Cutoffs for Symmetric Point-to-Point Distributed Algorithms

Thanh-Hai Tran[1], Igor Konnov[2(✉)], and Josef Widder[2]

[1] TU Wien, Vienna, Austria
[2] Informal Systems, Vienna, Austria
igor@informal.systems

Abstract. Distributed algorithms are typically parameterized in the number of participants. While in general, parameterized verification is undecidable, many distributed algorithms such as mutual exclusion, cache coherence, and distributed consensus enjoy the cutoff property, which reduces the parameterized verification problem to verification of a finite number of instances. Failure detection algorithms do not fall into one of the known classes. While consensus algorithms, for instance, are quorum-based, failure detectors typically rely on point-to-point communication and timeouts. In this paper, we formalize this communication structure and introduce the class of symmetric point-to-point algorithms. We show that the symmetric point-to-point algorithms have a cutoff. As a result, one can verify them by model checking small instances. We demonstrate the feasibility of our approach by specifying the failure detector by Chandra and Toueg in TLA$^+$, and by model checking them with the TLC and the APALACHE model checkers.

Keywords: TLA$^+$ · Parameterized model checking · Failure detectors · Symmetry · Point-to-point communication

1 Introduction

Nowadays, many high-reliability systems are distributed and parameterized in some manner, e.g. the number of participants, or the size of message buffers. Since the number and the cost of failures of these systems increases [2], industry has applied many automated techniques to reason about their correctness at the design and implementation levels, such as model checking [6,17,24,28], and testing [19]. While these methods report positive results in analyzing individual system configurations with fixed parameter values, the real goal is to verify *all* configurations, i.e., with infinitely many vectors of parameter values.

Unfortunately, the parameterized verification problem is typically undecidable, even if every participant follows the same code [1,3,27]. This negative result has led naturally to two approaches of algorithm analyses: (a) semi-automated

Supported by Interchain Foundation (Switzerland) and the Austrian Science Fund (FWF) via the Doctoral College LogiCS W1255.

C. Georgiou and R. Majumdar (Eds.): NETYS 2020, LNCS 12129, pp. 329–346, 2021.
https://doi.org/10.1007/978-3-030-67087-0_21

methods based on user-guided invariants and proof assistants, and (b) automatic techniques for restricted classes of algorithms and properties. A particularly fascinating case is the cutoff property that guarantees that analyzing a few small instances is sufficient to reason about the correctness of all instances [8,15]. In a nutshell, given a property ξ and a system that has a parameter $\mathbf{m}$, there exists a number $B \geq 1$ such that whenever all instances that assign a value not greater than B to a parameter $\mathbf{m}$ satisfy ξ, then all instances which assign an arbitrary number to $\mathbf{m}$ satisfy ξ. Hence, verification of algorithms that enjoy the cutoff property can be done by model checking of finite instances.

In this paper, we introduce the class of symmetric point-to-point algorithms that enjoys the cutoff property. Informally, an instance in this class contains N processes that follow the same algorithm, and communicate with each other by sending and receiving messages through point-to-point communication channels. At each process, local memory can be partitioned into regions such that one region corresponds one-to-one with a remote process, e.g. the array element $timeout\,[p, q]$ at a process p stores the maximum waiting time for a process q by the process p. The failure detector [5] is one example of this class. Let $1..N$ be a set of indexes. We show two cutoffs for these algorithms:

1. Let i be an index, and $\omega_{\{i\}}$ be an LTL\X (the stuttering-insensitive linear temporal logic) formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$. Properties of the form $\bigwedge_{i \in 1..N} \omega_{\{i\}}$ has a cutoff of 1.
2. Let i and j be different indexes, and $\psi_{\{i,j\}}$ be an LTL\X formula in which every predicate takes one of the (syntactic) forms: $Q_1(i)$, or $Q_2(j)$, or $Q_3(i, j)$, or $Q_4(j, i)$. Properties of the form $\bigwedge_{i,j \in 1..N}^{i \neq j} \psi_{\{i,j\}}$ has a cutoff of 2.

For instance, by the second cutoff result, we can verify the following property called the strong completeness property of the failure detector in [5] by model checking of an instance of size 2.

$$\mathbf{FG}(\forall i, j \in 1..N : (Correct(i) \wedge \neg Correct(j)) \Rightarrow Suspected(i, j))$$

This formula means that every crashed process is eventually permanently suspected by every correct process. We are writing **F** and **G** to denote "eventually" and "globally" operators of linear temporal logic (LTL), see [9]. We demonstrate the feasibility of our approach by specifying Chandra and Toueg's failure detectors [5] in the language TLA$^+$ [22], and model checking the specification with two model checkers: TLC [28] and APALACHE [20].

Related work. Our work is inspired by the cutoff results for various models of computation: ring-based message-passing systems [14,15], purely disjunctive guards and conjunctive guards [12,13], token-based communication [8], and quorum-based algorithms [23]. Additionally, there are semi-decision procedures based on invariants, induction, and abstraction that are successful in many interesting cases [4,7,18,21,25]. Interactive verification methods with proof assistants [10,16,26] have produced positive results in proving distributed algorithms.

The paper is organized as follows. Section 2 presents our motivating example - Chandra and Toueg's failure detector [5], and challenges in verification of these algorithms. Section 3 defines the model of computation as a transition system.

Section 4 shows our main contributions: two cutoff results in the class of symmetric point-to-point distributed algorithms. Section 5 presents how we encode the model of computation, and the failure detector of [5] in TLA$^+$, and the model checking results. Section 6 concludes the paper with a discussion of future extensions.

2 Motivating Example

This section starts with a description of our motivating example – Chandra and Toueg's failure detector [5]. Then, we present challenges in verification of the failure detector, and state-of-the-art verification techniques.

Algorithm 1 presents the pseudo-code of the failure detector of [5]. A system instance has N processes that communicate with each other by sending-to-all and receiving messages through N^2 point-to-point communication channels. A process performs local computation based on these messages (we assume that a process also receives the messages that it sends to itself). In one system step, all processes may take up to one step. Some processes may crash, i.e., stop operating. Correct processes follow Algorithm 1 to detect crashes in the system. Initially, every correct process sets a default value for a timeout of each other, i.e. how long it should wait for others and assumes that no processes have crashed (Line 4). Every correct process p has three tasks: (i) repeatedly sends an "alive" message to all (Line 6), and (ii) repeatedly produces predictions about crashes of other processes based on timeouts (Line 8), and (iii) increases a timeout for a process q if p has learned that its suspicion on q is wrong (Line 12). Notice that a process p raises suspicion on the operation of a process q (Line 8) by considering only information related to q: $timeout\,[p, q]$, $suspected\,[p, q]$, and messages that p has received from q recently. In other words, its suspicions about other processes grow independently.

