A Case Study on Parametric Verification of Failure Detectors

Thanh-Hai Tran, Igor Konnov, Josef Widder

A case study on parametric verification of failure detectors

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

[1] TU Wien, Austria
[2] Informal Systems, Austria

Abstract. Partial synchrony is a model of computation in many distributed algorithms and modern blockchains. Correctness of these algorithms requires the existence of bounds on message delays and on the relative speed of processes after reaching Global Stabilization Time (GST). This makes partially synchronous algorithms parametric in time bounds, which renders automated verification of partially synchronous algorithms challenging. In this paper, we present a case study on formal verification of both safety and liveness of a Chandra and Toueg failure detector that is based on partial synchrony. To this end, we specify the algorithm and the partial synchrony assumptions in three frameworks: TLA$^+$, Ivy, and counter automata. Importantly, we tune our modeling to use the strength of each method: (1) We are using counters to encode message buffers with counter automata, (2) we are using first-order relations to encode message buffers in Ivy, and (3) we are using both approaches in TLA$^+$. By running the tools for TLA$^+$ (TLC and APALACHE) and counter automata (FAST), we demonstrate safety for fixed time bounds. This helped us to find the inductive invariants for fixed parameters, which we used as a starting point for the proofs with Ivy. By running Ivy, we prove safety for arbitrary time bounds. Moreover, we show how to verify liveness of the failure detector by reducing the verification problem to safety verification. Thus, both properties are verified by developing inductive invariants with Ivy. We conjecture that correctness of other partially synchronous algorithms may be proven by following the presented methodology.

Keywords: Failure detectors · TLA$^+$ · Counter automata · FAST · Ivy.

1 Introduction

Distributed algorithms play a crucial role in modern infrastructure, but they are notoriously difficult to understand and to get right. Network topologies, message delays, faulty processes, the relative speed of processes, and fairness conditions might lead to behaviors that were neither intended nor anticipated by algorithm designers. Hence, many specification and verification techniques for distributed algorithms [23,28,30,14] have been developed.

Verification techniques for distributed algorithms usually focus on two models of computation: synchrony [32] and asynchrony [19,21]. Synchrony is hard

to implement in real systems, while many basic problems in fault-tolerant distributed computing are unsolvable in asynchrony.

Partial synchrony lies between synchrony and asynchrony, and escapes their shortcomings. To guarantee liveness properties, proof-of-stake blockchains [9,36] and distributed algorithms [11,8] assume time constraints under partial synchrony. That is the existence of bounds Δ on message delay, and Φ on the relative speed of processes after some time point. This combination makes partially synchronous algorithms parametric in time bounds. While partial synchrony is important for system designers, it is challenging for verification.

We thus investigate verification of distributed algorithms under partial synchrony, and start with the specific class of failure detectors: a Chandra and Toueg failure detector [11]. This is a well-known algorithm under partial synchrony that provides a service that can be used to solve many problems in fault-tolerant distributed computing.

Contributions. In this paper, we do parametric verification of both safety and liveness of the Chandra and Toueg failure detector in case of unknown bounds Δ and Φ. In this case, both Δ and Φ are arbitrary, and the constraints on message delay and the relative speeds hold in every execution from the start.

1. We extend the cutoff results in [35] for partial synchrony. In a nutshell, a cutoff for a parameterized algorithm $\mathcal{A}$ and a property ϕ is a number k such that ϕ holds for every instance of $\mathcal{A}$ if and only if ϕ holds for instances with k processes [16,7]. While the cutoff results [35] are for synchrony or asynchrony, our results are for partial synchrony. Hence, we verify the Chandra and Toueg failure detector under partial synchrony by checking instances with two processes.
2. We introduce the encoding techniques to efficiently specify the failure detector based on our cutoff results. These techniques can tune our modeling to use the strength of the tools: FAST, Ivy, and model checkers for TLA$^+$.
3. We demonstrate how to reduce the liveness properties Eventually Strong Accuracy, and Strong Completeness to safety properties.
4. We check the safety property Strong Accuracy, and the mentioned liveness properties on instances with fixed parameters by using FAST, and model checkers for TLA$^+$.
5. To verify cases of arbitrary bounds Δ and Φ, we find and prove inductive invariants of the failure detector with the interactive theorem prover Ivy. We reduce the liveness properties to safety properties by applying the mentioned techniques. While our specifications are not in the decidable theories that Ivy supports, Ivy requires no additional user assistance to prove most of our inductive invariants.

Related work. Research papers about partially synchronous algorithms, including papers about failure detectors [25,2,1] contain manual proofs and no formal specifications. Without these details, proving those distributed algorithms with interactive theorem provers [13,30] is impossible. To test a candidate I for an inductive invariant with fixed parameters, system designers can apply probabilistic random checking with TLA$^+$ and TLC [24]. However, this approach ran-

domly explores a subset of the execution space. Hence, it can show a counterexample to induction, but cannot prove that I is an inductive invariant. We prove inductive invariants in small cases with the model checker APALACHE [18]. System designers can use timed automata [3] and parametric verification frameworks [26,4,27] to specify and verify timed systems. In the context of timed systems, we are aware of only one paper about verification of failure detectors [5]. In this paper, the authors used three tools, namely UPPAAL [26], mCRL2 [10], and FDR2 [31] to verify small instances of a failure detector based on a logical ring arrangement of processes. Their verification approach required that message buffers were bounded, and had restricted behaviors in the specifications. Moreover, they did not consider the bound Φ on the relative speed of processes. In contrast, there are no restrictions on message buffers, and no ring topology in the Chandra and Toueg failure detector. Moreover, our work is to verify the Chandra and Toueg failure detector in case of arbitrary bounds. In recent years, automatic parameterized verification techniques [20,32,14] have been introduced for distributed systems, but they are designed for synchronous and/or asynchronous models. Interactive theorem provers have been used to prove correctness of distributed algorithms recently. For example, researchers proved safety of Tendermint consensus with Ivy [17].

