Eliminating Message Counters in Synchronous Threshold Automata

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

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

Abstract. In previous work, we introduced synchronous threshold automata for the verification of synchronous fault-tolerant distributed algorithms, and presented a verification method based on bounded model checking. Modeling a distributed algorithm by a threshold automaton requires to correctly deal with the semantics for sending and receiving messages based on the fault assumption. This step was done manually so far, and required human ingenuity. Motivated by similar results for asynchronous threshold automata, in this paper we show that one can start from a faithful model of the distributed algorithm that includes the sending and receiving of messages, and then automatically obtain a threshold automaton by applying quantifier elimination on the receive message counters. In this way, we obtain a fully automated verification pipeline. We present an experimental evaluation, discovering a bug in our previous manual encoding. Interestingly, while quantifier elimination in general produces larger threshold automata than the manual encoding, the verification times are comparable and even faster in several cases, allowing us to verify benchmarks that could not be handled before.

1 Introduction

Formal modeling and automated verification of fault-tolerant distributed algorithms [2,28] received considerable attention recently, e.g., [8,20,29,32,38]. In the more classic approach towards distributed algorithms' correctness, algorithms are described in pseudo code, using send and receive operations whose semantics are typically not formalized, but given in English. As a result, this may lead to ambiguities that are an obstacle both for implementing distributed algorithms faithfully, as well as for computer-aided verification. Threshold automata were introduced as a formalization of fault-tolerant distributed algorithms with precise semantics [5,23,26], and effective automated verification methods have been introduced both for the asynchronous [22] and for the synchronous [36] case. While they are a concise model that allows to capture precisely the nondeterminism distributed systems exhibit due to the communication model and

Partially supported by: Interchain Foundation, Switzerland; Austrian Science Fund (FWF) via doctoral college LogiCS W1255.

F. Henglein et al. (Eds.): VMCAI 2021, LNCS 12597, pp. 196–218, 2021.
https://doi.org/10.1007/978-3-030-67067-2_10

partial faults, threshold automata in fact constitute a manual abstraction: a threshold automaton has to capture two major ingredients of a distributed system: (i) the local program control flow that is based on received messages and (ii) the semantics of send and receive operations in a fault-prone environment. For many classical distributed algorithms, this manual abstraction is quite immediate, but as has been observed in [37], more involved distributed algorithms are harder to abstract manually. This manual process consists in understanding how a fault assumption—that typically is well-understood but not formalized—changes the semantics of sending and receiving messages, which is a formalization step that typically requires human ingenuity. The more desirable approach is to have a precise and formal description of (i) and (ii), and to construct the abstraction automatically. This also allows to reuse (ii), that is, the formalization of given distributed computing model for new benchmarks. Indeed, in [37], for asynchronous algorithms, we introduced a method that takes as input formalizations of (i) and (ii) and automatically constructs threshold automata. By this, we have reduced the required expertise of the user, increased the degree of automation on the verification process, and indeed found some bugs in manual abstractions of asynchronous algorithms. However, the approach in [37] focuses on (asynchronous) interleaving semantics, and asynchronous message passing, which pose different challenges than the synchronous setting.

While distributed algorithms are mostly designed for asynchronous systems, there exists a considerable amount of literature that focuses on *synchronous* distributed algorithms. The synchronous computation model is relevant, both theoretically and practically: (a) a well-known impossibility result [18] reveals a class of problems for which a solution in the asynchronous model does not exist, but which can be solved in the synchronous model, (b) some real-time systems are actually built on top of synchronous distributed algorithms [24], and (c) several verification approaches reduce the asynchronous to the synchronous setting [4,12,13,15,19,25], enabling the transfer of verification techniques. For these reasons, verification in the synchronous setting received significant interest recently [1,17,29]. Applying verification techniques discovered a bug in an already published synchronous consensus algorithm, as reported in [27].

In [36], we proposed a synchronous variant of threshold automata along with an automated verification method based on bounded model checking. We experimentally evaluated our approach on a large number of benchmarks coming from the distributed systems literature. However, the framework in [36] is based on the manual abstraction described above.

Our Contributions. In this paper, we bring the automatic generation of threshold automata to the synchronous setting. We propose a *synchronous* threshold automata (STA) framework that allows us to:

1. model a given algorithm with an STA, whose guards are linear integer arithmetic expressions over the number of *received* messages, such that the obtained STA is in one-to-one correspondence with the pseudo code,

2. model the implicit assumptions imposed by the computation and fault models explicitly, using a so-called *environment assumption*, which is specific to the respective fault model and can be reused for different algorithms,
3. automatically translate the guards over the *local receive* variables into guards over the number of *globally sent* messages, using quantifier elimination,
4. pass the output of the translation as input to the verification tool proposed in [36], which implements a semi-decision procedure for computing the diameter, and performs bounded model checking.

In [36], the STA given as input to the verification tool was produced manually, that is, the steps 1–3 above were done by the user. By automating these steps, we reduce the ingenuity required by the user. We encoded the control flow and the environment assumptions of several synchronous algorithms in our framework and compared the resulting STA with the existing manual encodings from [34]. We confirm that manual abstraction is error-prone, as we discovered glitches in previous manually encoded STA. For all benchmarks, the automatically generated STA are comparable with the manual encodings. For some, the automatically generated STA could be verified faster. Thus in addition to increasing the degree of automation, we also gained in performance.

2 Our Approach at a Glance

Synchronous Distributed Algorithms. A distributed algorithm is a collection of n processes that perform a common task and exchange messages. At most t of the n processes can be faulty, and f processes are actually faulty. The numbers n, t, f are parameters, where n and t are "known", that is, they appear in the code (see Fig. 1), while f may differ according to the individual executions. In the synchronous computation model, the actions that a process takes locally depend on the messages that the process has received in the current round by other processes. Often, a process checks whether a quorum has been obtained (e.g., majority, two-thirds, etc.) by counting the number of messages it has received. Obtaining a quorum means that the number of *received* messages has to pass a given threshold, which should guarantee that it is safe for a correct process to take an action, and move to a new local state.

The threshold automata framework [23] is based on the observation that from the viewpoint of enabled transitions in a transition system, we may substitute the check whether a quorum of messages has been *received* with a check whether enough messages have been *sent*. For some algorithms, this substitution is straightforward, but others have more complicated guard expressions over the number of received messages. Consider, for example, the pseudo code of the algorithm PhaseQueen [6,9], presented in Fig. 1. The algorithm operates in phases, with two rounds per phase (lines 3–8 and 9–11). In round 1, all processes broadcast their value stored in the variable v (line 3), receive messages from other processes (line 4), and count the number of messages with value 0 (line 5) and value 1 (line 6). If a process received more than $2t$ messages with value 1, then it sets its value to 1 (line 7), otherwise it sets its value to 0 (line 8). In round 2,

```
1   v := input($\{0, 1\}$)
2   for each phase 1 through $t + 1$ do
3     broadcast v  /* round 1: full message exchange */
4     receive messages from other processes
5     C[0] := number of received 0's
6     C[1] := number of received 1's
7     if C[1] $> 2t$ then v := 1
8     else v := 0
9     if phase $= i$ then broadcast v  /* round 2: queen's broadcast */
10    receive queen's message $v_q$
11    if C[v] $< n - t$ then v := $v_q$
```

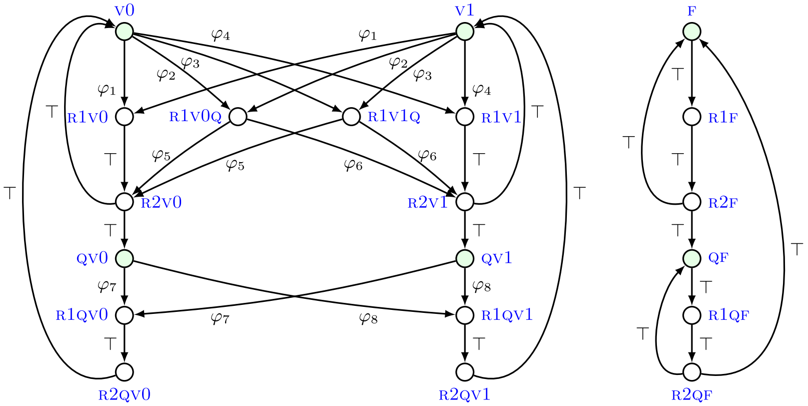
Fig. 1. The pseudo code of the Byzantine consensus algorithm PhaseQueen

a process i acts as a queen, if the number of the current phase is equal to i (line 9), and it is the only process that broadcasts (line 9). Each process receives the queen's value v_q (line 10), and checks if in round 1, it received less than $n-t$ messages with value equal to its own value v. If this is the case, the process sets its value to the value v_q received from the queen (line 11). This algorithm satisfies the property *agreement*: it ensures that after phase $t + 1$, i.e., after the loop on line 2 terminates, all correct processes have the same value v.

Receive Synchronous Threshold Automata. In Sect. 3, we propose a *new variant* of synchronous threshold automata, rSTA, with guards expressed over receive variables. Figure 2 shows the rSTA of the algorithm PhaseQueen. It corresponds to the control flow of the pseudo code in Fig. 1 as follows. The following locations capture local states of correct processes that are currently not a queen:

- Vi encodes that a process has the value $i \in \{0, 1\}$,
- R1Vi encodes that after the first round a process sets its value to $i \in \{0, 1\}$, and that it has received at least $n - t$ messages that have its value (i.e., the condition from line 11 evaluates to false),
- R1ViQ encodes that after the first round a process sets its value to $i \in \{0, 1\}$, and that it has received less than $n - t$ messages that have its value. Such a process will use the queen's message to update its value at the end of the second round (that is, the condition in line 11 evaluates to true),
- R2Vi encodes that after the second round a process sets its value to $i \in \{0, 1\}$.

From the location R2Vi, we have outgoing rules that bring the process back to the beginning of the next phase, i.e., to Vi, for $i \in \{0, 1\}$. Additionally, a process might move from the location R2Vi to QVi, for $i \in \{0, 1\}$, and thus become a queen in the next phase. The locations QVi, R1QVi, R2QVi, for $i \in \{0, 1\}$, capture the behavior of a correct process acting as a queen in the current phase. The Byzantine processes can act arbitrary, and their behavior is not explicitly modeled in the automaton. However, in some phase, the queen may be Byzantine. To capture this, we introduce locations, populated by a single Byzantine process, namely the locations $F = \{$F$, \ldots,$ R2QF$\}$. The queen is Byzantine in some phase, if the single Byzantine process moves from the location R2F to the location QF.



