1. *A threshold automaton TA* $= (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Gamma, \Delta, \Pi, RC, \mathsf{Env})$ *is* elementary *iff r.from is a wait location for every* $r \in \mathcal{R}$*, with* $r.\varphi \neq \top$.

We now define the two kinds of elementary TA: *receive* and *send* TA. To do so, we introduce a *receive guard*, as a Boolean combination of r-propositions and s-propositions, and a *shared guard*, as a Boolean combination of s-propositions.

Definition 2 (Receive TA). *The elementary TA* $(\mathcal{L}, \mathcal{I}, \mathcal{R}_\Delta, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Delta)$ *is a* receive TA *(denoted by rcvTA) if* $r.\varphi$ *is a receive guard, for* $r \in \mathcal{R}_\Delta$.

Definition 3 (Send TA). *The elementary TA* $(\mathcal{L}, \mathcal{I}, \mathcal{R}_\Gamma, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Gamma)$ *is a* send TA *(denoted by sndTA) if* $\Delta = \emptyset$*, and* $r.\varphi$ *is a shared guard, for* $r \in \mathcal{R}_\Gamma$*. We omit* Δ *from the signature, and define sndTA* $= (\mathcal{L}, \mathcal{I}, \mathcal{R}_\Gamma, \Gamma, \Pi, RC, \mathsf{Env}_\Gamma)$.

For example, the TA in Figure 1 is a rcvTA. In the remainder of this section, we define the semantics of the parallel composition of $N(\mathbf{p})$ copies of receive and send TA as an asynchronous transition system and counter system, respectively.

4.1 Asynchronous Transition System $\mathsf{ATS}(\mathbf{p})$

Definition 4 ($\mathsf{ATS}(\mathbf{p})$)**.** *Given a rcvTA and $\mathbf{p} \in \mathbf{P}_{RC}$, the triple* $\mathsf{ATS}(\mathbf{p}) = \langle S(\mathbf{p}), S_0(\mathbf{p}), T(\mathbf{p})\rangle$ *is an* asynchronous transition system*, where $S(\mathbf{p}), S_0(\mathbf{p})$ are the set of* states *and* initial states*, and $T(\mathbf{p})$ is the* transition relation*.*

A *state* $s \in S(\mathbf{p})$ is a tuple $s = \langle \boldsymbol{\ell}, \mathbf{g}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p}\rangle$, where $\boldsymbol{\ell} \in \mathcal{L}^{N(\mathbf{p})}$ is a vector of *locations*, such that $\boldsymbol{\ell}[i] \in \mathcal{L}$, for $1 \leq i \leq N(\mathbf{p})$, is the current location of process i, the vector $\mathbf{g} \in \mathbb{N}^{|\Gamma|}$ is a valuation of the shared variables vector $\boldsymbol{\gamma}$, and the vector $\mathbf{nr}_i \in \mathbb{N}^{|M|}$, is a valuation of the receive variables vector $\boldsymbol{\delta}$ for process i. Each state $s \in S(\mathbf{p})$ satisfies the constraints imposed by the environment assumption Env_Δ. A state s_0 is *initial*, i.e., $s_0 \in S_0(\mathbf{p}) \subseteq S(\mathbf{p})$, if $\boldsymbol{\ell} \in \mathcal{I}^{N(\mathbf{p})}$, and $\mathbf{g}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}$ are initialized to $\mathbf{0}$.

A receive guard $r.\varphi$, for $r \in \mathcal{R}_\Delta$, is evaluated over tuples (s, i), where $s \in S(\mathbf{p})$ and $1 \leq i \leq N(\mathbf{p})$. We define $(s, i) \models r.\varphi$ iff $(s.\mathbf{nr}_i, s.\mathbf{g}, s.\mathbf{p}) \models r.\varphi$.

Given two states, $s, s' \in S(\mathbf{p})$, we say that $(s, s') \in T(\mathbf{p})$, if there exists a process i, for $1 \leq i \leq N(\mathbf{p})$, and a rule $r \in \mathcal{R}_\Delta$ such that: (T1) $s.\boldsymbol{\ell}[i] = r.\mathit{from}$ and $(s, i) \models r.\varphi$, (T2) $s'.\mathbf{g} = s.\mathbf{g} + r.\mathbf{u}$, (T3) $s'.\boldsymbol{\ell}[i] = r.\mathit{to}$, (T4) $s.\mathbf{nr}_i[m] \leq s'.\mathbf{nr}_i[m]$, for $m \in M$, and (T5) for all j such that $1 \leq j \leq N(\mathbf{p})$ and $j \neq i$, we have $s'.\boldsymbol{\ell}[j] = s.\boldsymbol{\ell}[j]$ and $s'.\mathbf{nr}_j[m] = s.\mathbf{nr}_j[m]$, for $m \in M$. A rule $r \in \mathcal{R}_\Delta$ is *enabled* in a state $s \in S(\mathbf{p})$ if there exists a process i with $1 \leq i \leq N(\mathbf{p})$ such that (T1) holds. A state $s' \in S(\mathbf{p})$ is the *result* of applying r to s if there exists a process i, with $1 \leq i \leq N(\mathbf{p})$, such that r is enabled in s and if s' satisfies (T2) to (T5).

A *path* in $\mathsf{ATS}(\mathbf{p})$ is the finite sequence $\{s_i\}_{i=0}^k$ of states, such that $(s_i, s_{i+1}) \in T(\mathbf{p})$, for $0 \leq i < k$. A path $\{s_i\}_{i=0}^k$ is an *execution* if $s_0 \in S_0(\mathbf{p})$.