Structure. In Section 2, we summarize the Chandra and Toueg failure detector, and the cutoff results in [35]. In Section 3, we extend the cutoff results in [35] for partial synchrony. Our encoding technique is presented in Section 4. In Section 5, we present how to reduce the mentioned liveness properties to safety ones. Experiments for small Δ and Φ are described in Section 6. Ivy proofs for parametric Δ and Φ are discussed in Section 7.

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

2 Preliminaries

This section describes the Chandra and Toueg failure detector [11], and the cutoff results [35] that we can extend to this failure detector under partial synchrony.

A failure detector can be seen as an oracle to get information about crash failures in the distributed system. A failure detector usually guarantees some of the following properties [11] (numbers $1..N$ denote the process identifiers):

- Strong Accuracy: No process is suspected before it crashes.
  $$\mathbf{G}(\forall p, q \in 1..N : (\mathit{Correct}(p) \wedge \mathit{Correct}(q)) \Rightarrow \neg \mathit{Suspected}(p, q))$$
- Eventual Strong Accuracy: There is a time after which correct processes are not suspected by any correct process.
  $$\mathbf{F}\,\mathbf{G}(\forall p, q \in 1..N : (\mathit{Correct}(p) \wedge \mathit{Correct}(q)) \Rightarrow \neg \mathit{Suspected}(p, q))$$
- Strong Completeness: Eventually every crashed process is permanently suspected by every correct process.
  $$\mathbf{F}\,\mathbf{G}(\forall p, q \in 1..N : (\mathit{Correct}(p) \wedge \neg \mathit{Correct}(q)) \Rightarrow \mathit{Suspected}(p, q))$$

where $\mathbf{F}$ and $\mathbf{G}$ are operators in LTL (linear temporal logic), predicate $\mathit{Suspect}(p, q)$ refers to whether process p suspects process q in crashing, and predicate $\mathit{Correct}(p)$

Algorithm 1 The eventually perfect failure detector algorithm in [11]

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

refers to whether process p is correct. However, process p might crash later (and not recover). A crashed process p satisfies $\neg Correct(p)$.

Algorithm 1 presents the pseudo-code of the failure detector of [11]. A system instance has N processes that communicate with each other by sending-to-all and receiving messages through unbounded N^2 point-to-point communication channels. A process performs local computation based on received 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 5), and (ii) repeatedly produces predictions about crashes of other processes based on timeouts (Line 6), and (iii) increases a timeout for process q if p has learned that its suspicion on q is wrong (Line 9). Notice that process p raises suspicion on the operation of process q (Line 6) by considering only information related to q: $timeout\,[p, q]$, $suspected\,[p, q]$, and messages that p has received from q recently.

Algorithm 1 does not satisfy Eventually Strong Accuracy under asynchrony since there exists no bound on message delay, and messages sent by correct processes might always arrive after the timeout expired. Liveness of the failure detector is based on the existence of bounds Δ on the message delay, and Φ on the relative speed of processes after reaching the global stabilization at some time point T_0 [11]. There are many models of partial synchrony [15,11]. In this paper, we focus only on the case of unknown bounds Δ and Φ because other models might call for abstractions. In this case, $T_0 = 1$, and both parameters Δ and Φ are arbitrary. Moreover, the following constraints hold in every execution:

– Constraint 1: If message m is placed in the message buffer from process q to process p by some $Send(m, p)$ at a time $s_1 \geq 1$, and if process p executes a $Receive(p)$ at a time s_2 with $s_2 \geq s_1 + \Delta$, then message m must be delivered to p at time s_2 or earlier.

– Constraint 2: In every contiguous time interval $[t, t + \Phi]$ with $t \geq 1$, every correct process must take at least one step.

These constraints make the failure detector parametric in Δ and Φ.

Moreover, Algorithm 1 is parameterized by the initial value of the timeout. If a default value of the timeout is too small, there exists a case in which sent messages are delivered after the timeout expired. It violates Strong Accuracy.

In [35], Thanh-Hai et al. defined a class of symmetric point-to-point distributed algorithms that contains the failure detector [11], and proved cutoffs on the number of processes for this class under asynchrony. These cutoff results guarantee that analyzing instances with two processes is sufficient to reason about the correctness of all instances of the Chandra and Toueg failure detector under asynchrony. In the following section, we will generalize this result to partial synchrony, which allows us to verify the mentioned properties on the failure detector by checking instances with only two processes.

3 Cutoffs of the failure detector

In this section, we extend the cutoffs of symmetric point-to-point distributed algorithms in [35] for partial synchrony.

Notice that time parameters in partial synchrony only reduce the execution space compared to asynchrony. Hence, we can formalize the system behaviors under partial synchrony by extending the formalization of the system behaviors under asynchrony in [35] with the notion of time, message ages, time constraints under partial synchrony. (Our formalization is left for the full report [33].)