$\mathsf{sent}(m_0) = \{\text{V0}, \text{QV0}\}$

$\mathsf{sent}(m_1) = \{\text{V1}, \text{QV1}\}$

$\mathsf{sent}(m_{q0}) = \{\text{R1QV0}\}$

$\mathsf{sent}(m_{q1}) = \{\text{R1QV1}\}$

$\varphi_1 \equiv \mathsf{nr}(m_1) \leq 2t \wedge \mathsf{nr}(m_0) \geq n - t$

$\varphi_2 \equiv \mathsf{nr}(m_1) \leq 2t \wedge \mathsf{nr}(m_0) < n - t$

$\varphi_3 \equiv \mathsf{nr}(m_1) > 2t \wedge \mathsf{nr}(m_1) \geq n - t$

$\varphi_4 \equiv \mathsf{nr}(m_1) > 2t \wedge \mathsf{nr}(m_1) < n - t$

$\varphi_5 \equiv \mathsf{nr}(m_{q0}) \geq 1$

$\varphi_6 \equiv \mathsf{nr}(m_{q1}) \geq 1$

$\varphi_7 \equiv \mathsf{nr}(m_1) \leq 2t$

$\varphi_8 \equiv \mathsf{nr}(m_1) > 2t$

Fig. 2. The rSTA for the algorithm PhaseQueen [6], where $n > 4t \wedge t \geq f$.

Processes in locations Vi, QVi send messages of type m_i, that is, messages containing the value $i \in \{0, 1\}$. The message types m_{qi} are used to encode that the queen in the current phase sent a message with value $i \in \{0, 1\}$. When the queen process is Byzantine, it can send messages of type m_{q0} or m_{q1}. We write $\mathsf{sent}(m)$ to denote the set of locations where processes send a message of type m, and $\#\mathsf{sent}(m)$ for the number of sent messages of type m.

The receive guards $\varphi_1, \ldots, \varphi_8$ express conditions over the number of received messages of some message type, and capture expressions which appear in the pseudo code. We denote by $\mathsf{nr}(m_i)$ and $\mathsf{nr}(m_{qi})$ the number of messages containing the value $i \in \{0, 1\}$ that a process received from all processes in the first round of the phase (i.e., the value $\mathsf{C}[i]$ in the pseudo code, lines 5, 6) and by the queen in the second round of the phase, respectively. For example, the receive guard φ_1, occurring on rules that move processes to the location R1V0, checks if a process received at most $2t$ messages of type m_1 (the **else** branch is taken in line 8), and at least $n - t$ messages of type m_0 (the condition in line 11 is false).

We explicitly encode the relationship between the number of received and sent messages using an *environment assumption* Env, which bounds the number of received messages: (i) from below by the number of messages sent by the correct processes, and (ii) from above by the number of messages sent by both the correct and faulty processes. The bound (i) captures the assumptions of the synchronous communication, which requires that all messages sent by correct processes in a round are received in the same round, and the bound (ii) captures the non-determinism introduced by the faulty processes. E.g., in the algorithm $\mathsf{PhaseQueen}$, we have f Byzantine processes, which may send messages of arbitrary types. For the receive variable $\mathsf{nr}(m_i)$, we have the constraint $\#\mathsf{sent}(m_i) \leq \mathsf{nr}(m_i) \leq \#\mathsf{sent}(m_i) + f$ in the environment assumption Env.

The agreement property stated above is a safety property. To check if it holds, it suffices to check that after $t+1$ phases, either all processes are in locations $\text{V0}, \text{QV0}$, or in locations $\text{V1}, \text{QV1}$. The precise formalization of the properties we are interested in verifying can be found in [36].

Our Approach. In Sect. 6, we eliminate the receive variables in an rSTA using quantifier elimination for Presburger arithmetic [14,30,31]. We strengthen the receive guards by the environment assumption Env that imposes bounds on the values of the receive variables, which are existentially quantified. As a result, a quantifier-free guard expression over the number of sent messages is obtained. For example, the result of applying quantifier elimination to the guard φ_1 over the receive variables from Fig. 2, strengthened by the upper and lower bounds in the environment assumption Env, is the guard $\widehat{\varphi}_1$ with no receive variables:

$$\widehat{\varphi}_1 \equiv \#\mathsf{sent}(m_1) \leq 2t \wedge \#\mathsf{sent}(m_0) + f \geq n - t \wedge \widehat{\mathsf{Env}}$$

where $\widehat{\mathsf{Env}}$ are the residual constraints from eliminating the receive variables from the environment assumption Env. The condition $\mathsf{nr}(m_1) \leq 2t$ in the guard φ_1 is translated to $\#\mathsf{sent}(m_1) \leq 2t$, and the condition $\mathsf{nr}(m_0) \geq n-t$ to $\#\mathsf{sent}(m_0) + f \geq n - t$. That is, when translating the guards, the number of the faulty processes f is used in guards that check if the number of sent messages passes a threshold, whereas f is not used in guards that check if the number of sent messages is below a threshold. (Byzantine processes send messages arbitrarily.)

The STA where all guards over the receive variables are replaced by the automatically generated guards over the number of sent messages constitutes a valid input to the bounded model checking technique for STA from [36], which we use to verify their safety properties. We show that this method is sound and complete by showing the existence of a bisimulation between the composition of n copies of rSTA and the composition of n copies of the produced STA. Thus, eliminating the receive message counters preserves temporal properties. We implemented this technique and used it to automatically generate STA for a set of benchmarks, and compared them to the existing manually encoded STA for the same benchmarks. We discuss our the experimental results in Sect. 7.

3 Synchronous Threshold Automata

We recall synchronous threshold automata from [36] and extend them with receive variables below. A *synchronous threshold automaton (STA)* is the tuple $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$, whose locations $\mathcal{L}$, initial locations $\mathcal{I}$, rules $\mathcal{R}$, parameters Π, and resilience condition RC are defined below. We define the environment assumption Env in Sect. 3.2.

Parameters Π, Resilience Condition RC. We assume that the set Π of *parameters* contains at least the parameter n, denoting the total number of processes.

The *resilience condition* RC is a linear arithmetic expression over the parameters from Π. We call the vector $\boldsymbol{\pi} = \langle \pi_1, \ldots, \pi_{|\Pi|} \rangle$ the *parameter vector*, and the vector $\mathbf{p} = \langle \mathsf{p}_1, \ldots, \mathsf{p}_{|\Pi|} \rangle \in \mathbb{N}^{|\Pi|}$ a *valution* of $\boldsymbol{\pi}$. The set $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}^{|\Pi|} \mid \mathbf{p}$ is a valuation of $\boldsymbol{\pi}$ and $\mathbf{p}$ satisfies $RC\}$ contains the *admissible valuations* of $\boldsymbol{\pi}$. The mapping $N : \mathbf{P}_{RC} \to \mathbb{N}$ maps an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ to the number $N(\mathbf{p}) \in \mathbb{N}$ of *participating processes*, i.e., the number of processes whose behavior is modeled using the STA. We denote by $N(\boldsymbol{\pi})$ the linear combination of parameters that defines the number of participating processes.

Locations $\mathcal{L}, \mathcal{I}$. The *locations* $\ell \in \mathcal{L}$ encode the current value of the local variables of a process, together with information about the program counter. We assume that each local variable and the program counter ranges over a finite set of values, that is, we assume that the set $\mathcal{L}$ of locations is a finite set. The *initial locations* in $\mathcal{I} \subseteq \mathcal{L}$ encode the initial values of the local variables.

Message Types $\mathcal{M}$. Let $\mathcal{M}$ denote the set of *message types*. To encode sending messages in the STA, we define a mapping $\mathsf{sent} : \mathcal{M} \to 2^{\mathcal{L}}$, that maps a message type $m \in \mathcal{M}$ to a set $\mathsf{sent}(m) \subseteq \mathcal{L}$ of locations, such that $\mathsf{sent}(m) = \{\ell \in \mathcal{L} \mid$ a process in ℓ sends message of type $m\}$.

Let $L \subseteq \mathcal{L}$ denote a set of locations, and let $\#L$ denote the number of processes in locations from the set L. To define guards over the sent messages and express temporal properties, we define *c-propositions*:

$$\#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \text{ for } L \subseteq \mathcal{L},\ \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, \text{ and } b \in \mathbb{Z}$$

We denote by CP the set of c-propositions. If the set L of locations in the c-proposition is equal to the set $\mathsf{sent}(m)$, for some $m \in \mathcal{M}$, the c-proposition is used to check whether the number of messages of type $m \in \mathcal{M}$ is greater than or equal to a linear combination of the parameters, also called a *threshold*. Formally, the c-propositions are evaluated in tuples $(\boldsymbol{\kappa}, \mathbf{p})$, where $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of *counters*, and $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation:

$$(\boldsymbol{\kappa}, \mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{\ell \in L} \boldsymbol{\kappa}[\ell] \geq \boldsymbol{a} \cdot \mathbf{p} + b \tag{1}$$

Rules $\mathcal{R}$. A *rule* $r \in \mathcal{R}$ is a tuple $(\mathit{from}, \mathit{to}, \varphi)$, where: $\mathit{from}, \mathit{to} \in \mathcal{L}$ are locations, and φ is a *guard*, i.e., a Boolean combination of c-propositions. The guards $r.\varphi$, for $r \in \mathcal{R}$, analogously to (1), are evaluated in tuples $(\boldsymbol{\kappa}, \mathbf{p})$, and the semantics of the Boolean connectives is standard.

3.1 Receive Synchronous Threshold Automata

A *receive STA* is the tuple $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$, whose locations $\mathcal{L}$, initial locations $\mathcal{I}$, parameters Π, and resilience condition RC are defined as for STA. We define the receive variables Δ and rules $\mathcal{R}^{\Delta}$ below, and the environment assumption Env^{Δ} in Sect. 3.2.

Receive Variables Δ. The set Δ contains *receive variables* $\mathsf{nr}(m)$ that store the number of messages of type $m \in \mathcal{M}$ that were received by a process. Thus, $|\Delta| = |\mathcal{M}|$, as in Δ there is exactly one receive variable $\mathsf{nr}(m)$ per message type $m \in \mathcal{M}$. The values of the receive variables depend on the number of messages sent in a given round (discussed in more detail in Sect. 3.2).