Let $Correct(p)$ be a predicate whether a process p is correct. (However, p can crash later. A crashed process p_1 satisfies $\neg Correct(p_1)$.) Let $Suspected(p, q)$ be a predicate whether a process p suspects a process q. The failure detector should guarantee the following properties [5]:

- Strong completeness: Every crashed process is eventually permanently suspected by every correct process.

$$\mathbf{FG}(\forall p, q \in 1..N : (Correct(p) \wedge \neg Correct(q)) \Rightarrow Suspected(p, q))$$

- Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct processes.

$$\mathbf{FG}(\forall p, q \in 1..N : (Correct(p) \wedge Correct(q)) \Rightarrow \neg Suspected(p, q))$$

In the asynchronous model, Algorithm 1 does not satisfy eventually strong accuracy since there exists no bound on message delay, and messages sent by correct processes might always arrive after the timeout expires. The correctness of failure detectors is based on two implicit time constraints: (1) the transmission delay of messages and (2) the relative speeds of different processes [5]. Even if these upper bounds exist but are unknown, failure detectors can satisfy both strong completeness and eventually strong accuracy.

Algorithm 1. The eventually perfect failure detector algorithm in [5]

1: *Every process* $p \in \Pi$ *executes the following*:
2: **for all** $q \in \Pi$ **do** ▷ Initalization step
3: $timeout\,[p, q] =$ default-value
4: $suspected\,[p, q] = \bot$ }
5:
6: Send "alive" to all $q \in \Pi$ ▷ Task 1: repeat periodically
7:
8: **for all** $q \in \Pi$ **do** ▷ Task 2: repeat periodically
9: **if** $suspected\,[p, q] = \bot$ **and not hear** q during last $timeout\,[p, q]$ ticks **then**
10: $suspected\,[p, q] = \top$
11:
12: **if** $suspected\,[p, q]$ **then** ▷ Task 3: when receive "alive" from q
13: $timeout\,[p, q] \leftarrow timeout\,[p, q] + 1$
14: $suspected\,[p, q] = \bot$

Note that the symmetry exists in both the failure detectors of [5] and the above correctness properties. First, every process is isomorphic under renaming. A correct process p always sends a message to all and raises suspicion on a process q by considering only information related to q. Second, there are only point-to-point communication channels. Third, the contents of in-transit messages is identical. They are merely "keep-alive" messages that may arrive at different times. Finally, all variables in both properties strong completeness and eventual strong accuracy are variables over process indices, and they are bound by universal quantifiers. The symmetry is captured by our model of computation and is the key point in our proofs of the cutoff results.

As a result, verification of failure detectors faces the following challenges:

1. Algorithms are parameterized by the number of processes. Hence, we need to verify infinitely many instances of algorithms.
2. Its model of computation lies between synchrony and asynchrony since multiple processes can take a step in a global transition.
3. The algorithm relies on a global clock and local clocks. A straightforward encoding of a clock with an integer would produce an infinite state space.
4. The algorithm is parameterized with the upper bounds on transmission time of messages, and the relative speeds of different processes. These upper bounds are called Δ and Φ, respectively.

In this paper, we focus on Challenges 1–2: Our model of computation in Sect. 3 does not restrict the number of processes that simultaneously take a step, and we show cutoffs on the number of processes in Sect. 4. Our cutoff results apply for checking LTL\X formulas of the forms $\bigwedge_{i \in 1..N} \omega_{\{1\}}$ and $\bigwedge_{i,j \in 1..N}^{i \neq j} \psi_{\{1,2\}}$. Hence, we can verify the failure detector of [5] by model checking its few instances. We demonstrate the feasibility of our approach by specifying and model checking the failure detector in the *synchronous* case. Our specification contains optimizations for Challenge 3, which allows us to efficiently encode global and local clocks. In the synchronous case, we can skip Challenge 4, which we leave for furture work.

3 Model of Computation

In this section, we formalize a distributed system as a transition system. This formalization captures the semantics of the theoretical model of [5,11], but does not consider the restrictions on the execution space given by Δ and Φ. A global system is a composition of N processes, N^2 point-to-point outgoing message buffers, and N control components that capture what processes can take a step. Every process is identified with a unique index in $1..N$, and follows the same deterministic algorithm. Moreover, a global system allows: (i) multiple processes to take (at most) one step in one global transition, and (ii) some processes to crash. Every process may execute three kinds of transitions: *internal*, *round*, and stuttering. Notice that in one global transition, some processes may send a message to all, and some may receive messages and do computation. Hence, we need to decide which processes move, and what happens to the message buffers. We introduce four sub-rounds: *Schedule*, *Send*, *Receive*, and *Computation*. The transitions for these sub-rounds are called internal ones. A global round transition is a composition of four internal transitions. We formalize sub-rounds and global transitions later. As a result of modeling, there exists an arbitrary sequence of global configurations which is not accepted in [5,11]. We define so-called *admissible* sequences of global configurations that are accepted in [5,11]. We did encode our formalization in TLA$^+$, and our specification is presented in Sect. 5.

Since every process follows the same algorithm, we first define a process template that captures the process behavior. This formalization focuses on symmetric point-to-point algorithms parameterized by N. Every process is an instance of the process template. Then, we present the formalization of a global system.

3.1 The Process Template

We fix a set of process indexes as $1..N$. Moreover, we assume that the message content does not have indexes of its receiver and sender. We let `Msg` denote a set of potential messages, and `Set(Msg)` denote a set of sets of messages.

We model a process template as a transition system $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_N^0)$ where $Q_N = Loc \times \texttt{Set}(\texttt{Msg})^N \times \mathcal{D}^N$ is a set of template states, Tr_N is a set of template transitions, $Rel_N \subseteq Q_N \times Tr_N \times Q_N$ is a template transition relation, and $q_N^0 \in Q_N$ is an initial state. These components of $\mathcal{U}_N$ are defined as follows.

States. A *template state* ρ is a tuple $(\ell, S_1, \ldots, S_N, d_1, \ldots, d_N)$ where:

- $\ell \in Loc$ refers to a location of a program counter, and Loc is a set of locations. We assume that $Loc = Loc_{snd} \cup Loc_{rcv} \cup Loc_{comp} \cup \{\ell_{crash}\}$, and three sets Loc_{snd}, Loc_{rcv}, Loc_{comp} are disjoint, and ℓ_{crash} is a special location of crashes. To access the program counter, we use a function $pc\colon Q_N \to Loc$ that takes a template state at its input, and produces its program counter as the output. Let $\rho(k)$ denote the k^{th} component in a template state ρ. For every $\rho \in Q_N$, we have $pc(\rho) = \rho(1)$.
- $S_i \in \texttt{Set}(\texttt{Msg})$ refers to a set of messages. It is to store the messages received from a process p_i for every $i \in 1..N$. To access a set of received messages from

a particular process whose index in $1..N$, we use a function $rcvd\colon Q_N \times 1..N \to$ `Set(Msg)` that takes a template state ρ and a process index i at its input, and produces the $(i+1)^{th}$ component of ρ at the output, i.e. for every $\rho \in Q_N$, we have $rcvd(\rho, i) = \rho(1+i)$.
- $d_i \in \mathcal{D}$ refers to a local variable related to a process p_i for every $i \in 1..N$. To access a local variable related to a particular process whose index in $1..N$, we use a function $lvar\colon Q_N \times 1..N \to \mathcal{D}$ that takes a template state ρ and a process index i at its input, and produces the $(1+N+i)^{th}$ component of ρ as the output, i.e. $lvar(\rho, i) = \rho(1+N+i)$ for every $\rho \in Q_N$.