In a nutshell, our cutoff results for the symmetric point-to-point class allow us to verify the mentioned properties on the failure detector under partial synchrony by checking small instances with one and/or two processes. Intuitively, the proofs of our cutoffs are based on the following observations:

– The global transition system and the desired property are symmetric [35].
– Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be two instances of a symmetric point-to-point algorithm with 2 and N processes, respectively. By [35], two instances $\mathcal{G}_2$ and $\mathcal{G}_N$ are trace equivalent under a set of predicates in the desired property.
– We will now discuss that the constraints maintain partial synchrony. Let π_N be an execution in $\mathcal{G}_N$. We construct an execution π_2 in $\mathcal{G}_2$ by applying the index projection to π_N (formally defined in [35]). Intuitively, the index projection discards processes $3..N$ as well as their corresponding messages and buffers. Moreover, for every $k, \ell \in \{1, 2\}$, the index projection preserves (i) at which point in time process k takes a step, and (ii) what action process k takes at a time $t \geq 0$, and (iii) messages from process k to process ℓ. Figure 1 demonstrates an execution in $\mathcal{G}_2$ that is constructed based on a given execution in $\mathcal{G}_3$ with the index projection. Observe that Constraints 1 and 2 are maintained in this projection.
– Let π_2 be an execution in $\mathcal{G}_2$. We construct an execution π_N in $\mathcal{G}_N$ based on π_2 such that all processes $3..N$ crash from the beginning, and π_2 is an index projection of π_N [35]. For example, Figure 2 demonstrates an execution in

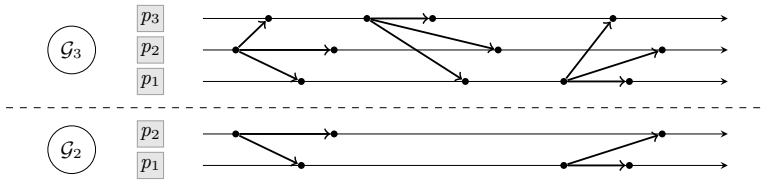

Fig. 1: Given execution in $\mathcal{G}_3$, construct an execution in $\mathcal{G}_2$ by index projection.

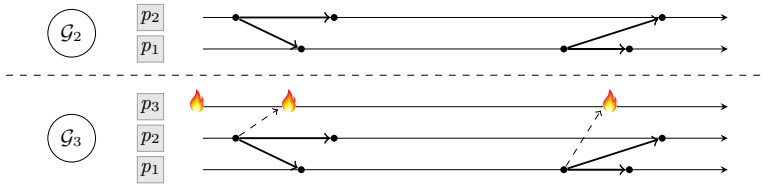

Fig. 2: Construct an execution in $\mathcal{G}_3$ based on a given execution in $\mathcal{G}_2$.

$\mathcal{G}_3$ that is constructed based on an given execution in $\mathcal{G}_2$. If Constraints 1 and 2 hold on π_2, these constraints also hold on π_N.

4 Encoding the Chandra and Toueg failure detector

In this section, we first discuss why it is sufficient to verify the failure detector by checking a system with only one sender and one receiver by applying the cutoffs presented in Section 3. Next, we introduce two approaches to encoding the message buffer, and an abstraction of in-transit messages that are older than Δ time-units. Finally, we present how to encode the relative speed of processes with counters over natural numbers. These techniques allow us to tune our models to the strength of the verification tools: FAST, Ivy, and model checkers for TLA$^+$.

4.1 The system with one sender and one receiver

We discussed our cutoff results in Section 3. These results allow us to verify the Chandra and Toueg failure detector under partial synchrony by checking only instances with two processes. In the following, we discuss the model with two processes, and formalize the properties with two-process indexes. By process symmetry, it is sufficient to verify Strong Accuracy, Eventually Strong Accuracy, and Strong Completeness by checking the following properties.

$$\mathbf{G}((\mathit{Correct}(1) \wedge \mathit{Correct}(2)) \Rightarrow \neg \mathit{Suspected}(2,1)) \tag{1}$$

$$\mathbf{F}\,\mathbf{G}((\mathit{Correct}(1) \wedge \mathit{Correct}(2)) \Rightarrow \neg \mathit{Suspected}(2,1)) \tag{2}$$

$$\mathbf{F}\,\mathbf{G}((\neg\mathit{Correct}(1) \wedge \mathit{Correct}(2)) \Rightarrow \mathit{Suspected}(2,1)) \tag{3}$$

We can take a further step towards facilitating verification of the failure detector. First, every process typically has a local variable to store messages that

it needs to send to itself, instead of using a real communication channel. Hence, we can assume that there is no delay for those messages, and that each correct process never suspects itself. Second, local variables in Algorithm 1 are arrays whose elements correspond one-to-one with a remote process, e.g., $timeout[2,1]$ and $suspected[2,1]$. Third, communication between processes is point-to-point. When this is not the case, one can use cryptography to establish one-to-one communication. Hence, reasoning about Properties 1–3 requires no information about messages from process 1 to itself, local variables of process 1, and messages from process 2.

Due to the above characteristics, it is sufficient to consider process 1 as a sender, and process 2 as a receiver. In detail, the sender follows Task 1 in Algorithm 1, but does nothing in Task 2 and Task 3. The sender does not need the initialization step, and local variables *suspected* and *timeout*. In contrast, the receiver has local variables corresponding to the sender, and follows only the initialization step, and Task 2, and Task 3 in Algorithm 1. The receiver can increase its waiting time in Task 1, but does not send any message.

4.2 Encoding the message buffer

Algorithm 1 assumes unbounded message buffers between processes that produce an infinite state space. Moreover, a sent message might be in-transit for a long time before it is delivered. We first introduce two approaches to encode the message buffer based on a logical predicate, and a counter over natural numbers. The first approach works for TLA$^+$ and Ivy, but not for counter automata (FAST). The latter is supported by all mentioned tools, but it is less efficient as it requires more transitions. Then, we present an abstraction of in-transit messages that are older than Δ time-units. This technique reduces the state space, and allows us to tune our models to the strength of the verification tools.

Encoding the message buffer with a predicate. In Algorithm 1, only "alive" messages are sent, and the message delivery depends only on the age of in-transit messages. Moreover, the computation of the receiver does not depend on the contents of its received messages. Hence, we can encode a message buffer by using a logical predicate $\mathsf{existsMsgOfAge}(x)$. For every $k \geq 0$, predicate $\mathsf{existsMsgOfAge}(k)$ refers to whether there exists an in-transit message that is k time-units old. The number 0 refers to the age of a fresh message in the buffer.