Let $M \subseteq \mathcal{M}$ denote a set of message types, and let $\#M$ denote the total number of messages of types $m \in M$, received by some process. Observe that the notation $\#M$ is a shorthand for $\sum_{m \in M} \mathsf{nr}(m)$. We will use these two notations interchangeably. Further, when M is a singleton set, that is, when $M = \{m\}$, we will simply use the notation $\mathsf{nr}(m)$ to denote $\#\{m\}$. For the purpose of expressing guards over the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, we define *r-propositions*:

$$\#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b, \text{ such that } M \subseteq \mathcal{M}, \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, b \in \mathbb{Z}$$

We denote by RP the set of r-propositions. The intended meaning of the r-propositions is to check whether the total number of messages of types $m \in M$ received by some process i passes some threshold. Formally, they are evaluated in tuples $(\mathbf{d}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of values assigned to each receive variable $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, and $\mathbf{p} \in \mathbf{P}_{RC}$. We define:

$$(\mathbf{d}, \mathbf{p}) \models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{m \in M} \mathbf{d}[m] \geq \boldsymbol{a} \cdot \mathbf{p} + b \tag{2}$$

Rules $\mathcal{R}^\Delta$. Similarly to the way we defined rules of STA above, the rules $r^\Delta \in \mathcal{R}^\Delta$ in rSTA are tuples $r^\Delta = (\mathit{from}, \mathit{to}, \varphi)$, where $r^\Delta.\mathit{from}, r^\Delta.\mathit{to} \in \mathcal{L}$ are locations, and $r^\Delta.\varphi$ is a *receive guard*, which is a Boolean combination of c-propositions and r-propositions. The receive guards $r^\Delta.\varphi$, for $r^\Delta \in \mathcal{R}^\Delta$, are evaluated in tuples $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$. Given a tuple $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of valuations of the receive variables $\mathsf{nr}(m)$, for $m \in \mathcal{M}$, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of counters, and $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation, we evaluate c-propositions and r-propositions (the semantics of the Boolean connectives is standard):

$$\begin{aligned} (\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) &\models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad &&\text{iff} \quad (\boldsymbol{\kappa}, \mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad &&\text{(cf. (1))} \\ (\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) &\models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad &&\text{iff} \quad (\mathbf{d}, \mathbf{p}) \models \#M \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad &&\text{(cf. (2))} \end{aligned}$$

3.2 Environment Assumption and Modeling Faults

Depending on the fault model, when constructing a (receive) STA that models the behavior of a process running a given algorithm, we typically need to introduce additional locations or rules that are used to capture the behavior of the faulty processes. Additionally, to faithfully model the faulty environment, we will introduce constraints on the number of processes in given locations in both STA and rSTA, expressed using c-propositions, as well as constraints on the values of the receive variables of the rSTA, expressed using *e-propositions*:

$$\#M \geq \#L + \boldsymbol{a} \cdot \boldsymbol{\pi} + b, \text{ such that } M \subseteq \mathcal{M}, L \subseteq \mathcal{L}, \boldsymbol{a} \in \mathbb{Z}^{|\Pi|}, b \in \mathbb{Z}$$

We denote by EP the set of e-propositions. The e-propositions are evaluated in tuples $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$ where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$ is a vector of valuations of the receive variables, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of counters, and $\mathbf{p} \in \mathbf{P}_{RC}$. We say that:

$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \#M \geq \#L + \boldsymbol{a} \cdot \boldsymbol{\pi} + b \quad \text{iff} \quad \sum_{m \in M} \mathbf{d}[m] \geq \sum_{\ell \in L} \boldsymbol{\kappa}[\ell] + \boldsymbol{a} \cdot \mathbf{p} + b$$

The e-propositions will be used to express that the number of received messages is in the range from the number of messages sent by *correct* processes to the total number of sent messages (sent by both correct and faulty processes).

For STA, the environment assumption Env is a conjunction of c-propositions and their negations. For rSTA, the environment assumption Env^{Δ} is a conjunction of c-propositions, e-propositions and their negations. The c-propositions restrict the number of processes in certain locations, while the e-propositions restrict the values of the receive variables by relating them to the number of sent messages of the same type. We define the environment assumptions Env and Env^{Δ} of the STA and rSTA, respectively, as $\mathsf{Env} \equiv \mathsf{Env}_{\mathrm{CP}}$ and $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}_{\mathrm{CP}} \wedge \mathsf{Env}_{\mathrm{EP}}$, where $\mathsf{Env}_{\mathrm{CP}}$ and $\mathsf{Env}_{\mathrm{EP}}$ are conjunctions of c-propositions and e-propositions and their negations, respectively, such that:

$$\mathsf{Env}_{\mathrm{CP}} \equiv \mathsf{C1} \wedge \mathsf{C2} \wedge \mathsf{Env}_{\mathrm{CP},*} \quad \text{and} \quad \mathsf{Env}_{\mathrm{EP}} \equiv \mathsf{E1} \wedge \mathsf{Env}_{\mathrm{EP},*}$$

where, irrespective of the fault model, we have the following constraints:

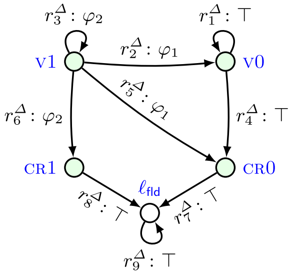
(C1) $\bigwedge_{\ell \in \mathcal{L}} \#\{\ell\} \geq 0$, i.e., the number of processes in a location ℓ is non-negative,
(C2) $\#\mathcal{L} = N(\boldsymbol{\pi})$, i.e., the number of processes in all locations $\mathcal{L}$ is equal to the number of participating processes,
(E1) $\bigwedge_{m \in \mathcal{M}} \#\mathsf{sent}(m) \leq \mathsf{nr}(m)$, i.e., the number $\mathsf{nr}(m)$ of received messages of each message type $m \in \mathcal{M}$ is bounded from below by the number $\#\mathsf{sent}(m)$ of messages of type m, sent by correct processes.

The formulas $\mathsf{Env}_{\mathrm{CP},*}$ and $\mathsf{Env}_{\mathrm{EP},*}$ for $* \in \{\mathsf{cr}, \mathsf{so}, \mathsf{byz}\}$, depend on the fault model, i.e., on whether we model crash, send omission, or Byzantine faults.

Crash Faults. Crash-faulty processes stop executing the algorithm prematurely and cannot restart. To model the behavior of the crash-faulty processes, the set $\mathcal{L}$ of locations of the (receive) STA is the set: $\mathcal{L} = \mathcal{L}_{\mathsf{corr}} \cup \mathcal{L}_{\mathsf{cr}} \cup \{\ell_{\mathsf{fld}}\}$, where $\mathcal{L}_{\mathsf{corr}}$ is a set of *correct* locations, $\mathcal{L}_{\mathsf{cr}} = \{\ell_{\mathsf{cr}} \mid \ell_{\mathsf{cr}} \text{ is a fresh copy of } \ell \in \mathcal{L}_{\mathsf{corr}}\}$ is a set of *crash* locations, and ℓ_{fld} is a *failed* location. The crash locations $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ model the same values of the local variables and program counter as their correct counterpart $\ell \in \mathcal{L}_{\mathsf{corr}}$. The difference is that processes in the crash locations $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ are flagged by the environment to crash in the current round. After crashing, they move to the failed location ℓ_{fld}, where they remain forever. This models that the crashed processes cannot restart.

A crash-faulty process may send a message to a subset of the other processes in the round in which it crashes. To model this, we introduce the mapping $\mathsf{sent}_{\mathsf{cr}} : \mathcal{M} \to 2^{\mathcal{L}_{\mathsf{cr}}}$, which defines, for each $m \in \mathcal{M}$, the set of crash locations $\mathsf{sent}_{\mathsf{cr}}(m) \subseteq$

```
best := input(V)
dec := ⊥
for rnd=1 to $\lfloor t/k \rfloor + 1$
  broadcast best
  receive values $b_1, \ldots b_\ell$
  best := min $\{b_1, \ldots b_\ell\}$
dec := best
```



$\varphi_1 \equiv \mathsf{nr}(m_0) \geq 1$
$\varphi_2 \equiv \mathsf{nr}(m_0) < 1$
$\mathsf{Env}_{\mathrm{CP},\mathsf{cr}} \equiv$
$\#\{\mathrm{CR0}, \mathrm{CR1}, \ell_{\mathsf{fld}}\} \leq f$
$\mathsf{Env}_{\mathrm{EP},\mathsf{cr}} \equiv$
$\mathsf{nr}(m_0) \leq \#\{\mathrm{V0}, \mathrm{CR0}\}$
$\wedge\, \mathsf{nr}(m_1) \leq \#\{\mathrm{V1}, \mathrm{CR1}\}$

Fig. 3. The pseudo code of the algorithm FloodMin for $k = 1$ [28], which tolerates crash faults, and the receive STA encoding its loop body.

$\mathcal{L}_{\mathsf{cr}}$ where processes send a message of type m. Then, $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m))$ denotes the number of messages sent by correct and crash-faulty processes. In addition to the new locations, we add the following new rules:

(cr1) for every rule $r \in \mathcal{R}$, if $r.\mathit{from} \in \mathcal{L}_{\mathsf{corr}}$ and $r.\mathit{to} \in \mathcal{L}_{\mathsf{corr}}$, then we add the rule $(r.\mathit{from}, \ell_{\mathsf{cr}}, r.\varphi)$, where $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$ is the crash location corresponding to $r.\mathit{to}$,
(cr2) for every crash location $\ell_{\mathsf{cr}} \in \mathcal{L}_{\mathsf{cr}}$, we add the rule $(\ell_{\mathsf{cr}}, \ell_{\mathsf{fld}}, \top)$,
(cr3) for the failed location ℓ_{fld}, we add the rule $(\ell_{\mathsf{fld}}, \ell_{\mathsf{fld}}, \top)$.

The rules (cr1) move processes from the correct to the crash locations, in rounds where the environment flags them as crashed. The rules (cr2) move processes from the crashed locations to the failed location, where they can only apply the self-loop rule (cr3), which keeps them in the failed location.