Initial State. The initial state q_N^0 is a tuple $q_N^0 = (\ell_0, \emptyset, \ldots, \emptyset, d_0, \ldots, d_0)$ where ℓ_0 is a location, every box for received messages is empty, and every local variable is assigned a constant $d_0 \in \mathcal{D}$.

Transitions. We define $Tr_N = CSnd \cup CRcv \cup \{comp, crash, stutter\}$ where

- *CSnd* is a set of transitions. Every transition in *CSnd* refers to a task that does some internal computation, and sends a message to all. For example, in task 1 in Algorithm 1, a process increases its local clock, and performs an instruction to send "alive" to all. We let $csnd(m)$ denote a transition referring to a task with an action to send a message $m \in$ `Msg` to all.
- *CRcv* is a set of transitions. Every transition in *CRcv* refers to a task that receives N sets of messages, and does some internal computation. For example, in task 2 in Algorithm 1, a process increases its local clock, receives messages, and remove false-negative predictions. We let $crcv(S_1, \ldots, S_N)$ denote a transition referring to a task with an action to receive sets $S_1, \ldots, S_N$ of messages. These sets $S_1, \ldots, S_N$ are delivered by the global system.
- *comp* is a transition which refers to a task with purely local computation. In other words, this task has neither send actions nor receive actions.
- *crash* is a transition for crashes.
- *stutter* is a transition for stuttering steps.

Transition Relation. For two states $\rho, \rho' \in Q_N$ and a transition $tr \in Tr_N$, we write $\rho \xrightarrow{tr} \rho'$, instead of $(\rho, \xrightarrow{tr}, \rho')$. In the model of [5,11], each process follows the same deterministic algorithm. Hence, we assume that for every $\rho_0 \xrightarrow{tr_0} \rho_0'$ and $\rho_1 \xrightarrow{tr_1} \rho_1'$, if $\rho_0 = \rho_1$ and $tr_0 = tr_1$, then it follows that $\rho_0' = \rho_1'$. Moreover, we assume that there exist the following functions which are used to define constraints on the template transition relation:

- A function $nextLoc\colon Loc \to Loc$ takes a location at its input and produces the next location as the output.
- A function $genMsg\colon Loc \to$ `Set(Msg)` a location at its input, and produces a singleton set that contains the message that is sent to all processes in the current task. The output can be an empty set. For example, if a process is performing a receive task, the output of *genMsg* is an empty set.
- A function $nextVar\colon Loc \times$ `Set(Msg)` $\times \mathcal{D} \to \mathcal{D}$ takes a location, a set of messages, and a local variable's value, and produces a new value of a local variable as the output.

Let us fix functions *nextLoc*, *genMsg* and *nextVar*. We define the template transitions as follows.

1. For every $m \in \texttt{Msg}$, for every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{csnd(m)} \rho'$ if and only if
   (a) $pc(\rho') = nextLoc(pc(\rho)) \wedge \{m\} = genMsg(pc(\rho))$
   (b) $\forall i \in 1..N\colon rcvd(\rho, i) = rcvd(\rho', i)$
   (c) $\forall i \in 1..N\colon lvar(\rho', i) = nextVar\big(pc(\rho), rcvd(\rho', i), lvar(\rho, i)\big)$
   Constraint (a) implies that the update of a program counter and the construction of a sent message m depend on only the current value of a program counter, and a process wants to send only m to all in this step. Constraint (b) is that no message was delivered. Constraint (c) implies that the value of $lvar(\rho', i)$ depends on only the current location, a set of messages that have been delivered and stored in $rcvd(\rho', i)$, and the value of $lvar(\rho, i)$.
2. For every $S_1, \ldots, S_N \subseteq \texttt{Msg}$, for every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{crcv(S_1,\ldots,S_N)} \rho'$ if and only if the following constraints hold:
   (a) $pc(\rho') = nextLoc(pc(\rho)) \wedge \emptyset = genMsg(pc(\rho))$
   (b) $\forall i \in 1..N\colon rcvd(\rho', i) = rcvd(\rho, i) \cup S_i$
   (c) $\forall i \in 1..N\colon lvar(\rho', i) = nextVar\big(pc(\rho)), rcvd(\rho', i), lvar(\rho, i)\big)$
   Constraint (a) in *crcv* is similar to constraint (a) in *csnd*, except that no message is sent in this sub-round. Constraint (b) refers that messages in a set S_i are from a process indexed i, and have been delivered in this step. Constraint (c) in *crcv* is similar to constraint (c) in *csnd*.
3. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{comp} \rho'$ if and only if the following constraints hold:
   (a) $pc(\rho') = nextLoc(pc(\rho)) \wedge \emptyset = genMsg(pc(\rho))$
   (b) $\forall i \in 1..N\colon rcvd(\rho', i) = rcvd(\rho, i)$
   (c) $\forall i \in 1..N\colon lvar(\rho', i) = nextVar\big(pc(\rho), rcvd(\rho', i), lvar(\rho, i)\big)$
   Hence, this step has only local computation. No message is sent or delivered.
4. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{crash} \rho'$ if and only if the following constraints hold:
   (a) $pc(\rho) \neq \ell_{crash} \wedge pc(\rho') = \ell_{crash}$
   (b) $\forall i \in 1..N\colon rcvd(\rho, i) = rcvd(\rho', i) \wedge lvar(\rho, i) = lvar(\rho', i)$
   Only the program counter is updated by switching to ℓ_{crash}.
5. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{stutter} \rho'$ if and only if $\rho = \rho'$.

3.2 Modeling the Distributed System

Given N processes which are instantiated from the same process template $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_N^0)$, the global system is a composition of (i) these processes, and (ii) N^2 point-to-point buffers for in-transit messages, and (iii) N control components that capture what processes can take a step. We formalize the global system as a transition system $\mathcal{G}_N = \big(\mathcal{C}_N, T_N, R_N, g_N^0\big)$ where $\mathcal{C}_N = (Q_N)^N \times \texttt{Set}(\texttt{Msg})^{N \cdot N} \times \texttt{Bool}^N$ is a set of global configurations, and T_N is a set of global *internal*, *round*, and stuttering transitions, and $R_N \subseteq \mathcal{C}_N \times T_N \times \mathcal{C}_N$ is a global

transition relation, and g_N^0 is an initial configuration. These components are defined as follows.