It is convenient to encode the message buffer's behaviors in this approach. For instance, Formulas 4 and 5 show constraints on the message buffer when a new message is sent:

$$\mathsf{existsMsgOfAge}'(0) = \top \tag{4}$$

$$\forall x \in \mathbb{N} \,.\, x > 0 \Rightarrow \mathsf{existsMsgOfAge}'(x) = \mathsf{existsMsgOfAge}(x) \tag{5}$$

where $\mathsf{existsMsgOfAge}'$ refers to the value of $\mathsf{existsMsgOfAge}$ in the next state. Formula 4 implies that a fresh message has been added to the message buffer. Formula 5 ensures that other in-transit messages are unchanged.

Fig. 3: The message buffer after increasing message ages in case of $\texttt{buf} = 6$

Another example is the relation between existsMsgOfAge and existsMsgOfAge′ after the message delivery. This relation is formalized with Formulas 6–9. Formula 6 requires that there exists an in-transit message in existsMsgOfAge that can be delivered. Formula 7 ensures that no old messages are in transit after the delivery. Formula 8 guarantees that no message is created out of thin air. Formula 9 implies that at least one message is delivered.

$$\exists x \in \mathbb{N} \,.\, \mathsf{existsMsgOfAge}(x) \tag{6}$$

$$\forall x \in \mathbb{N} \,.\, x \geq \Delta \Rightarrow \neg \mathsf{existsMsgOfAge}'(x) \tag{7}$$

$$\forall x \in \mathbb{N} \,.\, \mathsf{existsMsgOfAge}'(x) \Rightarrow \mathsf{existsMsgOfAge}(x) \tag{8}$$

$$\exists x \in \mathbb{N} \,.\, \mathsf{existsMsgOfAge}'(x) \neq \mathsf{existsMsgOfAge}(x) \tag{9}$$

This encoding works for TLA$^+$ and Ivy, but not for FAST, because the input language of FAST does not support functions.

Encoding the message buffer with a counter. In the following, we present an encoding technique for the buffer that can be applied in all tools TLA$^+$, Ivy, and FAST. This approach encodes the message buffer with a counter buf over natural numbers. The k^{th} bit refers to whether there exists an in-transit message with k time-units old.

In this approach, message behaviors are formalized with operations in Presburger arithmetic. For example, assume $\Delta > 0$, we write $\texttt{buf}' = \texttt{buf} + 1$ to add a fresh message in the buffer. Notice that the increase of buf by 1 turns on the 0^{th} bit, and keeps the other bits unchanged.

To encode the increase of the age of every in-transit message by 1, we simply write $\texttt{buf}' = \texttt{buf} \times 2$. Assume that we use the least significant bit (LSB) first encoding, and the left-most bit is the 0^{th} bit. By multiplying buf by 2, we have updated buf′ by shifting to the right every bit in buf by 1. For example, Figure 3 demonstrates the message buffer after the increase of message ages in case of $\texttt{buf} = 6$. We have $\texttt{buf}' = \texttt{buf} \times 2 = 12$. It is easy to see that the 1^{st} and 2^{nd} bits in buf are on, and the 2^{nd} and 3^{rd} bits in buf′ are on.

Recall that Presburger arithmetic does not allow one to divide by a variable. Therefore, to guarantee the constraint in Formula 8, we need to enumerate all constraints on possible values of buf and buf′ after the message delivery. For example, assume $\texttt{buf} = 3$, and $\Delta = 1$. After the message delivery, buf′ is either 0 or 1. If $\texttt{buf} = 2$ and $\Delta = 1$, buf′ must be 0 after the message delivery. Importantly, the number of transitions for the message delivery depends on the value of Δ.

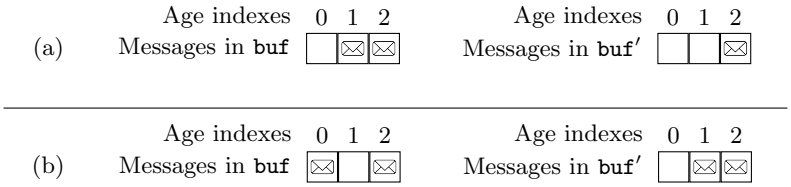

Fig. 4: The increase of message ages with the abstraction of old messages. In the case (a), we have $\Delta = 2$, $\texttt{buf} = 6$, and $\texttt{buf}' = 4$. In the case (b), we have $\Delta = 2$, $\texttt{buf} = 5$, and $\texttt{buf}' = 6$.

To avoid the enumeration of all possible cases, Formula 8 can be rewritten with bit-vector arithmetic. However, bit-vector arithmetic are currently not supported in all verification tools TLA$^+$, FAST, and Ivy.

The advantage of this encoding is that when bound Δ is fixed, every constraint in the system behaviors can be rewritten in Presburger arithmetic. Thus, we can use FAST, which accepts constraints in Presburger arithmetic. To specify cases with arbitrary Δ, the user can use TLA$^+$ or Ivy.

Abstraction of old messages. Algorithm 1 assumes underlying unbounded message buffers between processes. Moreover, a sent message might be in transit for a long time before it is delivered. To reduce the state space, we develop an abstraction of in-transit messages that are older than Δ time-units; we call such messages "old". This abstraction makes the message buffer between the sender and the receiver bounded. In detail, the message buffer has a size of Δ. Importantly, we can apply this abstraction to two above encoding techniques for the message buffer.

In partial synchrony, if process p executes Receive at some time point from the Global Stabilization Time, *every* old message sent to p will be delivered immediately. Moreover, the computation of a process in Algorithm 1 does not depend on the content of received messages. Hence, instead of tracking all old messages, our abstraction keeps only one old message that is Δ time-units old, does not increase its age, and throws away other old messages.