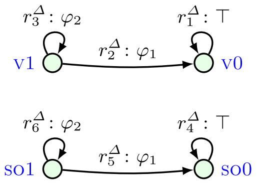
We model the behavior of crash-faulty processes explicitly, that is, we have $N(\boldsymbol{\pi}) = n$. The constraints $\mathsf{Env}_{\mathrm{CP},\mathsf{cr}}$ and $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$ for the crash fault model are:

$$\mathsf{Env}_{\mathrm{CP},\mathsf{cr}} = \#(\mathcal{L}_{\mathsf{cr}} \cup \{\ell_{\mathsf{fld}}\}) \leq f$$

$$\mathsf{Env}_{\mathrm{EP},\mathsf{cr}} \equiv \bigwedge_{m \in \mathcal{M}} \mathsf{nr}(m) \leq \#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m))$$

The formula $\mathsf{Env}_{\mathrm{CP},\mathsf{cr}}$ ensures that there are no more than f faults. The formula $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$ restricts the values of the receive variables by ensuring that the number of received messages of type $m \in \mathcal{M}$ for each process is a value, bounded from above by the number $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{cr}}(m))$ of messages of type m, sent by the correct processes and the processes flagged as crashed in the current round.

Figure 3 depicts the pseudo code and the rSTA of the crash-tolerant k-set agreement algorithm FloodMin, for $k = 1$ [28]. We identify the sets $\mathcal{L}_{\mathsf{corr}} = \{\mathrm{V0}, \mathrm{V1}\}$ of correct locations, $\mathcal{L}_{\mathsf{cr}} = \{\mathrm{CR0}, \mathrm{CR1}\}$ of crash locations, $\mathcal{M} = \{m_0, m_1\}$ of message types. The location $\mathrm{V}i$ encodes that a correct process has its variable best set to $i \in \{0, 1\}$, the location $\mathrm{CR}i$ encodes that the value of best of a crashed process is $i \in \{0, 1\}$, and the message type m_i encodes a message containing the value $i \in \{0, 1\}$. The failed location is ℓ_{fld}. We define $\mathsf{sent}(m_i) = \{\mathrm{V}i\}$ and $\mathsf{sent}_{\mathsf{cr}}(m_i) = \{\mathrm{CR}i\}$, for $i \in \{0, 1\}$. The two receive guards


$r_3^{\Delta} : \varphi_2$
$r_1^{\Delta} : \top$
$r_2^{\Delta} : \varphi_1$
v1
v0
$r_6^{\Delta} : \varphi_2$
$r_4^{\Delta} : \top$
$r_5^{\Delta} : \varphi_1$
so1
so0

$$\begin{aligned} \mathsf{Env}_{\mathrm{CP},\mathsf{so}} \equiv \quad & \#\{\mathrm{v}0, \mathrm{v}1\} = n - f \\ & \wedge \#\{\mathrm{so}0, \mathrm{so}1\} = f \\ \mathsf{Env}_{\mathrm{EP},\mathsf{so}} \equiv \quad & \mathsf{nr}(m_0) \leq \#\{\mathrm{v}0, \mathrm{so}0\} \\ & \wedge \mathsf{nr}(m_1) \leq \#\{\mathrm{v}1, \mathrm{so}1\} \end{aligned}$$

Fig. 4. The receive STA encoding the loop body of the algorithm FMinOmit for $k = 1$, which tolerates send omission faults and whose pseudo code is given in Fig. 3.

$\varphi_1 \equiv \mathsf{nr}(m_0) \geq 1$ and $\varphi_2 \equiv \mathsf{nr}(m_0) < 1$ check if a process received at least one message of type m_0 (i.e., if the minimal value 0 has been received in line 5 of the pseudo code) and no message of type m_0, respectively. The constraint $\mathsf{Env}_{\mathrm{CP},\mathsf{cr}}$ ensures that there are not more than f processes in the locations $\mathrm{CR}0, \mathrm{CR}1$, and ℓ_{fld} together. The constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$ bounds the values of the receive variables $\mathsf{nr}(m_i)$ from above by the number of processes in locations $\mathrm{V}i, \mathrm{CR}i$, for $i \in \{0, 1\}$.

Send Omission Faults. A send-omission-faulty process may omit to send a message, but acts as a correct process on the receiving side. We model algorithms tolerating send omission faults similarly to crash faults: the set $\mathcal{L}$ of locations is $\mathcal{L} = \mathcal{L}_{\mathsf{corr}} \cup \mathcal{L}_{\mathsf{so}}$, where $\mathcal{L}_{\mathsf{corr}}$ is a set of *correct* locations and $\mathcal{L}_{\mathsf{so}} = \{\ell_{\mathsf{so}} \mid \ell_{\mathsf{so}} \text{ is a fresh copy of } \ell \in \mathcal{L}_{\mathsf{corr}}\}$ is a set of *send-omission* locations. For every rule $r \in \mathcal{R}$ connecting two locations $\ell, \ell' \in \mathcal{L}_{\mathsf{corr}}$, there exists a rule $(\ell_{\mathsf{so}}, \ell'_{\mathsf{so}}, r.\varphi) \in \mathcal{R}$, connecting their two corresponding send-omission locations $\ell_{\mathsf{so}}, \ell'_{\mathsf{so}} \in \mathcal{L}_{\mathsf{so}}$. We introduce the mapping $\mathsf{sent}_{\mathsf{so}} : \mathcal{M} \rightarrow 2^{\mathcal{L}_{\mathsf{so}}}$, which defines the set of send-omission locations where processes send a message of type $m \in \mathcal{M}$.

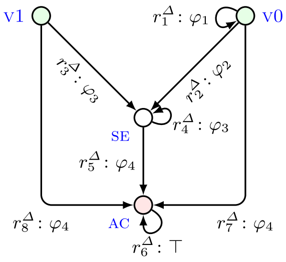
As there are no rules that connect the locations from $\mathcal{L}_{\mathsf{corr}}$ to the locations from $\mathcal{L}_{\mathsf{so}}$, the automaton consists of two parts: one used by the correct processes, and one used by the send-omission-faulty processes. The behavior of the send-omission-faulty processes is encoded explicitly, using locations and rules in the automaton, hence, we define $N(\boldsymbol{\pi}) = n$. The constraint $\mathsf{Env}_{\mathrm{CP},\mathsf{so}}$ ensures that the number of processes populating the correct locations is $n - f$, and the number of processes populating the send-omission locations is f. The constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{so}}$ ensures that the number of received messages of type $m \in \mathcal{M}$ for each process is bounded from above by the number $\#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{so}}(m))$ of messages of type m, sent by the correct and the send-omission-faulty processes. Formally:

$$\mathsf{Env}_{\mathrm{CP},\mathsf{so}} = \#\mathcal{L}_{\mathsf{corr}} = n - f \wedge \#\mathcal{L}_{\mathsf{so}} = f$$

$$\mathsf{Env}_{\mathrm{EP},\mathsf{so}} \equiv \bigwedge_{m \in \mathcal{M}} \mathsf{nr}(m) \leq \#(\mathsf{sent}(m) \cup \mathsf{sent}_{\mathsf{so}}(m))$$

Figure 4 depicts the rSTA for the k-set agreement algorithm FMinOmit, for $k = 1$, which is a variant of the algorithm FloodMin (Fig. 3) that tolerates send omission faults. We identify the sets $\mathcal{L}_{\mathsf{corr}} = \{\mathrm{v}0, \mathrm{v}1\}$ of correct locations, $\mathcal{L}_{\mathsf{so}} = \{\mathrm{so}0, \mathrm{so}1\}$ of send-omission locations, and $\mathcal{M} = \{m_0, m_1\}$ of message types. We define $\mathsf{sent}(m_i) = \{\mathrm{v}i\}$ and $\mathsf{sent}_{\mathsf{so}}(m_i) = \{\mathrm{so}i\}$, for $i \in \{0, 1\}$. The

```
1  v := input({0,1})
2  accept := ⊥
3  while true do
4    if v = 1 then
5      broadcast ECHO
6    receive messages
7    if received ECHO from
8    ≥ $t+1$ processes then
9      v := 1
10   if received ECHO from
11   ≥ $n-t$ processes then
12     accept := ⊤
```



$\varphi_1 \equiv \mathsf{nr}(m_{\mathsf{E}}) < t+1$
$\varphi_2 \equiv \mathsf{nr}(m_{\mathsf{E}}) \geq t+1$
$\varphi_3 \equiv \mathsf{nr}(m_{\mathsf{E}}) < n-t$
$\varphi_4 \equiv \mathsf{nr}(m_{\mathsf{E}}) \geq n-t$
$\mathsf{Env}_{\mathrm{EP},\mathsf{byz}} \equiv$
$\mathsf{nr}(m_{\mathsf{E}}) \leq \#\{\mathrm{V1}, \mathrm{SE}, \mathrm{AC}\} + f$

Fig. 5. The pseudo code of the algorithm RB [21], which tolerates Byzantine faults, and the receive STA encoding its loop body.

constraint $\mathsf{Env}_{\mathrm{CP},\mathsf{so}}$ ensures that there are exactly $n-f$ processes in the correct locations V0, V1, and exactly f processes in the send-omission locations SO0, SO1. The receive guards φ_1 and φ_2 are the syntactically same as in the rSTA for the crash-tolerant version of the algorithm FloodMin, for $k = 1$. However, the environment constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{so}}$ differs from $\mathsf{Env}_{\mathrm{EP},\mathsf{cr}}$: it restricts the number $\mathsf{nr}(m_i)$ of received messages of type m_i to a value which is less than or equal to the number of processes in locations Vi, SOi, for $i \in \{0, 1\}$.

Byzantine Faults. To model the behavior of the Byzantine-faulty processes, which can act arbitrary, no new locations and rules are introduced in the (receive) STA. Instead, the (receive) STA is used to model the behavior of the correct processes, and the effect that the Byzantine-faulty processes have on the correct ones is captured in the guards (and environment assumption). The number of messages sent by Byzantine-faulty processes is overapproximated by the parameter f, which denotes the number of faults. That is, for a message type $m \in \mathcal{M}$, the number $\#\mathsf{sent}(m) + f$ is the upper bound on the number of messages sent by correct and Byzantine-faulty processes.