Configurations. A *global configuration* κ is defined as a following tuple $\kappa = \big(q_1, \ldots, q_N, S_1^1, S_1^2 \ldots, S_s^r, \ldots S_N^N, act_1, \ldots, act_N\big)$ where:

- $q_i \in Q_N$: This component is a state of a process p_i for every $i \in 1..N$. To access a local state of a particular process, we use a function $lstate \colon \mathcal{C}_N \times 1..N \to Q_N$ that takes input as a global configuration κ and a process index i, and produces output as the i^{th} component of κ which is a state of a process p_i. Let $\kappa(i)$ denote the i^{th} component of a global configuration κ. For every $i \in 1..N$, we have $lstate(\kappa, i) = \kappa(i) = q_i$.
- $S_s^r \in \texttt{Set}(\texttt{Msg})$: This component is a set of in-transit messages from a process p_s to a process p_r for every $s, r \in 1..N$. To access a set of in-transit messages between two processes, we use a function $buf \colon \mathcal{C}_N \times 1..N \times 1..N \to \texttt{Set}(\texttt{Msg})$ that takes input as a global configuration κ, and two process indexes s, r, and produces output as the $(s \cdot N + r)^{th}$ component of κ which is a message buffer from a process p_s (sender) to a process p_r (receiver). Formally, we have $buf(\kappa, s, r) = \kappa(s \cdot N + r) = S_s^r$ for every $s, r \in 1..N$.
- $act_i \in \texttt{Bool}$: This component says whether a process p_i can take one step in a global transition for every $i \in 1..N$. To access a control component, we use a function $active \colon \mathcal{C}_N \times 1..N \to \texttt{Bool}$ that takes input as a configuration κ and a process index i, and produces output as the $((N+1) \cdot N + i)^{th}$ component of κ which refers to whether a process p_i can take a step. Formally, we have $active(\kappa, i) = \kappa((N+1) \cdot N + i) = b_i$ for every $i \in 1..N$. The environment sets the values of $act_1, \ldots, act_N$ in the sub-round Schedule defined later.

We will write $\kappa \in (Q_N)^N \times \texttt{Set}(\texttt{Msg})^{N \cdot N} \times \texttt{Bool}^N$ or $\kappa \in \mathcal{C}_N$.

Initial Configuration. The global system $\mathcal{G}_N$ has one initial configuration g_N^0, and it must satisfy the following constraints:

1. $\forall i \in 1..N \colon \neg active(g_N^0, i) \wedge lstate(N, i) = q_N^0$
2. $\forall s, r \in 1..N \colon buf(g_N^0, s, r) = \emptyset$

Global Round Transitions. Intuitively, every *round* transition is a sequence of a $\xrightarrow{\texttt{Sched}}$ transition, a $\xrightarrow{\texttt{Snd}}$ transition, a $\xrightarrow{\texttt{Rcv}}$ transition, and a $\xrightarrow{\texttt{Comp}}$ transition defined below. We let $\rightsquigarrow$ denote *round* transitions. The semantics of round transitions is defined as follows: for every pair of global configurations $\kappa_0, \kappa_4 \in \mathcal{C}_N$, we say $\kappa_0 \rightsquigarrow \kappa_4$ if there exist three global configurations $\kappa_1, \kappa_2, \kappa_3 \in \mathcal{C}_N$ such that $\kappa_0 \xrightarrow{\texttt{Sched}} \kappa_1 \xrightarrow{\texttt{Snd}} \kappa_2 \xrightarrow{\texttt{Rcv}} \kappa_3 \xrightarrow{\texttt{Comp}} \kappa_4$. Notice that every correct process can make at most one global internal transition in every global round transition. Moreover, round transitions allow some processes to crash only in the sub-round *Schedule*. We call these faults *clean-crashes*.

Global Stuttering Transition. In the proof of Lemma 5 presented in Section 4, we do projection. Therefore, we extend the relation $\rightsquigarrow$ with stuttering: for every configuration κ, we allow $\kappa \rightsquigarrow \kappa$.

Admissible Sequences. An infinite sequence $\pi = \kappa_0\kappa_1 \ldots$ of global configurations in $\mathcal{G}_N$ is *admissible* if the following constraints hold:

1. κ_0 is the initial state, i.e. $\kappa_0 = g_N^0$, and
2. π is stuttering equivalent with an infinite sequence $\pi' = \kappa'_0\kappa'_1 \ldots$ such that $\kappa'_{4k} \xrightarrow{\texttt{Sched}} \kappa'_{4k+1} \xrightarrow{\texttt{Snd}} \kappa'_{4k+2} \xrightarrow{\texttt{Rcv}} \kappa'_{4k+3} \xrightarrow{\texttt{Comp}} \kappa'_{4k+4}$ for every $k \geq 0$.

Notice that it immediately follows by this definition that if $\pi = \kappa_0\kappa_1 \ldots$ is an admissible sequence of configurations in $\mathcal{G}_N$, then $\kappa'_{4k} \rightsquigarrow \kappa'_{4k+4}$ for every $k \geq 0$. From now on, we only consider admissible sequences of global configurations.

Global Internal Transitions. In the model of [5], many processes can take a step in a global transition. We assume that a computation of the distributed system is organized in rounds, i.e.global ticks, and every round is organized as four sub-rounds called *Schedule, Send, Receive*, and *Computation*. To model that as a transition system, for every sub-round we define a corresponding transition: $\xrightarrow{\texttt{Sched}}$ for the sub-round *Schedule*, $\xrightarrow{\texttt{Snd}}$ for the sub-round *Send*, $\xrightarrow{\texttt{Rcv}}$ for the sub-round *Receive*, $\xrightarrow{\texttt{Comp}}$ for the sub-round Comp. These transitions are called global *internal* transitions. We define the semantics of these sub-rounds as follows.

1. Sub-round *Schedule*. The environment starts with a global configuration where every process is inactive, and move to another by non-deterministically deciding what processes become crashed, and what processes take a step in the current global transition. Every correct process takes a stuttering step, and every faulty process is inactive. If a process p is crashed in this sub-round, every incoming message buffer to p is set to the empty set. Formally, for $\kappa, \kappa' \in \mathcal{C}_N$, we have $\kappa \xrightarrow{\texttt{Sched}} \kappa'$ if the following constraints hold:
   (a) $\forall i \in 1..N\colon \neg \mathit{active}(\kappa, i)$
   (b) $\forall i \in 1..N\colon \mathit{lstate}(\kappa, i) \xrightarrow{\mathit{stutter}} \mathit{lstate}(\kappa', i) \vee \mathit{lstate}(\kappa, i) \xrightarrow{\mathit{crash}} \mathit{lstate}(\kappa', i)$
   (c) $\forall i \in 1..N\colon \mathit{pc}(\mathit{lstate}(\kappa', i)) = \ell_{\mathit{crash}} \Rightarrow \neg \mathit{active}(\kappa', i)$
   (d) $\forall s, r \in 1..N\colon \mathit{pc}(\mathit{lstate}(\kappa', r)) \neq \ell_{\mathit{crash}} \Rightarrow \mathit{buf}(\kappa, s, r) = \mathit{buf}(\kappa', s, r)$
   (e) $\forall r \in 1..N\colon \mathit{pc}(\mathit{lstate}(\kappa', r)) = \ell_{\mathit{crash}} \Rightarrow (\forall s \in 1..N\colon \mathit{buf}(\kappa', s, r) = \emptyset)$
2. Sub-round *Send*. Only processes that perform send actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Fresh sent messages are added to corresponding message buffers. To define the semantics of the sub-round Send, we use the following predicates:

$$
\begin{aligned}
\mathit{Enabled}(\psi, i, L) &\triangleq \mathit{active}(\kappa, i) \wedge \mathit{pc}(\mathit{lstate}(\kappa, i)) \in L \\
\mathit{Frozen}_S(\psi_1, \psi_2, i) &\triangleq \wedge\ \mathit{lstate}(\kappa, i) \xrightarrow{\mathit{stutter}} \mathit{lstate}(\kappa', i) \\
&\quad\ \wedge\ \mathit{active}(\kappa, i) = \mathit{active}(\kappa', i) \\
&\quad\ \wedge\ \forall r \in 1..N\colon \mathit{buf}(\kappa, i, r) = \mathit{buf}(\kappa', i, r) \\
\mathit{Sending}(\psi_1, \psi_2, i, m) &\triangleq \wedge\ \forall r \in 1..N\colon m \notin \mathit{buf}(\kappa, i, r) \\
&\quad\ \wedge\ \forall r \in 1..N\colon \mathit{buf}(\kappa', i, r) = \{m\} \cup \mathit{buf}(\kappa, i, r) \\
&\quad\ \wedge\ \mathit{lstate}(\kappa, i) \xrightarrow{\mathit{csnd}(m)} \mathit{lstate}(\kappa', i)
\end{aligned}
$$