In the following, we discuss how to integrate this abstraction into the encoding techniques of the message buffer. We demonstrate our ideas by showing the pseudo-code of the increase of message ages. It is straightforward to adopt this abstraction to the message delivery, and to the sending of a new message.

Figure 4(a) presents the increase of message ages with this abstraction in a case of $\Delta = 2$, and $\texttt{buf} = 6$. Unlike Figure 3, there exists no in-transit message that is 3 time-units old in Figure 4(a). Moreover, the message buffer in Figure 4(a) has a size of 3. In addition, $\texttt{buf}'$ has only one in-transit message that is 2 time-units old. We have $\texttt{buf}' = 4$ in this case. Figure 4(b) demonstrates another case of $\Delta = 2$, $\texttt{buf} = 5$, and $\texttt{buf}' = 6$.

1: **if** $\texttt{buf} < 2^{\Delta}$ **then** $\texttt{buf}' \leftarrow \texttt{buf} \times 2$
2: **else**
3: **if** $\texttt{buf} \geq 2^{\Delta} + 2^{\Delta-1}$ **then** $\texttt{buf}' \leftarrow \texttt{buf} \times 2 - 2^{\Delta+1}$
4: **else** $\texttt{buf}' \leftarrow \texttt{buf} \times 2 - 2^{\Delta+1} + 2^{\Delta}$

Fig. 5: Encoding the increase of message ages with a counter `buf`, and the abstraction of old messages.

Formally, Figure 5 presents the pseudo-code of the increase of message ages that is encoded with a counter `buf`, and the abstraction of old messages. There are three cases. In the first case (Line 1), there exist no old messages in `buf`, and we simply set $\texttt{buf}' = \texttt{buf} \times 2$. In other cases (Lines 3 and 4), `buf` contains an old message. Figure 4(a) demonstrates the second case (Line 3). We subtract $2^{\Delta+1}$ to remove an old message with $\Delta+1$ time-units old from the buffer. Figure 4(b) demonstrates the third case (Line 4). In the third case, we also need to remove an old message with $\Delta+1$ time-units old from the buffer. Moreover, we need to put an old message with Δ time-units old to the buffer by adding 2^{Δ}.

Now we discuss how to integrate the abstraction of old messages in the encoding of the message buffer with a predicate. Formulas 10–13 present the relation between $\mathsf{existsMsgOfAge}$ and $\mathsf{existsMsgOfAge}'$ when message ages are increased by 1, and this abstraction is applied. Formula 10 ensures that no fresh message will be added to $\mathsf{existsMsgOfAge}'$. Formula 11 ensures that the age of every message that is until $(\Delta - 2)$ time-units old will be increased by 1. Formulas 12–13 are introduced by this abstraction. Formula 12 implies that if there exists an old message or a message with $(\Delta - 1)$ time-units old in $\mathsf{existsMsgOfAge}$, there will be an old message that is Δ time-units old in $\mathsf{existsMsgOfAge}'$. Formula 13 ensures that there exists no message that is older than Δ time-units old.

$$\neg\mathsf{existsMsgOfAge}'(0) \tag{10}$$

$$\forall x \in \mathbb{N} \,.\, (0 \leq x \leq \Delta - 2) \Rightarrow \mathsf{existsMsgOfAge}'(x+1) = \mathsf{existsMsgOfAge}(x) \tag{11}$$

$$\mathsf{existsMsgOfAge}'(\Delta) = \mathsf{existsMsgOfAge}(\Delta) \vee \mathsf{existsMsgOfAge}(\Delta - 1) \tag{12}$$

$$\forall x \in \mathbb{N} \,.\, x > \Delta \Rightarrow \mathsf{existsMsgOfAge}'(x) = \bot \tag{13}$$

4.3 Encoding the relative speed of processes

Recall that we focus on the case of unknown bounds Δ and Φ. In this case, every correct process must take at least one step in every contiguous time interval containing Φ time-units [15].

To maintain this constraint on executions generated by the verification tools, we introduced two additional control variables sTimer and rTimer for the sender and the receiver, respectively. These variables work as timers to keep track of how long a process has not taken a step, and when a process can take a step. Since these timers play similar roles, we here focus on rTimer. In our encoding, only the environment can update rTimer. To schedule the receiver, the environment

non-deterministically executes one of two actions: (i) resets rTimer to 0, and (ii) if $\mathsf{rTimer} < \Phi$, increases rTimer by 1. Moreover, the receiver must take a step whenever $\mathsf{rTimer} = 0$.

5 Reduce liveness properties to safety properties

To verify the liveness properties Eventually Strong Accuracy and Strong Completeness with Ivy, we first need to reduce them to safety properties. Intuitively, these liveness properties are bounded; therefore, they become safety ones. This section demonstrates how to reduce Eventually Strong Accuracy to a safety one.

By cutoffs discussed in Section 3, it is sufficient to verify Eventually Strong Accuracy on the Chandra and Toueg failure detector by checking the following property on instances with 2 processes.

$$\mathbf{F}\,\mathbf{G}((\mathit{Correct}(1) \wedge \mathit{Correct}(2)) \Rightarrow \neg \mathit{Suspected}(2,1)) \tag{14}$$

In the failure detector [11], the receiver suspects the sender only if its waiting time reaches the timeout (see Line 6 in Algorithm 1). To reduce Formula 14 to a safety property, we found a specific guard g for timeout such that if $\mathsf{timeout} \geq g$ and the sender is correct, then $\texttt{waitingtime} < g$. Hence, it is sufficient to verify Formula 14 by checking the following property.

$$\mathbf{G}\left(\mathsf{timeout} \geq g \Rightarrow ((\mathit{Correct}(1) \wedge \mathit{Correct}(2)) \Rightarrow \neg \mathit{Suspected}(2,1))\right)$$

6 Experiments for small Δ and Φ

In this section, we describe our experiments with TLA$^+$ and FAST. We ran the following experiments on a virtual machine with Core i7-6600U CPU and 8GB DDR4. Our specifications can be found at [34].