4.2 Counter System $\mathsf{CS}(\mathbf{p})$

Definition 5 ($\mathsf{CS}(\mathbf{p})$ [13])**.** *Given a sndTA and $\mathbf{p} \in \mathbf{P}_{RC}$, the triple* $\mathsf{CS}(\mathbf{p}) = \langle \Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p})\rangle$ *is a* counter system*, where $\Sigma(\mathbf{p}), I(\mathbf{p})$ are the sets of* configurations *and* initial configurations*, and $R(\mathbf{p})$ is the* transition relation*.*

A *configuration* $\sigma \in \Sigma(\mathbf{p})$ is the triple $\sigma = \langle \boldsymbol{\kappa}, \mathbf{g}, \mathbf{p}\rangle$, where the vector $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is a vector of *counters*, s.t. $\sigma.\boldsymbol{\kappa}[\ell]$, for $\ell \in \mathcal{L}$, counts how many processes are in location ℓ, and the vector $\mathbf{g} \in \mathbb{N}^{|\Gamma|}$ is the valuation of the shared variables vector $\boldsymbol{\gamma}$. Every configuration $\sigma \in \Sigma(\mathbf{p})$ satisfies the constraint $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = N(\mathbf{p})$ and the environment assumption Env_Γ. A configuration σ_0 is *initial*, i.e., $\sigma_0 \in I(\mathbf{p}) \subseteq \Sigma(\mathbf{p})$, if $\sigma_0.\boldsymbol{\kappa}[\ell] = 0$, for $\ell \in \mathcal{L} \setminus \mathcal{I}$, and $\sigma_0.\mathbf{g} = \mathbf{0}$.

A shared guard $r.\varphi$, for $r \in \mathcal{R}_\Gamma$, is evaluated over $\sigma \in \Sigma(\mathbf{p})$ as follows. As $r.\varphi$ is a Boolean combination of s-propositions, we have $\sigma \models r.\varphi$ iff $(\sigma.\mathbf{g}, \sigma.\mathbf{p}) \models r.\varphi$.

Given $\sigma, \sigma' \in \Sigma(\mathbf{p})$, we say that $(\sigma, \sigma') \in R(\mathbf{p})$ if there exists a rule $r \in \mathcal{R}_\Gamma$ such that: (R1) $\sigma.\boldsymbol{\kappa}[r.\mathit{from}] \geq 1$ and $\sigma \models r.\varphi$, (R2) $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + r.\mathbf{u}$, (R3) $\sigma'.\boldsymbol{\kappa}[r.\mathit{from}] = \sigma.\boldsymbol{\kappa}[r.\mathit{from}] - 1$ and $\sigma'.\boldsymbol{\kappa}[r.\mathit{to}] = \sigma.\boldsymbol{\kappa}[r.\mathit{to}] + 1$, and (R4) for all $\ell \in \mathcal{L} \setminus \{r.\mathit{from}, r.\mathit{to}\}$, we have $\sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$. The rule $r \in \mathcal{R}_\Gamma$ is *enabled* in $\sigma \in \Sigma(\mathbf{p})$ if it satisfies condition (R1). We call $\sigma' \in \Sigma(\mathbf{p})$ the *result* of applying r to σ, if r is enabled in σ and σ' satisfies the conditions (R2) to (R4).

The *path* and *execution* in $\mathsf{CS}(\mathbf{p})$ are defined analogously as for $\mathsf{ATS}(\mathbf{p})$.

5 Abstracting rcvTA to sndTA

We perform the abstraction from rcvTA to sndTA in two steps. First, we add the environment assumption Env_Δ as a conjunct to every receive guard occurring on the rules of the rcvTA. Second, we eliminate the receive variables to obtain the shared guards and environment assumption Env_Γ of sndTA.

Let $\mathsf{rcvTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}_\Delta, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Delta)$ be a receive TA and let $\mathsf{rcvTA}' = (\mathcal{L}, \mathcal{I}, \mathcal{R}'_\Delta, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Delta)$ be the receive TA obtained by adding the environment assumption Env_Δ as a conjunct to every receive guard $r.\varphi$, for $r \in \mathcal{R}_\Delta$.

Definition 6. *Given a rule $r \in \mathcal{R}_\Delta$, its corresponding rule in rcvTA' is the rule $r' \in \mathcal{R}'_\Delta$, such that $r'.\mathit{from} = r.\mathit{from}$, $r'.\mathit{to} = r.\mathit{to}$, $r'.\mathbf{u} = r.\mathbf{u}$, and*

$$r'.\varphi = \mathsf{addEnv}_\Delta(r.\varphi)\ ,\ \textit{where}\ \mathsf{addEnv}_\Delta(r.\varphi) = \begin{cases} \top & \textit{if } r.\varphi = \top \\ r.\varphi \wedge \mathsf{Env}_\Delta & \textit{otherwise} \end{cases}$$

Proposition 1. *For every rule $r \in \mathcal{R}_\Delta$, state $s \in S(\mathbf{p})$, and process i, for $1 \leq i \leq N(\mathbf{p})$, we have $(s, i) \models r.\varphi$ iff $(s, i) \models \mathsf{addEnv}_\Delta(r.\varphi)$.*

Proposition 1 follows from Definitions 4 and 6. As a result of it, composing $N(\mathbf{p})$ copies of rcvTA and $N(\mathbf{p})$ copies of rcvTA' results in the same $\mathsf{ATS}(\mathbf{p})$.

Given the $\mathsf{rcvTA}' = (\mathcal{L}, \mathcal{I}, \mathcal{R}'_\Delta, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Delta)$, obtained from rcvTA by Definition 6, we now construct a $\mathsf{sndTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}_\Gamma, \Gamma, \Pi, RC, \mathsf{Env}_\Gamma)$ whose locations, shared variables, and parameters are the same as in rcvTA and rcvTA', and whose rules $\mathcal{R}_\Gamma$ and the environment assumption Env_Γ are defined as follows.