The (receive) STA for Byzantine faults is used to model the behavior of the correct processes, hence $N(\boldsymbol{\pi}) = n - f$. As we do not introduce new locations or rules, we have $\mathsf{Env}_{\mathrm{CP},\mathsf{byz}} \equiv \top$. The constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{byz}}$ encodes the effect that the Byzantine-faulty processes have on the correct processes, by bounding the receive variables $\mathsf{nr}(m)$ by $\mathsf{sent}(m) + f$ from above, for $m \in \mathcal{M}$:

$$\mathsf{Env}_{\mathrm{EP},\mathsf{byz}} \equiv \bigwedge_{m \in \mathcal{M}} \mathsf{nr}(m) \leq \mathsf{sent}(m) + f$$

Figure 5 shows the pseudo code of the Byzantine reliable broadcast algorithm RB [21]. The locations $\mathcal{L} = \{\mathrm{V0}, \mathrm{V1}, \mathrm{SE}, \mathrm{AC}\}$ model the behavior of the correct processes. The location Vi encodes that a process has value $i \in \{0, 1\}$, the location SE that a process has sent an ECHO message, and the location AC that a process sets its value to 1 in line 12. There is a single message type, m_{E}, which encodes a message containing the value ECHO. There are four receive guards, $\varphi_1, \ldots, \varphi_4$. The guard φ_2, for example, checks that at least $t+1$ ECHO messages are received, capturing line 8 of the pseudo code. The set of processes that send an ECHO message is $\mathsf{sent}(m_{\mathsf{E}}) = \{\text{V1}, \text{SE}, \text{AC}\}$. The constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{byz}}$ ensures that there are not more than $\#\{\text{V1}, \text{SE}, \text{AC}\} + f$ received messages of type m_{E}.

Remark on Algorithms with a Coordinator. When modeling Byzantine-tolerant algorithms where a process acts as a coordinator (such as, e.g., the algorithm $\mathsf{PhaseQueen}$ in Fig. 1), we need to take into account that at some point, the coordinator will be Byzantine. Thus, we add locations $\mathcal{L}_{\mathsf{byz}} \subseteq \mathcal{L}$ for a single Byzantine process, disjoint from the locations that are used by the correct processes. The new locations do not encode any values of the local variables; they ensure that the Byzantine process (which may become a coordinator) moves synchronously with the other processes. In the rSTA for the algorithm $\mathsf{PhaseQueen}$ (Fig. 2), we defined $\mathcal{L}_{\mathsf{byz}} = F = \{\text{F}, \ldots, \text{R2QF}\}$. As we model the behavior of a single Byzantine process explicitly, we have $N(\boldsymbol{\pi}) = n - f + 1$.

In this case, we define the constraints $\mathsf{Env}_{\mathrm{CP},\mathsf{co}}$, which restrict the number of processes in given locations. We also identify locations $\mathcal{L}_{\mathsf{co}} \subseteq \mathcal{L}$, which only a (correct or Byzantine) coordinator is allowed to populate. The environment constraint $\mathsf{Env}_{\mathrm{CP},\mathsf{co}}$ for Byzantine-tolerant algorithms with a coordinator is:

$$\mathsf{Env}_{\mathrm{CP},\mathsf{co}} \equiv \#\mathcal{L}_{\mathsf{co}} = 1 \wedge \#\mathcal{L}_{\mathsf{byz}} = 1$$

where $\#\mathcal{L}_{\mathsf{co}} = 1$ (resp. $\#\mathcal{L}_{\mathsf{byz}} = 1$) ensures that there is exactly one process in the coordinator locations $\mathcal{L}_{\mathsf{co}}$ (resp. in the Byzantine locations $\mathcal{L}_{\mathsf{byz}}$).

Additionally, we have message types $m_{\mathsf{co}} \in \mathcal{M}$ that model the coordinator messages, and denote by ℓ_F the location where the Byzantine process performs the coordinator broadcast. The constraint $\mathsf{Env}_{\mathrm{EP},\mathsf{co}}$ states that the number of received coordinator messages of type m_{co} does not exceed the total number of coordinator messages of type m_{co} sent by the correct and Byzantine coordinators:

$$\mathsf{Env}_{\mathrm{EP},\mathsf{co}} \equiv \mathsf{Env}_{\mathrm{EP},\mathsf{byz}} \wedge \bigwedge_{m_{\mathsf{co}} \in \mathcal{M}} \mathsf{nr}(m_{\mathsf{co}}) \leq \#(\mathsf{sent}(m_{\mathsf{co}}) \cup \{\ell_F\})$$

Thus, for the algorithm $\mathsf{PhaseQueen}$, whose rSTA we depicted in Fig. 2:

$$\mathsf{Env}_{\mathrm{CP},\mathsf{co}} \equiv \#\{\text{QV0}, \ldots, \text{R2QV1}, \text{QF}, \ldots, \text{R2QF}\} = 1 \wedge \#\{\text{F}, \ldots, \text{R2QF}\} = 1$$

$$\mathsf{Env}_{\mathrm{EP},\mathsf{co}} \equiv \bigwedge_{i \in \{0,1\}} (\mathsf{nr}(m_i) \leq \#\mathsf{sent}(m_i) + f \wedge \mathsf{nr}(m_{qi}) \leq \#(\mathsf{sent}(m_{qi}) \cup \{\text{R1QF}\}))$$

4 Counter Systems

For an $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, we recall the definition of a counter system from [36]. A *counter system* w.r.t. an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ and an $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ is the tuple $\mathsf{CS}(\mathsf{STA}, \mathbf{p}) = (\Sigma(\mathbf{p}), I(\mathbf{p}), R(\mathbf{p}))$, representing a system of $N(\mathbf{p})$ processes whose behavior is modeled using the STA, where $\Sigma(\mathbf{p})$ is the set of *configurations*, $I(\mathbf{p})$ is the set of *initial configurations*, and $R(\mathbf{p})$ is the *transition relation*.

A *configuration* $\sigma \in \Sigma(\mathbf{p})$ is a tuple $(\boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{p} \in \mathbf{P}_{RC}$ is an admissible valuation, and $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$ is an $|\mathcal{L}|$-dimensional vector of *counters*, such that $\sigma \models \mathsf{Env}$. For every $\sigma \in \Sigma(\mathbf{p})$, we have $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = N(\mathbf{p})$. This follows from $\sigma \models \mathsf{Env}$, in particular from $\sigma \models \#\mathcal{L} = N(\boldsymbol{\pi})$, the definition of $N(\mathbf{p})$, and the semantics of the *c*-propositions. A configuration $\sigma \in \Sigma(\mathbf{p})$ is *initial*, i.e., $\sigma \in I(\mathbf{p}) \subseteq \Sigma(\mathbf{p})$, iff $\sigma.\boldsymbol{\kappa}[\ell] = 0$, for every $\ell \in \mathcal{L} \setminus \mathcal{I}$. That is, the value $\sigma.\boldsymbol{\kappa}[\ell]$ of the counter for each non-initial location $\ell \in \mathcal{L} \setminus \mathcal{I}$ is set to 0 in $\sigma \in \mathcal{I}$.

To define the transition relation $R(\mathbf{p})$, we first define the notion of a transition. A *transition* is a function $tr : \mathcal{R} \to \mathbb{N}$ that maps each rule $r \in \mathcal{R}$ to a *factor* $tr(r) \in \mathbb{N}$. Given a valuation $\mathbf{p}$ of $\boldsymbol{\pi}$, the set $Tr(\mathbf{p}) = \{tr \mid \sum_{r \in \mathcal{R}} tr(r) = N(\mathbf{p})\}$ contains transitions whose factors sum up to $N(\mathbf{p})$. For a transition tr and a rule $r \in \mathcal{R}$, the factor $tr(r)$ denotes the number of processes that act upon this rule. By restricting the set $Tr(\mathbf{p})$ to contain transitions whose factors sum up to $N(\mathbf{p})$, we ensure that in a transition, every process takes a step. This captures the semantics of synchronous computation. A transition $tr \in Tr(\mathbf{p})$ is *enabled* in a tuple $(\boldsymbol{\kappa}, \mathbf{p})$, where $\boldsymbol{\kappa}$ is an $|\mathcal{L}|$- dimensional vector of counters and $\mathbf{p} \in \mathbf{P}_{RC}$ an admissible valuation, iff for every $r \in \mathcal{R}$, such that $tr(r) > 0$, it holds that $(\boldsymbol{\kappa}, \mathbf{p}) \models r.\varphi$, and for every $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{from} = \ell} tr(r)$. The former condition ensures that processes only use rules whose guards are satisfied, and the latter that every process moves in an enabled transition.

Given a transition $tr \in Tr(\mathbf{p})$, we define the *origin* $o(tr) = (\boldsymbol{\kappa}, \mathbf{p})$ of tr, where for every location $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{from} = \ell} tr(r)$, and the *goal* $g(tr) = (\boldsymbol{\kappa}', \mathbf{p})$ of tr, where for every location $\ell \in \mathcal{L}$, we have $\boldsymbol{\kappa}'[\ell] = \sum_{r \in \mathcal{R} \wedge r.\mathit{to} = \ell} tr(r)$. The origin $o(tr)$ is the unique tuple $(\boldsymbol{\kappa}, \mathbf{p})$ where the transition tr is enabled, while its goal $g(tr)$ is the unique tuple $(\boldsymbol{\kappa}', \mathbf{p})$ that is obtained by applying the transition tr to its origin $o(tr)$. The *transition relation* $R(\mathbf{p})$ is the relation $R(\mathbf{p}) \subseteq \Sigma(\mathbf{p}) \times Tr(\mathbf{p}) \times \Sigma(\mathbf{p})$, such that $\langle \sigma, tr, \sigma' \rangle \in R(\mathbf{p})$ iff $\sigma = o(tr)$ is the origin and $\sigma' = g(tr)$ is the goal of the transition tr.

5 Synchronous Transition Systems

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, and $\mathbf{p} \in \mathbf{P}_{RC}$ an admissible valuation of the parameter vector $\boldsymbol{\pi}$. A *synchronous transition system* (or *system*), w.r.t. an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$ and an rSTA is the triple $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p}) = \langle S(\mathbf{p}), S_0(\mathbf{p}), T(\mathbf{p}) \rangle$, representing a system of $N(\mathbf{p})$ processes whose behavior is modeled using the rSTA, where $S(\mathbf{p})$ is the set of *states*, $S_0(\mathbf{p})$ is the set of *initial states*, and $T(\mathbf{p})$ is the *transition relation*.