6.1 Model checkers for TLA$^+$: TLC and APALACHE

In our work, we use TLA$^+$ [23] to specify the failure detector with both encoding techniques for the message buffer, and the abstraction in Section 4. Then, we use the model checkers TLC [37] and APALACHE [18] to verify instances with fixed bounds Δ and Φ, and the GST $T_0 = 1$. This approach helps us to search constraints in inductive invariants in case of fixed parameters. The main reason is that counterexamples and inductive invariants in case of fixed parameters, e.g., $\Delta \leq 1$ and $\Phi \leq 1$, are simpler than in case of arbitrary parameters. Hence, if a counterexample is found, we can quickly analyze it, and change constraints in an inductive invariant candidate. After obtaining inductive invariants in small cases, we can generalize them for cases of arbitrary bounds, and check with theorem provers, e.g., Ivy (Section 7).

TLA$^+$ offers a rich syntax for sets, functions, tuples, records, sequences, and control structures [23]. Hence, it is straightforward to apply the encoding

```
1: SSnd ≜  ∧ ePC = "SSnd"
2:         ∧ IF (sTimer = 0 ∧ sPC = "SSnd")
3:           THEN buf' = buf + 1
4:           ELSE UNCHANGED buf
5:         ∧ ePC' = "RNoSnd"
6:         ∧ UNCHANGED ⟨sTimer, rTimer...⟩
```

Fig. 6: Sending a new message in TLA$^+$ in case of $\Delta > 0$

techniques and the abstraction presented in Section 4 in TLA$^+$. For example, Figure 6 represents a TLA$^+$ action *SSnd* for sending a new message in case of $\Delta > 0$. Variables *ePC* and *sPC* are program counters for the environment and the sender, respectively. Line 1 is a precondition, and refers to that the environment is in subround Send. Lines 2–3 say that if the sender is active in subround Send, the counter buf' is increased by 1. Otherwise, two counters *buf* and buf' are the same (Line 4). Line 5 implies that the environment is still in the subround Send, but it is now the receiver's turn. Line 6 guarantees that other variables are unchanged in this action. (The details are left for the full report [33].)

Now we present the experiments with TLC and APALACHE. We used these tools to verify (i) the safety property Strong Accuracy, and (ii) an inductive invariant for Strong Accuracy, and (iii) an inductive invariant for a safety property reduced from the liveness property Strong Completeness in case of fixed bounds, and GST = 1 (initial stabilization). The structure of the inductive invariants verified here are very close to one in case of arbitrary bounds Δ and Φ.

Table 1 shows the results in verification of Strong Accuracy in case of the initial stabilization, and fixed bounds Δ and Φ. Table 1 shows the experiments with the three tools TLC, APALACHE, and FAST. The column "#states" shows the number of distinct states explored by TLC. The column "#depth" shows the maximum execution length reached by TLC and APALACHE. The column "buf" shows how to encode the message buffer. The column "LOC" shows the number of lines in the specification of the system behaviors (without comments). The symbol "-" (minus) refers to that the experiments are intentionally missing since FAST does not support the encoding of the message buffer with a predicate. The abbreviation "pred" refers to the encoding of the message buffer with a predicate. The abbreviation "cntr" refers to the encoding of the message buffer with a counter. The abbreviation "TO" means a timeout of 6 hours. In these experiments, we initially set $\mathsf{timeout} = 6 \times \Phi + \Delta$, and Strong Accuracy is satisfied. The experiments show that TLC finishes its tasks faster than the others, and APALACHE prefers the encoding of the message buffer with a predicate.

Table 2 summarizes the results in verification of Strong Accuracy with the tools TLC, APALACHE, and FAST in case of the initial stabilization, and small bounds Δ and Φ, and initially $\mathsf{timeout} = \Delta + 1$. Since timeout is initialized with a too small value, there exists a case in which sent messages are delivered

Table 1: Showing Strong Accuracy for fixed parameters.

#	Δ	Φ	buf	TLC				APALACHE		FAST	
				time	#states	depth	LOC	time	depth	time	LOC
1	2	4	pred	3s	10.2K	176	190	8m	176	-	-
2			cntr	3s	10.2K	176	266	9m	176	16m	387
3	4	4	pred	3s	16.6K	183	190	12m	183	-	-
4			cntr	3s	16.6K	183	487	35m	183	TO	2103
5	4	5	pred	3s	44.7K	267	190	TO	222	-	-
6			cntr	3s	44.7K	267	487	TO	223	TO	2103

Table 2: Violating Strong Accuracy for fixed parameters.

#	Δ	Φ	buf	TLC			APALACHE		FAST
				time	#states	depth	time	depth	time
1	2	4	pred	1s	840	43	11s	42	-
2			cntr	1s	945	43	12s	42	10m
3	4	4	pred	2s	1.3K	48	15s	42	-
4			cntr	2s	2.4K	56	16s	42	TO
5	20	20	pred	TO	22.1K	77	1h15m	168	-

after the timeout expires. The tools reported an error execution where Strong Accuracy is violated. In these experiments, APALACHE is the winner. The abbreviation "TO" means a timeout of 6 hours. The meaning of other columns and abbreviations is the same as in Table 1.

Table 3 shows the results in verification of inductive invariants for Strong Accuracy and Strong Completeness with TLC and APALACHE in case of the initial stabilization, and slightly larger bounds Δ and Φ. The message buffer was encoded with a predicate in these experiments. In these experiments, inductive invariants hold, and APALACHE is faster than TLC in verifying them.