   Formally, for $\kappa, \kappa' \in \mathcal{C}_N$, we have $\kappa \xrightarrow{\texttt{Snd}} \kappa'$ if the following constraints hold:

(a) $\forall i \in 1..N\colon \neg Enabled(\kappa, i, Loc_{snd}) \Leftrightarrow Frozen_S(\kappa, \kappa', i)$
(b) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{snd}) \Leftrightarrow \exists m \in \texttt{Msg}\colon Sending(\kappa, \kappa', i, m)$
(c) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{snd}) \Rightarrow \neg active(\kappa', i)$

3. Sub-round *Receive*. Only processes that perform receive actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Delivered messages are removed from corresponding message buffers. To define the semantics of this sub-round, we use the following predicates:

$$\begin{aligned} Frozen_R(\psi_1, \psi_2, i) \triangleq\ & \wedge\ lstate(\kappa, i) \xrightarrow{stutter} lstate(\kappa', i) \\ & \wedge\ active(\kappa, i) = active(\kappa', i) \\ & \wedge\ \forall s \in 1..N\colon buf(\kappa, s, i) = buf(\kappa', s, i) \\ Receiving(\kappa, \kappa', i, S_1, \ldots, S_N) \triangleq\ & \wedge\ \forall s \in 1..N\colon S_s \not\subseteq buf(\kappa', s, i) \\ & \wedge\ \forall s \in 1..N\colon buf(\kappa', s, i) \cup S_s = buf(\kappa, s, i) \\ & \wedge\ lstate(\kappa, i) \xrightarrow{crcv(S_1,\ldots,S_N)} lstate(\kappa', i) \end{aligned}$$

Formally, for $\kappa, \kappa' \in \mathcal{C}_N$, we have $\kappa \xrightarrow{\texttt{Rcv}} \kappa'$ if the following constraints hold:
(a) $\forall i \in 1..N\colon \neg Enabled(\kappa, i, Loc_{rcv}) \Leftrightarrow Frozen_R(\kappa, \kappa', i)$
(b) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{rcv})$
$\Leftrightarrow \exists S_1, \ldots, S_N \subseteq \texttt{Msg}\colon Receiving(\kappa, \kappa', i, S_1, \ldots, S_N)$
(c) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{rcv}) \Rightarrow \neg active(\kappa', i)$

4. Sub-round *Computation*. Only processes that perform internal computation actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Every message buffer is unchanged. Formally, for $\kappa, \kappa' \in \mathcal{C}_N$, we have $\kappa \xrightarrow{\texttt{Comp}} \kappa'$ if the following constraints hold:
(a) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{comp}) \Leftrightarrow lstate(\kappa, i) \xrightarrow{comp} lstate(\kappa', i)$
(b) $\forall i \in 1..N\colon \neg Enabled(\kappa, i, Loc_{comp}) \Leftrightarrow lstate(\kappa, i) \xrightarrow{stutter} lstate(\kappa', i)$
(c) $\forall s, r \in 1..N\colon buf(\kappa, s, r) = buf(\kappa', s, r)$
(d) $\forall i \in 1..N\colon Enabled(\kappa, i, Loc_{comp}) \Rightarrow \neg active(\kappa', i)$

Remark 1. Observe that the definitions of $\kappa \xrightarrow{\texttt{Snd}} \kappa'$, and $\kappa \xrightarrow{\texttt{Rcv}} \kappa'$, and $\kappa \xrightarrow{\texttt{Comp}} \kappa'$ allow $\kappa = \kappa'$, that is stuttering. This captures, e.g. global transitions in [5,11] where no process sends a message.

4 Cutoff Results

Let $\mathcal{A}$ be a symmetric point–to–point algorithm. In this section, we show two cutoff results for the number of processes in the algorithm $\mathcal{A}$. With these cutoff results, one can verify the strong completeness and eventually strong accuracy of the failure detector of [5] by model checking two instances of sizes 1 and 2.

Theorem 1. *Let $\mathcal{A}$ be a symmetric point–to–point algorithm. Let $\mathcal{G}_1$ and $\mathcal{G}_N$ be instances of 2 and N processes respectively for some $N \geq 1$. Let $Path_1$ and $Path_N$ be sets of all admissible sequences of configurations in $\mathcal{G}_1$ and in $\mathcal{G}_N$, respectively. Let $\omega_{\{i\}}$ be a LTL\X formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$ where i is an index in $1..N$. Then,*

$$\Big(\forall \pi_N \in Path_N : \mathcal{G}_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{\{i\}}\Big) \Leftrightarrow \Big(\forall \pi_1 \in Path_1 : \mathcal{G}_1, \pi_1 \models \omega_{\{1\}}\Big).$$

Theorem 2. *Let $\mathcal{A}$ be a symmetric point–to–point algorithm. Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be instances of 2 and N processes respectively for some $N \geq 2$. Let $Path_2$ and $Path_N$ be sets of all admissible sequences of configurations in $\mathcal{G}_2$ and in $\mathcal{G}_N$, respectively. Let $\psi_{\{i,j\}}$ be an LTL\X formula in which every predicate takes one of the forms: $Q_1(i)$, or $Q_2(j)$, or $Q_3(i,j)$, or $Q_4(j,i)$ where i and j are different indexes in $1..N$. It follows that:*

$$\big(\forall \pi_N \in Path_N : \mathcal{G}_N, \pi_N \models \bigwedge_{i,j \in 1..N}^{i \neq j} \psi_{\{i,j\}}\big) \Leftrightarrow \big(\forall \pi_2 \in Path_2 : \mathcal{G}_2, \pi_2 \models \psi_{\{1,2\}}\big).$$

Since the proof of Theorem 1 is similar to the one of Theorem 2, we focus on Theorem 2 here. Its proof is based on the symmetric characteristics in the system model and correctness properties, and on the following lemmas.

- Lemma 1 says that every transposition on a set of process indexes $1..N$ preserves the structure of the process template $\mathcal{U}_N$.
- Lemma 2 says that every transposition on a set of process indexes $1..N$ preserves the structure of the global transition system $\mathcal{G}_N$ for every $N \geq 1$.
- Lemma 5 says that $\mathcal{G}_2$ and $\mathcal{G}_N$ are trace equivalent under a set $AP_{\{1,2\}}$ of predicates that take one of the forms: $Q_1(i)$, or $Q_2(j)$, or $Q_3(i,j)$, or $Q_4(j,i)$.

In the following, we present definitions and constructions to prove these lemmas. We end this section with the proof sketch of Theorem 2.