Definition 7. *Given a rule $r' \in \mathcal{R}'_\Delta$, its corresponding rule in sndTA is the rule $\widehat{r} \in \mathcal{R}_\Gamma$, such that $\widehat{r}.\mathit{from} = r'.\mathit{from}$, $\widehat{r}.\mathit{to} = r'.\mathit{to}$, $\widehat{r}.\mathbf{u} = r'.\mathbf{u}$, and*

$$\widehat{r}.\varphi = \mathsf{eliminate}_\Delta(r'.\varphi),\ \textit{with}\ \mathsf{eliminate}_\Delta(r'.\varphi) = \begin{cases} \top & \textit{if } r'.\varphi = \top \\ \mathsf{QE}(\exists \boldsymbol{\delta}\ r'.\varphi) & \textit{otherwise} \end{cases}$$

where $\boldsymbol{\delta}$ is the $|\Delta|$-dimensional vector of receive variables, and QE is a quantifier elimination procedure for linear integer arithmetic.

The environment assumption Env_Γ of sndTA is the formula $\mathsf{eliminate}_\Delta(\mathsf{Env}_\Delta)$.

To obtain the shared guards of a sndTA, we apply quantifier elimination to eliminate the existentially quantified variables from the formula $\exists \boldsymbol{\delta}\ r.\varphi \wedge \mathsf{Env}_\Delta$, where $r.\varphi$ is a receive guard. The result is a quantifier-free formula over the shared variables, which is logically equivalent to $\exists \boldsymbol{\delta}\ r.\varphi \wedge \mathsf{Env}_\Delta$. We obtain the environment assumption Env_Γ of a sndTA in a similar way. The following proposition is a consequence of Definition 7 and quantifier elimination.

Proposition 2. *For every rule $r' \in \mathcal{R}'_\Delta$ and state $s \in S(\mathbf{p})$, if there exists a process i, with $1 \leq i \leq N(\mathbf{p})$, such that $(s, i) \models r'.\varphi$, then $s \models \mathsf{eliminate}_\Delta(r'.\varphi)$.*

The converse of Proposition 2 does not hold in general, i.e., if $r.\varphi$ is a receive guard, $s \models \mathsf{eliminate}_\Delta(r'.\varphi)$ does not imply $(s, i) \models r.\varphi$, for some $1 \leq i \leq N(\mathbf{p})$. However, in this case, by quantifier elimination, we have that $s \models \mathsf{eliminate}_\Delta(r'.\varphi)$ implies $s \models \exists \boldsymbol{\delta}\ r'.\varphi$.

6 Soundness

The construction of sndTA defined in Section 5 is sound. Given a rcvTA and its corresponding sndTA, we show that there exists a simulation relation between the system $\mathsf{ATS}(\mathbf{p}) = \langle S(\mathbf{p}), S_0(\mathbf{p}), T(\mathbf{p})\rangle$, induced by rcvTA, and the counter system $\mathsf{CS}(\mathbf{p}) = \langle \Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p})\rangle$, induced by sndTA. From this, we conclude that every ACTL^* formula over a set AP of atomic propositions that holds in $\mathsf{CS}(\mathbf{p})$ also holds in $\mathsf{ATS}(\mathbf{p})$. In this paper, the set AP of *atomic propositions* contains ℓ-propositions and s-propositions (cf. Section 3).

Evaluating AP*.* We define two labeling functions: $\lambda_{S(\mathbf{p})}$ and $\lambda_{\Sigma(\mathbf{p})}$. The function $\lambda_{S(\mathbf{p})} : S(\mathbf{p}) \to 2^{\mathrm{AP}}$ assigns to a state $s \in S(\mathbf{p})$ the set of atomic propositions from AP that hold in s. The function $\lambda_{\Sigma(\mathbf{p})} : \Sigma(\mathbf{p}) \to 2^{\mathrm{AP}}$ is defined analogously. We define the semantics of ℓ-propositions: $\mathrm{p}(\ell)$ holds in $s \in S(\mathbf{p})$, i.e., $s \models \mathrm{p}(\ell)$, iff there exists a process i, with $1 \leq i \leq N(\mathbf{p})$, such that $s.\boldsymbol{\ell}[i] = \ell$. The ℓ-proposition $\mathrm{p}(\ell)$ holds in $\sigma \in \Sigma(\mathbf{p})$, that is, $\sigma \models \mathrm{p}(\ell)$ iff $\sigma.\boldsymbol{\kappa}[\ell] \geq 1$.

Simulation. A binary relation $R \subseteq S(\mathbf{p}) \times \Sigma(\mathbf{p})$ is a *simulation relation* [1] if:

1. for every $s_0 \in S_0(\mathbf{p})$, there exists $\sigma_0 \in I(\mathbf{p})$ such that $(s_0, \sigma_0) \in R$,
2. for every $(s, \sigma) \in R$ it holds that:
   (a) $\lambda_{S(\mathbf{p})}(s) = \lambda_{\Sigma(\mathbf{p})}(\sigma)$,
   (b) for every state $s' \in S(\mathbf{p})$ such that $(s, s') \in T(\mathbf{p})$, there exists a configuration $\sigma' \in \Sigma(\mathbf{p})$ such that $(\sigma, \sigma') \in R(\mathbf{p})$ and $(s', \sigma') \in R$.

We introduce an *abstraction mapping* from the set $S(\mathbf{p})$ of states of $\mathsf{ATS}(\mathbf{p})$ to the set $\Sigma(\mathbf{p})$ of configurations of $\mathsf{CS}(\mathbf{p})$.