As one sees from the tables, APALACHE is fast at proving inductive invariants, and at finding a counterexample when a desired safety property is violated. TLC is a better option in cases where a safety property is satisfied.

In order to prove correctness of the failure detector in cases where parameters Δ and Φ are arbitrary, the user can use the interactive theorem prover TLA$^+$ Proof System (TLAPS) [12]. A shortcoming of TLAPS is that it does not provide a counterexample when an inductive invariant candidate is violated.

Table 3: Proving inductive invariants with TLC and APALACHE.

#	Δ	Φ	Property	TLC		APALACHE
				time	#states	time
1	4	40	Strong Accuracy	33m	347.3M	12s
2	4	10	Strong Completeness	44m	13.4M	17s

```
1: transition SSnd_Active := {
2:     from := incMsgAge;
3:     to := ssnd;
4:     guard := sTimer = 0;
5:     action := buf′ = buf + 1; };
```

Fig. 7: Sending a new message in FAST in case of $\Delta > 0$

Moreover, proving the failure detector with TLAPS requires more human effort than with Ivy. Therefore, we provide Ivy proofs in Section 7.

6.2 FAST

A shortcoming of the model checkers TLC and APALACHE is that parameters Δ and Φ must be fixed before running these tools. FAST is a tool designed to reason about safety properties of counter systems, i.e. automata extended with unbounded integer variables [6]. If Δ is fixed, and the message buffer is encoded with a counter, the failure detector becomes a counter system. We specified the failure detector in FAST, and made experiments with different parameter values to understand the limit of FAST: (i) the initial stabilization, and small bounds Δ and Φ, and (ii) the initial stabilization, fixed Δ, but unknown Φ.

Figure 7 represents a FAST transition for sending a new message in case of $\Delta > 0$. Line 2 describes the (symbolic) source state of the transition, and region `incMsgAge` is a set of configurations in the failure detector that is reachable from a transition for increasing message ages. Line 3 mentions the (symbolic) destination state of the transition, and region `sSnd` is a set of configurations in the failure detector that is reachable from a transition named "SSnd_Active" for sending a new message. Line 4 represents the guard of this transition. Line 5 is an action. Every unprimed variable that is not written in Line 5 is unchanged.

The input language of FAST is based on Presburger arithmetics for both system and properties specification. Hence, we cannot apply the encoding of the message buffer with a predicate in FAST.

Tables 1 and 2 described in the previous subsection summarize the experiments with FAST, and other tools where all parameters are fixed. Moreover, we ran FAST to verify Strong Accuracy in case of the initial stabilization, $\Delta \leq 4$, and arbitrary Φ. FAST is a semi-decision procedure; therefore, it does not terminate on some inputs. Unfortunately, FAST could not prove Strong Accuracy in case of arbitrary Φ, and crashed after 30 minutes.

7 Ivy proofs for parametric Δ and Φ

While TLC, APALACHE, and FAST can automatically verify some instances of the failure detector with fixed parameters, these tools cannot handle cases with unknown bounds Δ and Φ. To overcome this problem, we specify and prove correctness of the failure detector with the interactive theorem prover Ivy [29].

In the following, we first discuss the encoding of the failure detector, and then presents the experiments with Ivy.

The encoding of the message buffer with a counter requires that bound Δ is fixed. We here focus on cases where bound Δ is unknown. Hence, we encode the message buffer with a predicate in our Ivy specifications,

In Ivy, we declare `relation` existsMsgOfAge(X : `num`). Type `num` is interpreted as integers. Since Ivy does not support primed variables, we need an additional relation tmpExistsMsgOfAge(X : `num`). Intuitively, we first compute and store the value of existsMsgOfAge in the next state in tmpExistsMsgOfAge, then copy the value of tmpExistsMsgOfAge back to existsMsgOfAge. We do not consider the requirement of tmpExistsMsgOfAge as a shortcoming of Ivy since it is still straightforward to transform the ideas in Section 4 to Ivy.

Figure 2 represents how to add a fresh message in the message buffer in Ivy. Line 1 means that tmpExistsMsgOfAge is assigned an arbitrary value. Line 2 guarantees the appearance of a fresh message. Line 3 ensures that every in-transit message in existsMsgOfAge is preserved in tmpExistsMsgOfAge. Line 4 copies the value of tmpExistsMsgOfAge back to existsMsgOfAge.

Algorithm 2 Adding a fresh message in Ivy

1: tmpExistsMsgOfAge(X) := $*$;
2: **assume** tmpExistsMsgOfAge(0);
3: **assume forall** X : num . $0 < X \rightarrow$ existsMsgOfAge(X) = tmpExistsMsgOfAge(X);
4: existsMsgOfAge(X) := tmpExistsMsgOfAge(X);

Importantly, our specifications are not in decidable theories supported by Ivy. In Formula 11, the interpreted function " + " (addition) is applied to a universally quantified variable x.

The standard way to check whether a safety property *Prop* holds in an Ivy specification is to find an inductive invariant *IndInv* with *Prop*, and to (interactively) prove that *Indinv* holds in the specification. To verify the liveness properties Eventually Strong Accuracy, and Strong Completeness, we reduced them into safety properties by applying a reduction technique in Section 5, and found inductive invariants containing the resulted safety properties. These inductive invariants are the generalization of the inductive invariants in case of fixed parameters that were found in the previous experiments.