Recall that the environment assumption Env^{Δ} of the rSTA is the conjunction $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}_{\mathrm{CP}} \wedge \mathsf{Env}_{\mathrm{EP}}$. A *state* $s \in S(\mathbf{p})$ is a tuple $s = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$, where $\boldsymbol{\ell} \in \mathcal{L}^{N(\mathbf{p})}$ is an $N(\mathbf{p})$-dimensional vector of locations, and $\mathbf{nr}_i \in \mathbb{N}^{|\mathcal{M}|}$, for $1 \le i \le N(\mathbf{p})$, is a vector of valuations of the receive variables $\mathsf{nr}(m)$, with $m \in \mathcal{M}$, for each process i, such that $s \models \mathsf{Env}_{\mathrm{CP}}$. In a state $s \in S(\mathbf{p})$, the vector $\boldsymbol{\ell}$ of locations is used to store the current location $s.\boldsymbol{\ell}[i] \in \mathcal{L}$ for each process i, while the vector $\mathbf{nr}_i \in \mathbb{N}^{|\mathcal{M}|}$ stores the values of the receive variables for each process i, with $1 \le i \le N(\mathbf{p})$. Further, each state $s \in S(\mathbf{p})$ satisfies $\mathsf{Env}_{\mathrm{CP}}$.

To formally define that a state $s \in S(\mathbf{p})$ satisfies the environment constraint $\mathsf{Env}_{\mathrm{CP}}$, we define the semantics of c-propositions w.r.t. states $s \in S(\mathbf{p})$. Let $\mathsf{counters}_{\mathbf{p}} : S(\mathbf{p}) \times \mathcal{L} \rightarrow \mathbb{N}$ denote a mapping that maps a state $s \in S(\mathbf{p})$ and a location $\ell \in \mathcal{L}$ to the number of processes that are in location ℓ in the state s, that is, $\mathsf{counters}_{\mathbf{p}}(s, \ell) = |\{i \mid 1 \leq i \leq N(\mathbf{p}) \wedge s.\boldsymbol{\ell}[i] = \ell\}|$. Further, let $\boldsymbol{\kappa}(s) \in \mathbb{N}^{|\mathcal{L}|}$ denote the $|\mathcal{L}|$-dimensional vector of counters w.r.t. the state $s \in S(\mathbf{p})$, where for every location $\ell \in \mathcal{L}$, we have that $\boldsymbol{\kappa}(s)[\ell]$ stores the number of processes that are in location ℓ in the state s, that is, $\boldsymbol{\kappa}(s)[\ell] = \mathsf{counters}_{\mathbf{p}}(s, \ell)$. We say that $s \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$ iff $(\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \#L \geq \boldsymbol{a} \cdot \boldsymbol{\pi} + b$. A state $s \in S(\mathbf{p})$ satisfies the environment constraints $\mathsf{Env}_{\mathrm{CP}}$, that is, $s \models \mathsf{Env}_{\mathrm{CP}}$ iff $(\boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}_{\mathrm{CP}}$.

In an *initial state* $s_0 \in S_0(\mathbf{p})$, the vector $\boldsymbol{\ell}$ of locations stores only initial locations, i.e., $\boldsymbol{\ell}[i] \in \mathcal{I}$, for $1 \leq i \leq N(\mathbf{p})$, and all receive variables of all processes are initialized to 0. Formally, a state $s_0 = \langle \boldsymbol{\ell}, \mathbf{nr}_1, \ldots, \mathbf{nr}_{N(\mathbf{p})}, \mathbf{p} \rangle$ is *initial*, i.e., $s_0 \in S_0(\mathbf{p})$, if $s_0.\boldsymbol{\ell} \in \mathcal{I}^{N(\mathbf{p})}$ and $s_0.\mathbf{nr}_i[m] = 0$, for $1 \leq i \leq N(\mathbf{p})$ and $m \in \mathcal{M}$.

We now define the transition relation $T(\mathbf{p}) \subseteq S(\mathbf{p}) \times S(\mathbf{p})$, where we will use the environment constraint $\mathsf{Env}_{\mathrm{EP}}$ to restrict the values of the receive variables. A transition $(s, s') \in T(\mathbf{p})$ encodes one round in the execution of the distributed algorithm. In a round, the processes send and receive messages, and update their variables based on the received messages. Further, all the messages sent in the current round are received in the same round. The process variable updates are captured by moving processes from one location to another, based on the values of the receive variables. The *transition relation* $T(\mathbf{p})$ is a binary relation $T(\mathbf{p}) \subseteq S(\mathbf{p}) \times S(\mathbf{p})$, where $(s, s') \in T(\mathbf{p})$ iff for every process i, with $1 \leq i \leq N(\mathbf{p})$:

1. $0 \leq s'.\mathbf{nr}_i[m] \leq N(\mathbf{p})$, such that $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models \mathsf{Env}_{\mathrm{EP}}$, for $m \in \mathcal{M}$,
2. there exists $r^{\Delta} \in \mathcal{R}^{\Delta}$ such that:
   - $s.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{from}$,
   - $(s'.\mathbf{nr}_i, \boldsymbol{\kappa}(s), s.\mathbf{p}) \models r^{\Delta}.\varphi$,
   - $s'.\boldsymbol{\ell}[i] = r^{\Delta}.\mathit{to}$.
3. $s'.\mathbf{p} = s.\mathbf{p}$ and $s' \models \mathsf{Env}_{\mathrm{CP}}$.

In a transition $(s, s') \in T(\mathbf{p})$, the receive variables and locations of each process are updated. That is, the value $s'.\mathbf{nr}_i[m]$ of the receive variable $\mathsf{nr}(m)$ of process i is assigned a value in the range from 0 to $N(\mathbf{p})$ non-deterministically, such that the environment constraint $\mathsf{Env}_{\mathrm{EP}}$ is satisfied. This ensures that the number of received messages of type m is non-negative, that it does not exceed the number of participating processes, and that the receive variables of each process are assigned values that satisfy the constraints of the environment assumption. In the case of the synchronous computation model, this captures that all messages sent by correct processes in a round are received in the same round, and that the number of messages of type m, received by process i, is bounded by above by the total number of messages of type m, sent by both correct and faulty processes. To update the locations, each process i picks a rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ that it applies to update its location, if the process i is in location $r^{\Delta}.\mathit{from}$ in the state s, and if the newly assigned values of the receive variables of process i in the state s' satisfy the receive guard $r^{\Delta}.\varphi$. If this is the case, the process i updates its location to $r^{\Delta}.to$ in the state s'. The parameter values remain unchanged, and we require that the state s' satisfies $\mathsf{Env}_{\mathrm{CP}}$, i.e., it is a valid state.

6 Abstracting rSTA to STA

Given an rSTA, our goal is to construct an STA, which differs from the rSTA only in the guards on its rules and the environment assumption. For each rule $r^{\Delta} \in \mathcal{R}^{\Delta}$ in the rSTA, whose guard $r^{\Delta}.\varphi$ is a receive guard, we will construct a rule $r \in \mathcal{R}$ in the STA, such that the guard $r.\varphi$ is a Boolean combination of c-propositions. We will perform the abstraction in two steps: (i) we will strengthen each receive guard $r^{\Delta}.\varphi$, occurring on the rules $r^{\Delta} \in \mathcal{R}^{\Delta}$ of the rSTA, with the constraints imposed by the faulty environment and the synchronous computation model, encoded in the environment assumption Env^{Δ}, and (ii) we will eliminate the receive variables from the receive guards and environment assumptions of rSTA to obtain the guards and environment assumption of STA.

6.1 Guard Strengthening

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, where the rules $r^{\Delta} \in \mathcal{R}^{\Delta}$ have guards containing expressions over the receive variables $\mathsf{nr}(m) \in \Delta$, and where the environment assumption $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}_{\mathrm{CP}} \wedge \mathsf{Env}_{\mathrm{EP}}$ is a conjunction of two environment constraints, $\mathsf{Env}_{\mathrm{CP}}$ and $\mathsf{Env}_{\mathrm{EP}}$, where the latter restricts the values of the receive variables. Recall that in Sect. 3.2, we defined different environment constraints $\mathsf{Env}_{\mathrm{EP}}$ for the different fault models. In general, these constraints express that for each message type $m \in \mathcal{M}$, the receive variable $\mathsf{nr}(m)$ is assigned a value which is greater or equal to the number of messages of type m sent by correct processes, and which is smaller or equal to the total number of messages of type m, sent by both correct and faulty processes (e.g., $\#\mathsf{sent}(m) \leq \mathsf{nr}(m) \leq \#\mathsf{sent}(m) + \#\mathsf{sent}_{\mathsf{cr}}(m)$ for crash faults). As a first step towards eliminating the receive variables from the receive guards, we strengthen the rules from the set $\mathcal{R}^{\Delta}$, such that we add the environment constraints $\mathsf{Env}_{\mathrm{EP}}$ to their guards in order to bound the values of the receive variables.

Definition 1. *Given $r^{\Delta} \in \mathcal{R}^{\Delta}$, its* strengthened rule *is $\widehat{r}^{\Delta} = \mathsf{strengthen}(r^{\Delta})$, such that: $\widehat{r}^{\Delta}.from = r^{\Delta}.from$, $\widehat{r}^{\Delta}.to = r^{\Delta}.to$, $\widehat{r}^{\Delta}.\varphi = r^{\Delta}.\varphi \wedge \mathit{Env}_{\mathrm{EP}}$.*

We denote by $\widehat{\mathcal{R}}^{\Delta} = \{\mathsf{strengthen}(r^{\Delta}) \mid r^{\Delta} \in \mathcal{R}^{\Delta}\}$ the set of strengthened rules in $\mathit{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathit{Env}^{\Delta})$, where $\mathit{Env}^{\Delta} \equiv \mathit{Env}_{\mathrm{CP}} \wedge \mathit{Env}_{\mathrm{EP}}$.

6.2 Eliminating the Receive Variables

Let $\mathsf{rSTA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}^{\Delta}, \Delta, \Pi, RC, \mathsf{Env}^{\Delta})$ be a receive STA, and let $\widehat{\mathcal{R}}^{\Delta}$ be the set of strengthened rules (Definition 1). We define an $\mathsf{STA} = (\mathcal{L}, \mathcal{I}, \mathcal{R}, \Pi, RC, \mathsf{Env})$ whose locations, initial locations, and parameters are the same as in rSTA, while we construct the rules $\mathcal{R}$ and the environment assumption Env of the STA below.

Recall that $\mathsf{Env}^{\Delta} \equiv \mathsf{Env}_{\mathrm{CP}} \wedge \mathsf{Env}_{\mathrm{EP}}$. To define the environment assumption Env of the constructed STA, we set $\mathsf{Env} \equiv \mathsf{Env}_{\mathrm{CP}}$. Before we define the rules of the constructed STA, we define the mapping $\mathsf{eliminate}$.