4.1 Index Transpositions And symmetric point–to–point systems

We first recall the definition of transposition. Given a set $1..N$ of indexes, we call a bijection $\alpha : 1..N \to 1..N$ a transposition between two indexes $i, j \in 1..N$ if the following properties hold: $\alpha(i) = j$, and $\alpha(j) = i$, and $\forall k \in 1..N : (k \neq i \wedge k \neq j) \Rightarrow \alpha(k) = k$. We let $(i \leftrightarrow j)$ denote a transposition between two indexes i and j.

The application of a transposition to a template state is given in Definition 1. Informally, applying a transposition $\alpha = (i \leftrightarrow j)$ to a template state ρ generates a new template state by switching only the evaluation of *rcvd* and *lvar* at indexes i and j. The application of a transposition to a global configuration is provided in Definition 2. In addition to process configurations, we need to change message buffers and control components. We override notation by writing $\alpha_S(\rho)$ and $\alpha_C(\kappa)$ to refer the application of a transposition α to a state ρ and to a configuration κ, respectively. These functions α_S and α_C are named a local transposition and a global transposition, respectively.

Definition 1 (Local Transposition). *Let $\mathcal{U}_N$ be a process template with process indexes $1..N$, and $\rho = (\ell, S_1, \ldots, S_N, d_1, \ldots, d_N)$ be a state in $\mathcal{U}_N$. Let $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$. The application of α to ρ, denoted as $\alpha_S(\rho)$, generates a tuple $(\ell', S'_1, \ldots, S'_N, d'_1, \ldots, d'_N)$ such that*

1. $\ell = \ell'$, *and* $S_i = S'_j$, *and* $S_j = S'_i$, *and* $d_i = d'_j$ *and* $d_j = d'_i$, *and*
2. $\forall k \in 1..N\colon (k \neq i \wedge k \neq j) \Rightarrow (S_k = S'_k \wedge d_k = d'_k)$

Definition 2 (Global Transposition). *Let $\mathcal{G}_N$ be a global system with process indexes $1..N$, and κ be a configuration in $\mathcal{G}_N$. Let $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$. The application of α to κ, denoted as $\alpha_C(\kappa)$, generates a configuration in $\mathcal{G}_N$ which satisfies following properties:*

1. $\forall i \in 1..N\colon lstate(\alpha_C(\kappa), \alpha(i)) = \alpha_S(lstate(\kappa, i))$.
2. $\forall s, r \in 1..N\colon buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) = buf(\kappa, s, r)$
3. $\forall i \in 1..N\colon active(\alpha_C(\kappa), \alpha(i)) = active(\kappa, i)$

Since the content of every message in `Msg` does not have indexes of the receiver and sender, no transposition affects the messages. We define the application of a transposition to one of send, compute, crash, and stutter template transitions return the same transition. We extend the application of a transposition to a receive template transition as in Definition 3.

Definition 3 (Receive-transition Transposition). *Let $\mathcal{U}_N$ be a process template with process indexes $1..N$, and $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$. Let $crcv(S_1, \ldots, S_N)$ be a transition in $\mathcal{U}_N$ which refers to a task with a receive action. We let $\alpha_R(crcv(S_1, \ldots, S_N))$ denote the application of α to $crcv(S_1, \ldots, S_N)$, and this application returns a new transition $crcv(S'_1, \ldots, S'_N)$ in $\mathcal{U}_N$ such that:*

1. $S_i = S'_j$, *and* $S_j = S'_i$, *and*
2. $\forall k \in 1..N\colon (k \neq i \wedge k \neq j) \Rightarrow (S_k = S'_k \wedge d_k = d'_k)$

We let $\alpha_U(\mathcal{U}_N)$ and $\alpha_G(\mathcal{G}_N)$ denote the application of a transposition α to a process template $\mathcal{U}_N$ and a global transition $\mathcal{G}_N$, respectively. Since these definitions are straightforward, we skip them in this paper. We prove later that $\alpha_S(\mathcal{U}_N) = \mathcal{U}_N$ and $\alpha_C(\mathcal{G}_N) = \mathcal{G}_N$ (see Lemmas 1 and 2).

Lemma 1 (Symmetric Process Template). *Let $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_N^0)$ be a process template with indexes $1..N$. Let $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$, and α_S be a local transposition based on α (from Definition 1). The following properties hold:*

1. *α_S is a bijection from Q_N to itself.*
2. *The initial state is preserved under α_S, i.e. $\alpha_S(q_N^0) = q_N^0$.*
3. *Let $\rho, \rho' \in \mathcal{U}_N$ be states such that $\rho \xrightarrow{crcv(S_1,\ldots,S_N)} \rho'$ for some sets of messages $S_1, \ldots, S_N \in \texttt{Set}(\texttt{Msg})$. It follows $\alpha_S(\rho) \xrightarrow{\alpha_R(crcv(S_1,\ldots,S_N))} \alpha_S(\rho')$.*

4. *Let ρ, ρ' be states in $\mathcal{U}_N$, and $tr \in Tr_N$ be one of send, local computation, crash and stutter transitions such that $\rho \xrightarrow{tr} \rho'$. Then, $\alpha_S(\rho) \xrightarrow{tr} \alpha_S(\rho')$.*

Lemma 2 (Symmetric Global System). *Let $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_N^0)$ be a process template, and $\mathcal{G}_N = (\mathcal{C}_N, T_N, R_N, g_N^0)$ be a global transition system such that the process indexes is a set $1..N$, and every process is instantiated with $\mathcal{U}_N$. Let α be a transposition on $1..N$, and α_C be a global transposition based on α (from Definition 2). The following properties hold:*

1. *α_C is a bijection from $\mathcal{C}_N$ to itself.*
2. *The initial configuration is preserved under α_C, i.e. $\alpha_C(g_N^0) = g_N^0$.*
3. *Let κ and κ' be configurations in $\mathcal{G}_N$, and $tr \in T_N$ be either a internal transition such that $\kappa \xrightarrow{tr} \kappa'$. It follows $\alpha_C(\kappa) \xrightarrow{tr} \alpha_C(\kappa')$.*
4. *Let κ and κ' be configurations in $\mathcal{G}_N$. If $\kappa \rightsquigarrow \kappa'$, then $\alpha_C(\kappa) \rightsquigarrow \alpha_C(\kappa')$.*

4.2 Trace Equivalence of $\mathcal{G}_2$ and $\mathcal{G}_N$ Under $AP_{\{1,2\}}$

Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be two global transition systems whose processes follow the same symmetric point–to–point algorithm. In the following, our goal is to prove Lemma 5 that says $\mathcal{G}_2$ and $\mathcal{G}_N$ are trace equivalent under a set $AP_{\{1,2\}}$ of predicates which take one of the forms: $Q_1(1), Q_2(2), Q_3(1,2)$, or $Q_4(2,1)$. To do that, we first present two construction techniques: Construction 1 to construct a state in $\mathcal{U}_2$ from a state in $\mathcal{U}_N$, and Construction 2 to construct a global configuration in $\mathcal{G}_2$ from a given global configuration in $\mathcal{G}_N$. Then, we present two Lemmas 3 and 4. These lemmas are required in the proof of Lemma 5.