Definition 8. *The* abstraction mapping $\alpha_{\mathbf{p}} : S(\mathbf{p}) \to \Sigma(\mathbf{p})$ *maps* $s \in S(\mathbf{p})$ *to* $\sigma \in \Sigma(\mathbf{p})$*, s.t.* $\sigma.\boldsymbol{\kappa}[\ell] = |\{i \mid s.\boldsymbol{\ell}[i] = \ell\}|$*, for* $\ell \in \mathcal{L}$*,* $\sigma.\mathbf{g} = s.\mathbf{g}$*, and* $\sigma.\mathbf{p} = s.\mathbf{p}$*.*

The main result of this section is stated in the theorem below. It follows from: (i) a state s in $\mathsf{ATS}(\mathbf{p})$ and its abstraction $\sigma = \alpha_{\mathbf{p}}(s)$ in $\mathsf{CS}(\mathbf{p})$ satisfy the same atomic propositions, a consequence of the semantics of atomic propositions, and (ii) if a rule $r \in \mathcal{R}_\Delta$ is enabled in s, then the rule $\widehat{r} \in \mathcal{R}_\Gamma$, obtained by Definitions 6 and 7, is enabled in σ, a consequence of Propositions 1 and 2.

Theorem 1. *The binary relation* $R = \{(s, \sigma) \mid s \in S(\mathbf{p}), \sigma \in \Sigma(\mathbf{p}), \sigma = \alpha_{\mathbf{p}}(s)\}$ *is a simulation relation.*

7 Sufficient Condition for Completeness

We introduce the class of *common* rcvTA, that formalizes assumptions often implicitly assumed by distributed algorithms designers. In a common rcvTA, for every two wait locations ℓ and ℓ', where ℓ' is reachable from ℓ, either: (1) a process waits for messages of different types in ℓ and ℓ', or (2) a process waits for more messages of the same type in ℓ' than in ℓ. Below, we give one possible formalization of common rcvTA, which allows us to establish stutter-trace inclusion [1] between the counter system $\mathsf{CS}(\mathbf{p})$ and the system $\mathsf{ATS}(\mathbf{p})$.

Theorem 2. *Let rcvTA be common, and sndTA its corresponding send TA. For every execution of* $\mathsf{CS}(\mathbf{p})$ *induced by sndTA, there exists a stutter-equivalent execution of* $\mathsf{ATS}(\mathbf{p})$ *induced by the common rcvTA.*

From Theorem 2, every $\mathsf{LTL}_{\text{-}\mathsf{X}}$ formula satisfied in $\mathsf{ATS}(\mathbf{p})$ is also satisfied in $\mathsf{CS}(\mathbf{p})$. As $\mathsf{LTL}_{\text{-}\mathsf{X}}$ is a fragment of ACTL^*, a consequence of Theorems 1 and 2 is:

Corollary 1. *Let rcvTA be common, and sndTA its corresponding send TA. Let* ϕ *be an* $\mathsf{LTL}_{\text{-}\mathsf{X}}$ *formula over the set* AP *of atomic propositions. For a given* $\mathbf{p} \in \mathbf{P}_{RC}$*, we have* $\mathsf{ATS}(\mathbf{p}) \models \phi$ *iff* $\mathsf{CS}(\mathbf{p}) \models \phi$*.*

We define the properties of common rcvTA, that allow us to show Theorem 2.

Definition 9. *A guard* φ *is* monotonic *iff for every* $\mathbf{d}, \mathbf{d}' \in \mathbb{N}^{|\Delta|}$*,* $\mathbf{g}, \mathbf{g}' \in \mathbb{N}^{|\Gamma|}$*,* $\mathbf{p} \in \mathbf{P}_{RC}$*,* $(\mathbf{d}, \mathbf{g}, \mathbf{p}) \models \varphi$*,* $\mathbf{d}[m] \leq \mathbf{d}'[m]$*, for* $m \in M$*, and* $\mathbf{g}[g] \leq \mathbf{g}'[g]$*, for* $g \in \Gamma$*, implies* $(\mathbf{d}', \mathbf{g}', \mathbf{p}) \models \varphi$*.*

The monotonicity of the guards captures constraints imposed by the message communication and distributed computation. It states that a monotonic guard changes its truth value at most once as the processes update the values of the receive and shared variables.

Definition 10. *Let* $P_\Delta = \{D_1, \ldots, D_k\}$ *be a partition over the set* Δ *of receive variables. An environment assumption* Env_Δ *is:*

- P_Δ-independent *iff* Env_Δ *is of the form* $\mathsf{Env}_\Delta = \bigwedge_{D \in P_\Delta} \psi_D$*, where* ψ_D *is a subformula of* Env_Δ*, such that* $\mathsf{Vars}_\Delta(\psi_D) = D$*.*
- D-closed under joins*, for* $D \in P_\Delta$*, iff for every* $\mathbf{d}, \mathbf{d}' \in \mathbb{N}^{|\Delta|}$*,* $\mathbf{g} \in \mathbb{N}^{|\Gamma|}$*, and* $\mathbf{p} \in \mathbf{P}_{RC}$*, such that* $\mathbf{d}[m] = \mathbf{d}'[m]$*, for* $\mathsf{nr}_i(m) \in \Delta \setminus D$*, we have* $(\mathbf{d}, \mathbf{g}, \mathbf{p}) \models \mathsf{Env}_\Delta$ *and* $(\mathbf{d}', \mathbf{g}, \mathbf{p}) \models \mathsf{Env}_\Delta$ *implies* $(\max\{\mathbf{d}, \mathbf{d}'\}, \mathbf{g}, \mathbf{p}) \models \mathsf{Env}_\Delta$*.*

The constraints of a P_Δ-independent environment assumption Env_Δ are expressions over disjoint sets of variables. Under P_Δ-independence, for an environment assumption Env_Δ which is D-closed under joins, for $D \in P_\Delta$, there exists a maximal valuation of the variables in D such that Env_Δ is satisfied.