Definition 2. *Let ϕ be a propositional formula over r-, c-, and e-propositions. Let $\boldsymbol{\delta} = \langle \mathsf{nr}(m_1), \ldots, \mathsf{nr}(m_{|\mathcal{M}|})\rangle$ denote the $|\mathcal{M}|$-dimensional* receive variables vector*, and* QE *denote the quantifier elimination procedure for Presburger arithmetic. The formula* $\mathsf{eliminate}(\phi) = \mathsf{QE}(\exists \boldsymbol{\delta}\ \phi)$ *is a quantifier-free formula, with no occurrence of receive variables* $\mathsf{nr}(m) \in \Delta$*, which is logically equivalent to* $\exists \boldsymbol{\delta}\ \phi$*.*

To construct a rule $r \in \mathcal{R}$ of an STA, given a rule $r^\Delta \in \mathcal{R}^\Delta$ of an rSTA, we will apply the mapping eliminate to each guard of the strengthened rule $\widehat{r}^\Delta \in \widehat{\mathcal{R}}^\Delta$, where $\widehat{r}^\Delta = \mathsf{strengthen}(r^\Delta)$. The result of quantifier elimination is a quantifier-free formula over c-propositions, which is logically equivalent to $\exists \boldsymbol{\delta}\ \widehat{r}^\Delta.\varphi$.

Definition 3. *Given $r^\Delta \in \mathcal{R}^\Delta$, its* constructed rule *is* $r = \mathsf{construct}(r^\Delta) \in \mathcal{R}$*, such that: r.from = r^Δ.from, r.to = r^Δ.to, r.φ =* $\mathsf{eliminate}(\widehat{r}^\Delta.\varphi)$*, where* $\widehat{r}^\Delta = \mathsf{strengthen}(r^\Delta)$*.*

Proposition 1. *For every strengthened rule $\widehat{r}^\Delta \in \widehat{\mathcal{R}}^\Delta$ and every tuple $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p})$, where $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$, $\boldsymbol{\kappa} \in \mathbb{N}^{|\mathcal{L}|}$, and $\mathbf{p} \in \mathbf{P}_{RC}$, we have:*

$$(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \widehat{r}^\Delta.\varphi \quad \textit{implies} \quad (\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^\Delta.\varphi)$$

Proposition 1 is a consequence of quantifier elimination. Note that the converse of this proposition does not hold in general. That is, $(\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^\Delta.\varphi)$ does not imply that $(\mathbf{d}, \boldsymbol{\kappa}, \mathbf{p}) \models \widehat{r}^\Delta.\varphi$, for every $\mathbf{d} \in \mathbb{N}^{|\mathcal{M}|}$. However, by quantifier elimination, we have that $(\boldsymbol{\kappa}, \mathbf{p}) \models \mathsf{eliminate}(\widehat{r}^\Delta.\varphi)$ implies $(\boldsymbol{\kappa}, \mathbf{p}) \models \exists \boldsymbol{\delta}\ \widehat{r}^\Delta.\varphi$.

6.3 Soundness and Completeness

This construction of an STA is sound and complete. That is, given a rSTA and an admissible valuation $\mathbf{p} \in \mathbf{P}_{RC}$, we show that there exists a bisimulation relation between the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$, induced by rSTA and $\mathbf{p}$, and a counter system $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, induced by the constructed STA and $\mathbf{p}$. The existence of a bisimulation implies that $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ and $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$ satisfy the same CTL^* formulas [3]. To express temporal formulas, as atomic propositions we use the c-propositions from the set CP. We define two labeling functions, $\lambda_{S(\mathbf{p})}$ and $\lambda_{\Sigma(\mathbf{p})}$, where $\lambda_{S(\mathbf{p})} : S(\mathbf{p}) \to 2^{\mathrm{CP}}$ assigns to a state $s \in S(\mathbf{p})$ the set of c-propositions that hold in it (the function $\lambda_{\Sigma(\mathbf{p})} : \Sigma(\mathbf{p}) \to 2^{\mathrm{CP}}$ is defined analogously).

We introduce an *abstraction mapping* $\alpha_{\mathbf{p}} : S(\mathbf{p}) \to \Sigma(\mathbf{p})$ that maps states $s \in S(\mathbf{p})$ of $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$ to configurations $\sigma \in \Sigma(\mathbf{p})$ of $\mathsf{CS}(\mathsf{STA}, \mathbf{p})$, such that $\sigma = \alpha_{\mathbf{p}}(s)$ iff $\sigma = (\boldsymbol{\kappa}(s), s.\mathbf{p})$. By the definition of the abstraction mapping $\alpha_{\mathbf{p}}$ and the semantics of c-propositions, we have that a state and its abstraction satisfy the same c-propositions. Further, given a configuration $\sigma \in \Sigma(\mathbf{p})$, we can construct a state $s \in S(\mathbf{p})$, such that $\sigma = \alpha_{\mathbf{p}}(s)$. While this is always possible, the constructed state s might not be reachable in any execution of the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$. However, we can use the constraint $\mathsf{Env}_{\mathrm{EP}}$ to restrict the value of the receive variables in the constructed state s, such that it is a valid state in the system $\mathsf{STS}(\mathsf{rSTA}, \mathbf{p})$. The main result of this section is stated below. The detailed proof of this result can be found in the first author's PhD thesis.

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

7 Experimental Evaluation

To show the usefulness of translating rSTA to STA, we: (i) encoded synchronous fault-tolerant distributed algorithms using rSTA, (ii) implemented the method from Sect. 6 in a prototype, (iii) compared the output to the existing manual encodings from [34], some of which are artifacts of the experimental evaluation from [36] and were given as examples throughout this paper, and (iv) verified the properties of the generated STA using the technique from [36].

Encoding Algorithms as rSTA. We extended the STA encoding from [36], to support (i) declarations of receive variables and (ii) constraints given by the environment assumption. The algorithms we encoded are listed in Table 1, and their rSTA can be found in [35]. For each of them, there already existed a manually produced STA [34]. The manually produced rSTA and STA have the same structure w.r.t. locations and rules, and differ only in the guards that occur on the rules: in the rSTA, we have receive guards, which are Boolean combinations of r-propositions and c-propositions, while in the manually encoded STA, the guards are Boolean combinations of c-propositions.

Applying Quantifier Elimination. We implemented a script that parses the input rSTA and creates an STA whose rules have guards that are Boolean combinations of c-propositions, according to the abstraction from Sect. 6. To automate the quantifier elimination step, we applied Z3 [16] tactics for quantifier elimination [10,11], to formulas of the form $\exists \boldsymbol{\delta}\ \widehat{r}^{\Delta}.\varphi$, where $\widehat{r}^{\Delta}.\varphi \equiv r^{\Delta}.\varphi \wedge \mathsf{Env}_{\mathrm{EP}}$ is the strengthened guard of the receive guard $r^{\Delta}.\varphi$, for $r^{\Delta} \in \mathcal{R}^{\Delta}$. For all our benchmarks, the STA is generated within seconds, as reported in Table 1.

Analyzing the Automatically Generated STA. We compared the guards of the automatically generated STA (autoSTA) to the manually encoded STA (manSTA). Syntactically, the guards of autoSTA are larger in general, as they contain additional constraints that result from quantifier elimination. Semantically, we check whether the guards for the autoSTA imply the guards of the manSTA. For each automatically generated guard φ_{auto}, we check whether its corresponding guard φ_{man} from the manual encoding is implied by φ_{auto}, for all values of the parameters and number of sent messages by checking the validity of the formula:

$$\forall \mathbf{p} \in \mathbf{P}_{RC}\ \forall L_1 \ldots \forall L_{|\mathcal{M}|}\ \varphi_{\mathsf{auto}}(L_1, \ldots, L_{|\mathcal{M}|}) \rightarrow \varphi_{\mathsf{man}}(L_1, \ldots, L_{|\mathcal{M}|}) \quad (3)$$

where $L_j = \mathsf{sent}(m_j)$, for $m_j \in \mathcal{M}$ and $1 \leq j \leq |\mathcal{M}|$, denotes the set of locations where processes send messages of type m_j. We automate the validity check of (3) using an SMT solver, such as Z3, to check the unsatisfiability of its negation. With this check we are able to either verify that the earlier manSTA faithfully model the benchmark algorithms, or detect discrepancies, which we investigated

Table 1. The algorithms we encoded as rSTA and the results of applying the verification technique from [36]. The column QE states the time needed to produce an autoSTA from an rSTA. The column ⇒ states if (3) is valid all, some, or none of guards. We report on the time it took the solvers Z3 and CVC4 to (i) check the guard implications (only Z3), (ii) compute the diameter for the autoSTA, and (iii) check the safety properties of the autoSTA, (iv) compute the diameter for the manSTA, (v) check the safety properties of the manSTA, using the SMT-based procedure from [36].

			(i)		(ii) autoSTA		(iii) autoSTA		(iv) manSTA		(v) manSTA	
algorithm	QE	⇒	⇒ time	d	d time		BMC time		d time		BMC time	
	Z3		Z3		Z3	CVC4	Z3	CVC4	Z3	CVC4	Z3	CVC4
RB	0.16s	all	0.18s	2	0.09s	0.26s	0.03s	0.03s	0.07s	0.27s	0.02s	0.03s
HybridRB	0.39s	all	0.41s	2	0.14s	0.75s	0.03s	0.06s	0.09s	0.67s	0.03s	0.05s
OmitRB	0.34s	all	0.36s	2	0.11s	0.69s	0.03s	0.05s	0.09s	0.67s	0.02s	0.04s
FairCons	0.25s	all	0.44s	2	0.17s	2.82s	0.07s	0.16s	0.14s	2.68s	0.06s	0.14s
FloodMin, $k = 1$	0.10s	all	0.19s	2	0.07s	0.25s	0.06s	0.11s	0.06s	0.25s	0.06s	0.09s
FloodMin, $k = 2$	0.26s	all	0.35s	2	0.13s	1.72s	0.07s	0.19s	0.15s	2.22s	0.06s	0.17s
FMinOmit, $k = 1$	0.10s	all	0.13s	1	0.03s	0.03s	0.01s	0.01s	0.06s	0.04s	0.01s	0.01s
FMinOmit, $k = 2$	0.27s	all	0.26s	1	0.05s	0.08s	0.01s	0.03s	0.05s	0.08s	0.01s	0.03s
FloodSet	0.20s	all	0.31s	2	0.11s	0.71s	0.07s	0.17s	0.10s	0.90s	0.06s	0.15s
kSetOmit, $k = 1$	0.59s	all	0.52s	3	2.71s	53.36s	0.22s	0.85s	1.09s	1m8s	0.23s	0.81s
kSetOmit, $k = 2$	1.43s	all	1.18s	–	t.o.	t.o.	–	–	t.o.	t.o.	–	–
PhaseKing	1.19s	all	1.57s	4	3.53s	16.51s	0.24s	1.57s	3.67s	15.80s	0.25s	1.47s
ByzKing	1.16s	all	1.58s	4	1.92s	1m19s	0.27s	1.97s	3.73s	38.50s	0.24s	2.26s
HybridKing	3.59s	some	3.03s	4	0.33s	6.34s	0.18s	1.11s	t.o.	t.o.	–	–
OmitKing	3.09s	all	2.79s	4	0.26s	6.12s	0.15s	0.91s	1h15m	t.o.	9.08s	1m27s
PhaseQueen	0.42s	all	0.90s	3	0.37s	4.46s	0.04s	0.61s	0.40s	4.72s	0.06s	0.50s
ByzQueen	0.42s	all	0.91s	3	0.39s	17.15s	0.09s	0.58s	0.53s	10.6s	0.08s	0.61s
HybridQueen	1.34s	some	1.77s	3	0.13s	2.04s	0.05s	0.37s	t.o.	t.o.	–	–
OmitQueen	1.13s	all	1.56s	3	0.13s	2.18s	0.20s	0.46s	0.57s	8.87s	0.27s	1.21s