To keep the presentation simple, when the context is clear, we simply write $\mathcal{U}_N$, instead of fully $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_N^0)$. We also write $\mathcal{G}_N$, instead of fully $\mathcal{G}_N = (\mathcal{C}_N, T_N, R_N, g_N^0)$.

Construction 1 (State Projection). *Let $\mathcal{A}$ be an arbitrary symmetric point–to–point algorithm. Let $\mathcal{U}_N$ be a process template of $\mathcal{A}$ for some $N \geq 2$, and ρ^N be a process configuration of $\mathcal{U}_N$. We construct a tuple $\rho^2 = (pc_1, rcvd_1, rcvd_2, v_1, v_2)$ based on ρ^N and a set $\{1,2\}$ of process indexes in the following way:*

1. *$pc_1 = pc(\rho^N)$.*
2. *For every $i \in \{1,2\}$, it follows $rcvd_i = rcvd(\rho^N, i)$.*
3. *For every $i \in \{1,2\}$, it follows $v_i = lvar(\rho^N, i)$.*

Construction 2 (Configuration Projection). *Let $\mathcal{A}$ be a symmetric point–to–point algorithm. Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be two global transitions of two instances of $\mathcal{A}$ for some $N \geq 2$, and $\kappa^N \in \mathcal{C}_N$ be a global configuration in $\mathcal{G}_N$. We construct a tuple $\kappa^2 = (s_1, s_2, buf_1^1, buf_1^2, buf_2^1, buf_2^2, act_1, act_2)$ based on the configuration κ^N and a set $\{1,2\}$ of indexes in the following way:*

1. *For every $i \in \{1,2\}$, a component s_i is constructed from $lstate(\kappa^N, i)$ with Construction 1 and indexes $\{1,2\}$.*

2. *For every* $s, r \in \{1, 2\}$, *it follows* $buf_s^r = buf(\kappa^N, s, r)$.
3. *For every process* $i \in \{1, 2\}$, *it follows* $act_i = active(\kappa^N, i)$.

Note that a tuple ρ^2 constructed with Construction 1 is a state in $\mathcal{U}_2$, and a tuple κ^2 constructed with Construction 2 is a configuration in $\mathcal{G}_2$. We call ρ^2 (and κ^2) the *index projection* of ρ^N (and κ^N) on indexes $\{1, 2\}$. The following Lemma 3 says that Construction 2 allows us to construct an admissible sequence of global configurations in $\mathcal{G}_2$ based on a given admissible sequence in $\mathcal{G}_N$.

Lemma 3. *Let* $\mathcal{A}$ *be a symmetric point–to–point algorithm. Let* $\mathcal{G}_2$ *and* $\mathcal{G}_N$ *be two transition systems such that all processes in* $\mathcal{G}_2$ *and* $\mathcal{G}_N$ *follow* $\mathcal{A}$, *and* $N \geq 2$. *Let* $\pi^N = \kappa_0^N \kappa_1^N \ldots$ *be an admissible sequence of configurations in* $\mathcal{G}_N$. *Let* $\pi^2 = \kappa_0^2 \kappa_1^2 \ldots$ *be a sequence of configurations in* $\mathcal{G}_2$ *such that* κ_k^2 *is the index projection of* κ_k^N *on indexes* $\{1, 2\}$ *for every* $k \geq 0$. *Then,* π^2 *is admissible in* $\mathcal{G}_2$.

The proof of Lemma 3 is based on the following observations:

1. The application of Construction 1 to an initial template state of $\mathcal{U}_N$ constructs an initial template state of $\mathcal{U}_2$.
2. Construction 1 preserves the template transition relation.
3. The application of Construction 2 to an initial global configuration of $\mathcal{G}_N$ constructs an initial global configuration of $\mathcal{G}_2$.
4. Construction 2 preserves the global transition relation.

Moreover, Lemma 4 says that given an admissible sequence $\pi^2 = \kappa_0^2 \kappa_1^2 \ldots$ in $\mathcal{G}_2$, there exists an admissible sequence $\pi^N = \kappa_0^N \kappa_1^N \ldots$ in $\mathcal{G}_N$ such that κ_i^2 is the index projection of κ_i^N on indexes $\{1, 2\}$ for every $0 \leq i$.

Lemma 4. *Let* $\mathcal{A}$ *be an arbitrary symmetric point–to–point algorithm. Let* $\mathcal{G}_2$ *and* $\mathcal{G}_N$ *be global transition systems of* $\mathcal{A}$ *for some* $N \geq 2$. *Let* $\pi^2 = \kappa_0^2 \kappa_1^2 \ldots$ *be an admissible sequence of configurations in* $\mathcal{G}_2$. *There exists an admissible sequence* $\pi^N = \kappa_0^N \kappa_1^N \ldots$ *of configurations in* $\mathcal{G}_N$ *such that* κ_i^2 *is index projection of* κ_i^N *on indexes* $\{1, 2\}$ *for every* $i \geq 0$.

Lemma 5. *Let* $\mathcal{A}$ *be a symmetric point–to–point algorithm. Let* $\mathcal{G}_2$ *and* $\mathcal{G}_N$ *be its instances for some* $N \geq 2$. *Let* $AP_{\{1,2\}}$ *be a set of predicates that take one of the forms:* $Q_1(1)$, $Q_2(2)$, $Q_3(1, 2)$ *or* $Q_4(2, 1)$. *It follows that* $\mathcal{G}_2$ *and* $\mathcal{G}_N$ *are trace equivalent under* $AP_{\{1,2\}}$.

4.3 Cutoff Results Of symmetric point–to–point algorithms

In the following, we prove the cutoff result in Theorem 2 (see Page 10). A proof of another cutoff result in Theorem 1 is similar.

Proof sketch of Theorem 2. We have $\left(\forall \pi_N \in Path_N \colon \mathcal{G}_N, \pi_N \models \bigwedge_{i,j \in 1..N}^{i \neq j} \psi_{\{i,j\}}\right)$ $\Leftrightarrow$ $\left(\bigwedge_{i,j \in 1..N}^{i \neq j} (\forall \pi_N \in Path_N \colon \mathcal{G}_N, \pi_N \models \psi_{\{i,j\}})\right)$. Let i and j be two process indexes in a set $1..N$ such that $i \neq j$. It follows that $\alpha^1 = (i \leftrightarrow 1)$ and $\alpha^2 = (j \leftrightarrow 2)$ are transpositions on $1..N$ (*). By Lemma 2, we have: (i) $\psi_{\{\alpha^1(i),\alpha^2(j)\}} = \psi_{\{1,2\}}$, and (ii) $\alpha^2((\alpha^1(\mathcal{G}_N))) = \alpha^2(\mathcal{G}_N) = \mathcal{G}_N$, and (iii) $\alpha^2((\alpha^1(g_N^0))) = \alpha^2(g_N^0) = g_N^0$.