For Byzantine faults, Env_Δ is D-closed under joins iff $|D| = 1$. To show a counterexample where $|D| > 1$, consider some $D = \{\mathsf{nr}_i(0), \mathsf{nr}_i(1)\}$ of size 2, and a receive guard $\varphi = \mathsf{nr}_i(0) + \mathsf{nr}_i(1) \geq n - t$. Then, Env_Δ has three conjuncts: (i) $\mathsf{nr}_i(0) \leq \mathsf{ns}(0) + f$, (ii) $\mathsf{nr}_i(1) \leq \mathsf{ns}(1) + f$, and (iii) $\mathsf{nr}_i(0) + \mathsf{nr}_i(1) \leq \mathsf{ns}(0) + \mathsf{ns}(1) + f$. Consider $f = \mathsf{ns}(0) = \mathsf{ns}(1) = 1$, $\mathsf{nr}_i(0) = 2$, $\mathsf{nr}_i(1) = 1$, and $\mathsf{nr}_i(0) = 1$, $\mathsf{nr}_i(1) = 2$. Taking the maximum of these two valuations violates the constraint (iii). For crash faults, Env_Δ is D-closed under joins for all $D \in P_\Delta$.

The rule r is a *predecessor* of rule r', for $r, r' \in \mathcal{R}_\Delta$, iff $r.\varphi \neq \top$, $r'.\varphi \neq \top$, and $r'.\mathit{from}$ is reachable from $r.\mathit{from}$ by a path that starts with r.

Definition 11. *A* $\mathit{rcvTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}_\Delta, \Gamma, \Delta, \Pi, RC, \mathsf{Env}_\Delta)$ *is* common *iff*

- *there exists a partition* P_Δ*, where* $\mathsf{Vars}_\Delta(r.\varphi) \in P_\Delta$*, for* $r \in \mathcal{R}_\Delta$*,*
- Env_Δ *is* P_Δ*-independent,*

– *for every two rules $r, r' \in \mathcal{R}_\Delta$, such that r is a predecessor of r' either*
   1. $\mathsf{Vars}_\Delta(r.\varphi) \cap \mathsf{Vars}_\Delta(r'.\varphi) = \emptyset$, *or*
   2. Env_Δ *is* $\mathsf{Vars}_\Delta(r'.\varphi)$*-closed under joins and the guard* $r'.\varphi$ *is monotonic.*

The rcvTA of all our benchmarks are common. They are either: Byzantine, with non-overlapping variables on the guards outgoing of wait locations or with partition elements of size 1; or crash-faulty. Consider Figure 1. We have $P_\Delta = \{\mathsf{Vars}_\Delta(r_{SR}.\varphi), \mathsf{Vars}_\Delta(r_{SP}.\varphi)\}$, where $\mathsf{Vars}_\Delta(r_{SR}.\varphi) = \{\mathsf{nr}_i(0), \mathsf{nr}_i(1)\}$, and $\mathsf{Vars}_\Delta(r_{SP}.\varphi) = \{\mathsf{nr}_i(2), \mathsf{nr}_i(3), \mathsf{nr}_i(4)\}$. For Byzantine faults and the partition P_Δ, we have Env_Δ is P_Δ-independent. For $r_{SP} \in \{r_8, \ldots, r_{12}\}$ and its predecessors $r_{SR} \in \{r_4, \ldots, r_6\}$ SR, we have $\mathsf{Vars}_\Delta(r_{SP}) \cap \mathsf{Vars}_\Delta(r_{SR}) = \emptyset$.

Stutter-equivalent executions. Let rcvTA be common, and let sndTA be its corresponding send TA. Given an infinite execution $exec_{\mathsf{CS}} = \{\sigma_j\}_{j\in\mathbb{N}}$ in $\mathsf{CS}(\mathbf{p})$, induced by sndTA, we construct an infinite execution $exec_{\mathsf{ATS}} = \{s_i\}_{i\in\mathbb{N}}$ in $\mathsf{ATS}(\mathbf{p})$, induced by the common rcvTA, which is stutter-equivalent to $exec_{\mathsf{CS}}$ as follows:

1. constructing the initial state s_0 of $exec_{\mathsf{ATS}}$, and
2. for every transition (σ, σ') in $exec_{\mathsf{CS}}$, extending the execution $exec_{\mathsf{ATS}}$ either by a single transition or by a path consisting of two transitions.

The construction satisfies two invariants: (I1) for $\sigma \in \Sigma(\mathbf{p})$, which is the origin of the transition (σ, σ') in step 2, and $s \in S(\mathbf{p})$ at the tail of $exec_{\mathsf{ATS}}$ before executing step 2, it holds that $\sigma = \alpha_{\mathbf{p}}(s)$, and (I2) for every i, with $1 \le i \le N(\mathbf{p})$, and $s \in S(\mathbf{p})$ at the tail of $exec_{\mathsf{ATS}}$, we have $s.\mathbf{nr}_i[m] = 0$, for $m \in M$, s.t. $\mathsf{nr}_i(m)$ occurs on guards of rules that process i has not applied before reaching $s.\boldsymbol{\ell}[i]$.

Constructing the initial state. Let $\sigma_0 \in I(\mathbf{p})$ be the initial configuration of $exec_{\mathsf{CS}}$. We construct a state $s_0 \in S(\mathbf{p})$ such that $\sigma_0 = \alpha_{\mathbf{p}}(s_0)$, where $s_0.\mathbf{nr}_i[m] = 0$, for $m \in M$ and $1 \le i \le N(\mathbf{p})$.