further. Our translation technique produces the strongest possible guards, due to the soundness and completeness result. Hence, we expected that the implication holds for all the guards of all the benchmarks we considered. This is however not the case for the algorithms HybridKing and HybridQueen which are designed to tolerate hybrid faults, in particular, send omissions and Byzantine faults. There, we found that one automatically generated guard does not imply its corresponding manual guard, and concluded that this is due to a flaw in the manual encoding by manual inspection. We found a similar problem with a missing rule in the (purely) Byzantine versions of these algorithms, namely ByzKing and ByzQueen. By adding these rules and correcting the appropriate manual guards, we were able to establish the validity of (3) for all guards.

Model Checking of Safety Properties. We gave the STA we obtained as output of our translation procedure as input to the bounded model checking tool from [36], which computes a diameter of a counter system and performs bounded model checking for safety properties. The experiments were run on a machine with 2.8 GHz Quad-Core Intel(R) Core(TM) i7 CPU and 16GB. The results of applying the SMT-based procedure from [36] to the autoSTA, as well as to the extended set [34] of manSTA from [36], are presented in Table 1. The timeout, denoted by t.o. in the table, was set to 24 h. For all algorithms, we note that bounded model checking with both Z3 and CVC4 performs similarly for both autoSTA and manSTA. For computing the diameter, we observe that for the algorithms: RB [21] (Fig. 5), HybridRB, OmitRB [9], FairCons [33], FloodMin, for $k = 1$ (Fig. 3) and $k = 2$ [28], FMinOmit, for $k = 1$ (Fig. 4) and $k = 2$ [28], kSetOmit, for $k = 2$ [33], FloodSet [28], PhaseKing [7], and PhaseQueen [6] (Fig. 1), we obtain comparable results on both the autoSTA and manSTA. For the other algorithms, we found:

- computing the diameter for the autoSTA of kSetOmit, with $k = 1$ [33], is slightly slower with Z3 and slightly faster with CVC4 than for the manSTA;
- Z3 performs better when computing the diameter for the autoSTA than for the manSTA of both ByzKing and ByzQueen [9], while CVC4 performs worse. Note that in Table 1 we report the times for the manSTA of ByzKing and ByzQueen that have missing rules. After adding the rules to the manSTA, computing the diameter on the autoSTA is still faster with both solvers;
- Z3 and CVC4 compute the diameter for the autoSTA of HybridKing and HybridQueen [9] within seconds, in contrast to both timing out for the manSTA;
- computing the diameter with Z3 is significantly faster for the autoSTA than for the manSTA of OmitKing [9]. CVC4 computes the diameter for autoSTA of OmitKing, while for manSTA it times out. The computed diameter $\mathsf{d} = 4$ for autoSTA is smaller than the diameter 8, computed for manSTA;
- Z3 and CVC4 compute the diameter for the autoSTA of OmitQueen [9] faster than for manSTA.

8 Conclusions

We established a fully automated pipeline that for a synchronous distributed algorithm: (1) starts from a formal model that captures its pseudo code, (2) produces a formal model suitable for verification, and (3) automatically verifies its safety properties. Our technique thus closes the gap between the original description of an algorithm (using received messages) and the synchronous threshold automaton of the algorithm given as an input to a verification tool.

There are two major differences to the asynchronous case considered in [37]. First, the asynchronous model uses interleaving semantics, while in the synchronous model all processes take a step in a transition. Second, in the asynchronous model, there are no limitations when a message will be delivered. The lower bound on the number of received messages, given in the synchronous model by the number of sent messages by correct processes, is only *eventually* satisfied in the asynchronous model, and thus is not used in the process of eliminating the receive variables from the receive guards.

We did extensive experimental evaluation of our method. We attribute the better performance of the bounded model checking technique from [36] on the automatically generated STA to the fact that the automatically generated guards contain more additional constraints, coming from the environment assumption, which help guide the SMT solvers. Moreover, not only do we obtain the diameter bounds faster, we also obtain better bounds for the automatically generated STA of some benchmarks. These findings confirm the conjecture that manual encoding of distributed algorithms is a tedious and error-prone task and suggest that there is a real benefit of producing guards automatically.

References

1. Aminof, B., Rubin, S., Stoilkovska, I., Widder, J., Zuleger, F.: Parameterized model checking of synchronous distributed algorithms by abstraction. VMCAI 2018. LNCS, vol. 10747, pp. 1–24. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_1
2. Attiya, H., Welch, J.: Distributed Computing, 2nd edn. Wiley, Hoboken (2004)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MITP, United States (2008)
4. Bakst, A., von Gleissenthall, K., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL **1**(OOPSLA), 1–27 (2017)
5. Balasubramanian, A.R., Esparza, J., Lazić, M.: Complexity of verification and synthesis of threshold automata. In: ATVA (2020)
6. Berman, P., Garay, J.A., Perry, K.J.: Asymptotically Optimal Distributed Consensus. Technical report, Bell Labs (1989). http://plan9.bell-labs.co/who/garay/asopt.ps
7. Berman, P., Garay, J.A., Perry, K.J.: Towards optimal distributed consensus (Extended Abstract). In: FOCS, pp. 410–415 (1989)
8. Bertrand, N., Konnov, I., Lazić, M., Widder, J.: Verification of randomized consensus algorithms under round-rigid adversaries. In: CONCUR, pp. 1–15 (2019)
9. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theor. Comput. Sci. **412**(40), 5602–5630 (2011)
10. Bjørner, N.: Linear quantifier elimination as an abstract decision procedure. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 316–330. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_27
11. Bjørner, N., Janota, M.: Playing with quantified satisfaction. LPAR **35**, 15–27 (2015)
12. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 372–391. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_23
13. Chaouch-Saad, M., Charron-Bost, B., Merz, S.: A reduction theorem for the verification of round-based distributed algorithms. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 93–106. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04420-5_10
14. Cooper, D.C.: Theorem proving in arithmetic without multiplication. Mach. Intell. **7**(91–99), 300 (1972)
15. Damian, A., Drăgoi, C., Militaru, A., Widder, J.: Communication-closed asynchronous protocols. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 344–363. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_20

16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
17. Drăgoi, C., Henzinger, T.A., Veith, H., Widder, J., Zufferey, D.: A logic-based framework for verifying consensus algorithms. In: McMillan, K.L., Rival, X. (eds.) VMCAI 2014. LNCS, vol. 8318, pp. 161–181. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54013-4_10
18. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
19. Gleissenthall, K.V., Gökhan Kici, R., Bakst, A., Stefan, D., Jhala, R.: Pretend synchrony. In: POPL (2019)
20. Hawblitzel, C., et al.: Ironfleet: proving safety and liveness of practical distributed systemsp. Commun. ACM **60**(7), 83–92 (2017)
21. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. J. ACM **34**(3), 626–645 (1987)
22. Konnov, I., Lazić, M., Veith, H., Widder, J.: A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. In: POPL, pp. 719–734 (2017)
23. Konnov, I., Veith, H., Widder, J.: On the completeness of bounded model checking for threshold-based distributed algorithms: reachability. Inf. Comput. **252**, 95–109 (2017). https://doi.org/10.1016/j.ic.2016.03.006
24. Kopetz, H., Grünsteidl, G.: TTP - a protocol for fault-tolerant real-time systems. IEEE Comput. **27**(1), 14–23 (1994). https://doi.org/10.1109/2.248873
25. Kragl, B., Qadeer, S., Henzinger, T.A.: Synchronizing the asynchronous. In: CONCUR, pp. 1–17 (2018)
26. Kukovec, J., Konnov, I., Widder, J.: Reachability in parameterized systems: all flavors of threshold automata. In: CONCUR. LIPIcs, vol. 118, pp. 1–17 (2018)
27. Lincoln, P., Rushby, J.: A formally verified algorithm for interactive consistency under a hybrid fault model. In: FTCS, pp. 402–411 (1993)
28. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
29. Marić, O., Sprenger, C., Basin, D.: Cutoff bounds for consensus algorithms. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 217–237. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_12
30. 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, pp. 92–101 (1929)
31. Pugh, W.: A practical algorithm for exact array dependence analysis. Commun. ACM **35**(8), 102–114 (1992)
32. Rahli, V., Guaspari, D., Bickford, M., Constable, R.L.: Formal specification, verification, and implementation of fault-tolerant systems using EventML. ECEASST **72** (2015)
33. Raynal, M.: Fault-tolerant agreement in synchronous message-passing systems. Synth. Lect. Distrib. Comput. Theory **1**(1), 1–189 (2010)
34. Stoilkovska, I.: Manually Encoded Synchronous Threshold Automata. https://github.com/istoilkovska/syncTA/algorithms. Accessed Oct 2020
35. Stoilkovska, I.: Receive Synchronous Threshold Automata. https://github.com/istoilkovska/syncTA/receiveSTA. Accessed Oct 2020
36. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying safety of synchronous fault-tolerant algorithms by bounded model checking. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 357–374. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_20

37. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Eliminating message counters in threshold automata. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 196–212. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_11
38. Wilcox, J.R., et al.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI, pp. 357–368 (2015)