Since $\psi_{\{i,j\}}$ is an LTL\X formula, $\mathbf{A}\psi_{\{i,j\}}$ is a CTL*\X formula where $\mathbf{A}$ is a path operator in CTL*\X (see [9]). By the semantics of the operator $\mathbf{A}$, it follows $\forall \pi_N \in Path_N \colon \mathcal{G}_N, \pi_N \models \psi_{\{i,j\}}$ if and only if $\mathcal{G}_N, g_N^0 \models \mathbf{A}\psi_{\{i,j\}}$. By point (*), it follows $\mathcal{G}_N, g_N^0 \models \mathbf{A}\psi_{\{i,j\}}$ if and only if $\mathcal{G}_N, g_N^0 \models \mathbf{A}\psi_{\{1,2\}}$. We have that $\mathcal{G}_N, g_N^0 \models \bigwedge_{i,j \in 1..N}^{i \neq j} \mathbf{A}\psi(i,j)$ if and only if $\mathcal{G}_N, g_N^0 \models \mathbf{A}\psi(1,2)$, because both i and j are arbitrary and different. By the semantics of the operator $\mathbf{A}$, we have $\mathcal{G}_N, g_N^0 \models \mathbf{A}\psi(1,2)$ if and only if $\forall \pi_N \in Path_N \colon \mathcal{G}_N, \pi_N \models \psi(1,2)$. It follows $\forall \pi_N \in Path_N \colon \mathcal{G}_N, \pi_N \models \psi(1,2)$ if and only if $\forall \pi_2 \in Path_2 \colon \mathcal{G}_2, \pi_2 \models \psi(1,2)$ by Lemma 5. Then, Theorem 2 holds. □

5 Experiments

To demonstrate the feasibility of our approach, we specified the failure detector [5] in TLA$^+$ [22] [1]. Our specification follows the model of computation in Section 3. It is close to the pseudo-code in 1, except that these tasks are organized in a loop: task 1, task 2, and task 3. Moreover, our encoding contains the upper bounds on transmission time of messages and on the relative speeds of different processes, called Δ and Φ respectively. The user can verify our specification with different values of Δ and Φ by running model checkers TLC [28] and APALACHE [20]. Our experiments were set up in the synchronous case where $\Delta = 0$ and $\Phi = 1$. To reduce the state space, we apply abstractions to a global clock, local clocks, and received messages. Our abstractions are explained in detail in our TLA$^+$ specification.

We ran the following experiments on a laptop with a core i7-6600U CPU and 16GB DDR4. Table 1 presents the results in model checking the failure detectors [5] in the synchronous model. From the theoretical viewpoint, an instance with $N = 1$ is necessary, but we show only interesting cases with $N \geq 2$ in Table 1. (We did check an instance with $N = 1$, and there are no errors in this instance.) The strong accuracy property is the following safety property: $\mathbf{G}(\forall p, q \in 1..N \colon (Correct(p) \wedge Correct(q)) \Rightarrow \neg Suspected(p, q))$. The column "depth" shows the maximum execution length used by our tool as well as the maximum depth reached by TLC while running breadth-first search. For the second and forth benchmarks, we used the diameter bound that was reported by TLC, which does exhaustive state exploration. Hence, the verification results with APALACHE are complete. The abbreviation "TO" means timeout of 10 h.

[1] Our specification is available at https://github.com/banhday/netys20.git.

Table 1. Checking the failure detector [5] in the synchronous case

#	Property	N	Tool	Runtime	Memory	Depth
1	Strong accuracy	2	TLC	2 s	112M	36
2		2	APALACHE	1 m	1.12G	37
3		4	TLC	17 m	774 M	40
4		4	APALACHE	72 m	2.27G	41
5		6	TLC	TO	943M	2
6		6	APALACHE	TO	3M	31
7	Eventually strong accuracy	2	TLC	2 s	140M	36
8		4	TLC	20 m	683M	40
9		6	TLC	TO	839M	2
10	Strong completeness	2	TLC	2 s	134 M	36
11		4	TLC	23 m	678 M	40
12		6	TLC	TO	789M	3
13	Inductive invariant	2	TLC	20 s	192M	
14		2	APALACHE	1 m	674M	
15		3	TLC	TO	1.1G	
16		3	APALACHE	3 m	798M	
17		4	APALACHE	31 m	1.14G	

The inductive invariant is on the transition $\rightsquigarrow$, and contains type invariants, constraints on the age of in-transit messages, and constraints on when a process executes a task.

6 Conclusion

We have introduced the class of symmetric point-to-point algorithms that capture some well-known algorithms, e.g. failure detectors. The symmetric point-to-point algorithms enjoy the cutoff property. We have shown that checking properties of the form $\omega(i)$ has a cutoff of 1, and checking properties of the form $\psi(i, j)$ has a cutoff of 2 where $\omega(i)$ is an LTL\X formula whose predicates inspect only variables with a process index i, and $\psi(i, j)$ is an LTL\X formula whose predicates inspect only variables with two different process indexes $i \neq j$. We demonstrated the feasibility of our approach by specifying and model checking the failure detector by Chandra and Toueg under synchrony with two model checkers TLC and APALACHE.

We see two directions for future work. The first is to find new cutoffs for checking other properties in symmetry point-to-point algorithms. For example, given a correctness property with k universal quantifiers over process index variables, we conjecture that checking k small instances whose size is less than or equal to k is sufficient to reason about the correctness of all instances. The second

is to extend our results to the model of computation under partial synchrony. This model has additional time constraints on message delay Δ and the relative process speed Φ. Algorithms under partial synchrony are parameterized by Δ and Φ. We explore techniques to deal with these parameters.

References

1. Apt, K., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. IPL **15**, 307–309 (1986)
2. Bailis, P., Kingsbury, K.: The network is reliable. Queue **12**(7), 20–32 (2014)
3. Bloem, R., et al.: Decidability of parameterized verification. Syn. Lect. Dist. Comput. Theory **6**(1), 1–170 (2015)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
6. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. CAV. LNCS **2404**, 359–364 (2002)
7. Clarke, E., Talupur, M., Veith, H.: Proving ptolemy right: the environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_4
8. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_18
9. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model checking. MIT press (2018)
10. Debrat, H., Merz, S.: Verifying fault-tolerant distributed algorithms in theheard-of model. Archive of Formal Proofs **2012** (2012)
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
12. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 236–254. Springer, Heidelberg (2000). https://doi.org/10.1007/10721959_19
13. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_22
14. Emerson, E.A., Kahlon, V.: Parameterized model checking of ring-based message passing systems. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 325–339. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30124-0_26
15. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 85–94 (1995)

16. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S., Zill, B.: Ironfleet: Proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017)
17. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997)
18. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_55
19. Kingsbury, K.: Jepsen: Testing the partition tolerance of postgresql, redis, mongodb and riak, 2013
20. Konnov, I., Kukovec, J., Tran, T.H.: TLA$^+$ model checking made symbolic. Proceedings of the ACM on Programming Languages 3(OOPSLA), 1–30 (2019)
21. Kurshan, R.P., McMillan, K.: A structural induction theorem for processes. In: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing. pp. 239–247 (1989)
22. Lamport, L.: Specifying systems: The TLA$^+$ language and tools for hardwareand software engineers. Addison-Wesley (2002)
23. 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
24. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Comm. ACM **58**(4), 66–73 (2015)
25. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_7
26. Schiper, N., Rahli, V., Van Renesse, R., Bickford, M., Constable, R.L.: Developing correctly replicated databases using formal tools. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 395–406. IEEE (2014)
27. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)
28. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_6