Proposition 3. *Given $\sigma_0 \in I(\mathbf{p})$, the state $s_0 \in S(\mathbf{p})$, such that $\sigma_0 = \alpha_{\mathbf{p}}(s_0)$ and $s_0.\mathbf{nr}_i[m] = 0$, for $m \in M$ and $1 \le i \le N(\mathbf{p})$, is an initial state in* $\mathsf{ATS}(\mathbf{p})$.

Extending the execution $exec_{\mathsf{ATS}}$. The construction of $exec_{\mathsf{ATS}}$ proceeds iteratively: given a transition (σ, σ') in $exec_{\mathsf{CS}}$, the execution $exec_{\mathsf{ATS}}$ is extended by a single transition or a path consisting of two transitions. Let $r \in \mathcal{R}_\Delta$ denote the rule in rcvTA, which was used to construct the rule $\widehat{r} \in \mathcal{R}_\Gamma$, that was applied in the transition (σ, σ'). Let s be the state at the tail of $exec_{\mathsf{ATS}}$. By the invariant of the construction, $\alpha_{\mathbf{p}}(s) = \sigma$. There are two cases:

1. r is enabled in s – $exec_{\mathsf{ATS}}$ is extended by a single transition (s, s'),
2. r is not enabled in s – $exec_{\mathsf{ATS}}$ is extended by two transitions: (s, s'), (s', s'').

When r is enabled in s, the construction picks such a process i, and applies the rule r to the state s, such that the receive variables of process i are not updated. The result is the state s', such that: (A1) $s'.\boldsymbol{\ell}[i] = r.to$ and $s'.\mathbf{g} = s.\mathbf{g} + r.\mathbf{u}$, (A2) $s'.\mathbf{nr}_i[m] = s.\mathbf{nr}_i[m]$, for $m \in M$, that is, the process i does update its receive variables, (A3) for all j such that $1 \le j \le N(\mathbf{p})$, and $i \ne j$, we have $s'.\boldsymbol{\ell}[j] = s.\boldsymbol{\ell}[j]$ and $s'.\mathbf{nr}_j[m] = s.\mathbf{nr}_j[m]$, for $m \in M$.

Proposition 4. *Suppose r is enabled in s. Let $s' \in S(\mathbf{p})$ be obtained by applying (A1)-(A3). Then, $(s, s') \in T(\mathbf{p})$ is a transition in $\mathsf{ATS}(\mathbf{p})$.*

In the case when r is not enabled in s, there is no process i, with $1 \leq i \leq N(\mathbf{p})$ and $s.\boldsymbol{\ell}[i] = r.\mathit{from}$, such that $(s, i) \models r.\varphi$. This can happen if $r.\varphi$ is a receive guard, i.e., $\ell = r.\mathit{from}$ is a wait location. By $\sigma.\boldsymbol{\kappa}[\ell] \geq 1$ and the invariant (I1), there exists a process i in the wait location ℓ. The construction extends $\mathit{exec}_{\mathsf{ATS}}$ with: one transition in which the receive variables of process i are updated to values $\mathbf{nr} \in \mathbb{N}^{|\Delta|}$, such that $r.\varphi$ becomes enabled, and a second transition in which process i applies the rule r, without updating its receive variables.

By quantifier elimination and Definition 11, we find values $\mathbf{nr}$ that satisfy $r.\varphi$, where only the values of variables in $\mathsf{Vars}_\Delta(r.\varphi)$ get updated, i.e., where $\mathbf{nr}[m] = s.\mathbf{nr}_i[m]$ for $\mathsf{nr}_i(m) \in \Delta \setminus \mathsf{Vars}_\Delta(r.\varphi)$. For the values $\mathbf{nr}[m]$, for $\mathsf{nr}_i(m) \in \mathsf{Vars}_\Delta(r.\varphi)$, we take the maximum of $s.\mathbf{nr}_i[m]$ and the values for $\mathsf{nr}_i(m)$ in some arbitrary valuation that satisfies $r.\varphi$. Thus, process i can apply the self-loop rule $r' = (\ell, \ell, \top, \mathbf{0})$ to update its receive variables to $\mathbf{nr}$. The result is a state s', such that: (B1) $s'.\boldsymbol{\ell}[i] = s.\boldsymbol{\ell}[i]$ and $s'.\mathbf{g} = s.\mathbf{g}$, (B2) $s'.\mathbf{nr}_i[m] = \mathbf{nr}[m]$, for $m \in M$, (B3) for all j such that $1 \leq j \leq N(\mathbf{p})$, and $i \neq j$, we have $s'.\boldsymbol{\ell}[j] = s.\boldsymbol{\ell}[j]$ and $s'.\mathbf{nr}_j[m] = s.\mathbf{nr}_j[m]$, for $m \in M$. The rule r is enabled in s', as $s'.\boldsymbol{\ell}[i] = \ell$ and $(s', i) \models r.\varphi$, and is applied to s', using (A1)-(A3), resulting in the state s''.

Proposition 5. *Suppose r is not enabled in s. Let $s' \in S(\mathbf{p})$ be obtained by applying (B1)-(B3). Then, $(s, s') \in T(\mathbf{p})$ is a transition in $\mathsf{ATS}(\mathbf{p})$.*

We sketch how to prove Proposition 5 using the assumptions from Definition 11. If there exists a predecessor r_p of the rule r, such that $\mathsf{Vars}_\Delta(r_p.\varphi)$ and $\mathsf{Vars}_\Delta(r.\varphi)$ overlap, the monotonicity of $r.\varphi$ ensures that $(\mathbf{nr}, s.\mathbf{g}, s.\mathbf{p}) \models r.\varphi$, which implies $(s', i) \models r.\varphi$. The P_Δ-independence and $\mathsf{Vars}_\Delta(r.\varphi)$-closure under joins of Env_Δ allows us to reason only about the variables from $\mathsf{Vars}_\Delta(r.\varphi)$ to show that $(\mathbf{nr}, s.\mathbf{g}, s.\mathbf{p}) \models \mathsf{Env}_\Delta$, from which $s' \models \mathsf{Env}_\Delta$ follows. Otherwise, i.e., if $\mathsf{Vars}_\Delta(r_p.\varphi)$ and $\mathsf{Vars}_\Delta(r.\varphi)$ do not overlap, $(s', i) \models r.\varphi$ and $s' \models \mathsf{Env}_\Delta$ follow from the P_Δ-independence of Env_Δ and the invariant (I2).

Stutter-equivalence. For a given common rcvTA and its corresponding sndTA, the construction produces an infinite execution $\mathit{exec}_{\mathsf{ATS}}$ in $\mathsf{ATS}(\mathbf{p})$, given an infinite execution $\mathit{exec}_{\mathsf{CS}}$ in $\mathsf{CS}(\mathbf{p})$. Propositions 3, 4 and 5 ensure that $\mathit{exec}_{\mathsf{ATS}}$ is an execution in $\mathsf{ATS}(\mathbf{p})$. The invariants (I1) and (I2) are preserved during the construction. When r is enabled in s, it is easy to check that $\alpha_{\mathbf{p}}(s') = \sigma'$, where s' is obtained by (A1)-(A3), and no receive variables are updated. When r is not enabled in s, we have $\alpha_{\mathbf{p}}(s') = \sigma$, and $\alpha_{\mathbf{p}}(s'') = \sigma'$ where s' is obtained by (B1)-(B3), and s'' by (A1)-(A3). Here, only the receive variables of process i occurring on the rule r applied in s' are updated. Stutter-equivalence follows from (I1) and because s and σ satisfy the same atomic propositions.

8 Experimental Evaluation

In our experimental evaluation, we: (1) encoded several distributed algorithms from the literature as rcvTA, (2) implemented the method from Section 5 in a

Table 1. The algorithms we encoded as rcvTA and the model checking results. The experiments were run on a machine with 2,8 GHz Quad-Core Intel Core i7 and 16GB. We used Z3 v4.8.7 and ByMC v2.4.2. The columns stand for: QE – time to produce a sndTA, given a rcvTA as input; $|\Phi|$ – number of properties ByMC checked; sndTA $\models \phi$ (TA $\models \phi$) – time ByMC took to verify the properties of the automatically generated sndTA (manually encoded TA); $\Rightarrow$ – if all, some, or none of the sndTA guards imply the manual TA guards; **+L, F** – guard implications after adding lemmas or fixes.

Algorithm	Reference	Faults	QE	$\lvert\Phi\rvert$	sndTA $\models \phi$	TA $\models \phi$	$\Rightarrow$	**+L, F**
ABA	[7, Fig. 3]	Byzantine	0.43s	3	0.94s	0.8s	all	–
Ben-Or-Byz	[2, Prot. B]	Byzantine	1.04s	4	36.4s	0.64s	some	all
Ben-Or-clean	[2, Prot. A]	clean crash	0.78s	4	2m8s	1.03s	some	all
Ben-Or-crash	[2, Prot. A]	crash	1.31s	4	4×M.O.	23h, 2×M.O.	some	all
Bosco	[18, Alg. 1]	Byzantine	2.01s	5	30h5m	9m14s	some	some
CC-clean	[15, Fig. 1]	clean crash	0.65s	3	0.95s	0.67s	all	–
CC-crash	[15, Fig. 1]	crash	0.33s	3	1h59m	40.8s	all	–
FRB	[8, Fig. 4]	crash	0.14s	3	3.66s	0.97s	none	all
RS-Bosco	[18, Alg. 2]	Byzantine	9.51s	5	–	–	some	some
STRB	[19, Fig. 2]	Byzantine	0.31s	3	0.97s	0.75s	all	–

prototype that produces the corresponding sndTA, (3) compared the output to the existing manual encodings from the benchmark repository [3], and (4) verified the properties of the sndTA using the tool ByMC [14].

Encoding rcvTA. To encode distributed algorithms as rcvTA, we extended the TA format defined by ByMC with declarations of receive variables and environment constraints. For each of our benchmarks (cf. Table 1), there already exists a manually produced TA. For some crash-tolerant benchmarks, we also encoded a "clean crash" variant, where the crashed processes do not send messages.

Quantifier Elimination. We implemented a script that parses the input rcvTA, and creates a sndTA according to the abstraction from Section 5, whose rules have shared guards, obtained by applying Z3 [10] tactics for quantifier elimination [5,6], to formulas of the form $\exists \boldsymbol{\delta}\ r.\varphi \wedge \mathsf{Env}_\Delta$, where $r.\varphi$ is a receive guard. For all our benchmarks, the sndTA is generated within seconds, as shown in Table 1.

Analyzing the sndTA. We used Z3 to check if the guards for the automatically generated sndTA imply the guards of the manually encoded TA from [3]. This check allowed us to either verify that the earlier, manually encoded TA faithfully model the benchmark algorithms, or detect discrepancies, which we investigated further. Due to our completeness result (cf. Section 7), our technique produces the strongest possible guards. Hence, we expected implication for all the benchmarks. This is indeed the case for ABA, CC-*, and STRB. To our surprise, the implication did not hold for all the guards of the remaining benchmarks.

For Ben-Or-crash and FRB, we found flaws in the manual encodings. These algorithms tolerate crash faults, where the number of messages sent by faulty processes is stored in shared variables $\mathsf{ns}_f(m)$, and the environment assumption has constraints of the form $\mathsf{nr}_i(m) \leq \mathsf{ns}(m) + \mathsf{ns}_f(m)$. We identified that the variables $\mathsf{ns}_f(m)$ did not occur in the manual guards, i.e., it was assumed that $\mathsf{nr}_i(m) \leq \mathsf{ns}(m)$ (this encodes clean crashes). We fixed the manual guards by adding the variables $\mathsf{ns}_f(m)$, which made the benchmark Ben-Or-crash harder to check than previously reported in [4]: for the corrected TA, ByMC checked two properties in a day and ran out of memory for the other two (and for all four properties when checking the sndTA). By adding $\mathsf{ns}_f(m)$ to the manual guards of FRB we verified that the automatically generated guards are indeed stronger.

For all the Ben-Or-* benchmarks, we had to add lemmas to the environment assumption Env_Δ in order to verify that the automatically generated guards imply the manual guards. The key finding is that these lemmas were implicitly used in the manual encoding of the benchmarks in [4]. For instance, to get guards for $r_8, \ldots, r_{12}$ in Figure 1 that imply the manual guards, we added the lemma $\mathsf{ns}(2) = 0 \vee \mathsf{ns}(3) = 0$ to Env_Δ. To ensure soundness, it suffices to check (with Z3) that the rules which increment $\mathsf{ns}(2)$ or $\mathsf{ns}(3)$ cannot both be enabled.

For the most complicated benchmarks, Bosco and RS-Bosco, we could not find the right lemmas which ensure that all automatically generated guards imply all manual guards. Further, after inspecting the manual guards for several hours, we were not able to establish if those which are not implied are indeed wrong. Still, we successfully verified the properties of sndTA for Bosco with ByMC. Checking the manual TA for RS-Bosco requires running ByMC on an MPI cluster, to which we currently have no access. Hence, we could not verify RS-Bosco.

Model Checking with ByMC. We verified the properties of eight out of ten sndTA that our script produced. We ran ByMC with both the automatically generated sndTA and the already existing manual TA as input (the results are in Table 1). The timeout for ByMC was set to 24 hours for each property that we checked.

Quantifier elimination for $\exists \boldsymbol{\delta}\ r.\varphi \wedge \mathsf{Env}_\Delta$ in Presburger arithmetic may produce a quantifier-free formula which contains divisibility constraints; that are not supported by ByMC. We encountered divisibility constraints in the automatically generated guards for the benchmarks Bosco and RS-Bosco. To apply ByMC, we extend our analysis by a phase that generates different versions of the rcvTA according to the different evaluations of the divisibility constraints. For example, if the divisibility constraint $n\%2 = 0$ occurs on a guard, we create two versions of the rcvTA: one where n is odd, and one where n is even. Based on these two rcvTA, our script produces two sndTA, which we check with ByMC.

9 Conclusions

Our automated method helped in finding glitches in the existing encoding of several benchmarks, which confirms our motivation of automatically constructing threshold automata. In addition to the glitches discussed in Section 8, we found the following problems in manual encodings: redundant rules (whose guards always evaluate to false), swapped guards (on rules r, r', where the guard of r should be $r'.\varphi$, and vice versa), and missing rules (that were simply omitted). This indicates that there is a real benefit of producing guards automatically.

However, our experimental results show that ByMC performs worse on the sndTA than on the manual TA. Since the automatically generated guards have more s-propositions than the manual guards, the search space that ByMC explores is larger than for the manual TA. In this paper, we focus on soundness and completeness of the translation rather than on efficiency. We suggest that a simplification step which eliminates redundant s-propositions will lead to a performance comparable to manual encodings, and we leave that for future work.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MITP (2008)
2. Ben-Or, M.: Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In: PODC (1983)
3. `https://github.com/konnov/fault-tolerant-benchmarks`
4. Bertrand, N., Konnov, I., Lazić, M., Widder, J.: Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In: CONCUR (2019)
5. Bjørner, N.: Linear Quantifier Elimination as an Abstract Decision Procedure. In: IJCAR (2010)
6. Bjørner, N., Janota, M.: Playing with Quantified Satisfaction. In: LPAR (2015)
7. Bracha, G., Toueg, S.: Asynchronous Consensus and Broadcast Protocols. J. ACM **32**(4) (1985)
8. Chandra, T.D., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. J. ACM **43**(2) (1996)
9. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Machine intelligence **7**(91-99) (1972)
10. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS (2008)
11. John, A., Konnov, I., Schmid, U., Veith, H., Widder, J.: Parameterized Model Checking of Fault-Tolerant Distributed Algorithms by Abstraction. In: FMCAD (2013)
12. Konnov, I., Lazić, M., Veith, H., Widder, J.: A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms. In: POPL (2017)
13. Konnov, I., Veith, H., Widder, J.: On the Completeness of Bounded Model Checking for Threshold-Based Distributed Algorithms: Reachability. Information and Computation (2017)
14. Konnov, I., Widder, J.: ByMC: Byzantine Model Checker. In: ISOLA (2018)
15. Mostéfaoui, A., Mourgaya, E., Parvédy, P.R., Raynal, M.: Evaluating the Condition-Based Approach to Solve Consensus. In: DSN (2003)
16. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus du I congres de Mathématiciens des Pays Slaves (1929)
17. Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM **35**(8) (1992)
18. Song, Y.J., van Renesse, R.: Bosco: One-Step Byzantine Asynchronous Consensus. In: DISC (2008)
19. Srikanth, T., Toueg, S.: Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms. Dist. Comp. (1987)
20. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. In: TACAS (2019)