Table 4 shows the experiments on verification of the failure detector with Ivy in case of unknown Δ and Φ. The symbol $\star$ refers to that the initial value of timeout is arbitrary. The column "#line$_I$" shows the number of lines of an inductive invariant, and the column "#strengthening steps" shows the number of lines of strengthening steps that we provided for Ivy. The meaning of other columns is the same as in Table 1. While our specifications are not in the decidable theories supported in Ivy, our experiments show that Ivy needs no user-given strengthening steps to prove most of our inductive invariants. Hence, it took us about 4 weeks to learn Ivy from scratch, and to prove these inductive invariants.

Table 4: Proving inductive invariants with Ivy for arbitrary Δ and Φ.

#	Property	$\mathsf{timeout}_{\mathrm{init}}$	time	LOC	$\#\mathrm{line}_I$	#strengthening steps
1	Strong Accuracy	$= 6 \times \Phi + \Delta$	4s	183	30	0
2	Eventually Strong Accuracy	$= \star$	4s	186	35	0
3	Strong Completeness	$= 6 \times \Phi + \Delta$	8s	203	111	0
4		$\geq 6 \times \Phi + \Delta$	22s	207	124	15
5		$= \star$	44s	207	129	0

The most important thing to prove a property satisfied in an Ivy specification is to find an inductive invariant. Our inductive invariants use non-linear integers, quantifiers, and uninterpreted functions. (The inductive invariants in Table 4 are given in the full report [33].)

8 Conclusion

We have presented verification of both safety and liveness of the Chandra and Toeug failure detector by using the verification tools: model checkers for TLA$^+$ (TLC and APALACHE), counter automata (FAST), and the theorem prover Ivy. To do that, we first prove the cutoff results that can apply to the failure detector under partial synchrony. Next, we develop the encoding techniques to efficiently specify the failure detector, and to tune our models to the strength of the mentioned tools. We verified safety in case of fixed parameters by running the tools TLC, APALACHE, and FAST. To cope with cases of arbitrary bounds Δ and Φ, we reduced liveness properties to safety properties, and proved inductive invariants with desired properties in Ivy. While our specifications are not in the decidable theories supported in Ivy, our experiments show that Ivy needs no additional user assistance to prove most of our inductive invariants.

Modeling the failure detector in TLA$^+$ helps us understand and find inductive invariants in case of fixed parameters. Their structure is simpler but similar to the structure of parameterized inductive invariants. We found that the TLA$^+$ Toolbox [22] has convenient features, e.g., Profiler and Trace Exploration. A strong point of Ivy is in producing a counterexample quickly when a property is violated, even if all parameters are arbitrary. In contrast, FAST reports no counterexample in any case. Hence, debugging in FAST is very challenging.

While our specification describes executions of the Chandra and Toueg failure detector, we conjecture that many time constraints on network behaviors, correct processes, and failures in our inductive invariants can be reused to prove other algorithms under partial synchrony. We also conjecture that correctness of other partially synchronous algorithms may be proven by following the presented methodology. For future work, we would like to extend the above results for cases where GST is arbitrary. It is also interesting to investigate how to express discrete partial synchrony in timed automata [3], e.g., UPPAAL [26].

References

1. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing omega in systems with weak reliability and synchrony assumptions. Distributed Computing **21**(4), 285–314 (2008)
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with Byzantine failures and little system synchrony. In: DSN. pp. 147–155. IEEE (2006)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science **126**(2), 183–235 (1994)
4. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: FM. pp. 33–36. Springer (2012)
5. Atif, M., Mousavi, M.R., Osaiweran, A.: Formal verification of unreliable failure detectors in partially synchronous systems. In: SAC. pp. 478–485 (2012)
6. Bardin, S., Leroux, J., Point, G.: Fast extended release. In: CAV. pp. 63–66 (2006)
7. Bloem, R., Jacobs, S., Khalimov, A., Konnov, I., Rubin, S., Veith, H., Widder, J.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers (2015). https://doi.org/10.2200/S00658ED1V01Y201508DCT013, https://doi.org/10.2200/S00658ED1V01Y201508DCT013
8. Bravo, M., Chockler, G., Gotsman, A.: Making Byzantine consensus live. In: DISC. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
9. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. arXiv preprint arXiv:1807.04938 (2018)
10. Bunte, O., Groote, J.F., Keiren, J.J., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.: The mCRL2 toolset for analysing concurrent systems. In: TACAS. pp. 21–39. Springer (2019)
11. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43**(2), 225–267 (1996)
12. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ Proof System: Building a heterogeneous verification platform. In: ICTAC. pp. 44–44. Springer (2010)
13. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA$^+$ proofs. In: FM. pp. 147–154. Springer (2012)
14. Drăgoi, C., Widder, J., Zufferey, D.: Programming at the edge of synchrony. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–30 (2020)
15. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35**(2), 288–323 (1988)
16. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL. pp. 85–94 (1995)
17. Galois, I.: Ivy proofs of tendermint, URL: https://github.com/tendermint/spec/tree/master/ivy-proofs, accessed: December 2020
18. Konnov, I., Kukovec, J., Tran, T.H.: TLA$^+$ model checking made symbolic. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–30 (2019)
19. Konnov, I., Lazic, M., Veith, H., Widder, J.: Para^2: Parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. Formal Methods in System Design **51**(2), 270–307 (2017)
20. Konnov, I., Lazić, M., Veith, H., Widder, J.: Para^2: parameterized path reduction, acceleration, and SMT for reachability in threshold-guarded distributed algorithms. Formal Methods in System Design **51**(2), 270–307 (2017)
21. 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)

22. Kuppe, M.A., Lamport, L., Ricketts, D.: The TLA$^+$ toolbox. arXiv preprint arXiv:1912.10633 (2019)
23. Lamport, L.: Specifying systems: The TLA$^+$ language and tools for hardware and software engineers. Addison-Wesley (2002)
24. Lamport, L.: Using tlc to check inductive invariance (2018)
25. Larrea, M., Arévalo, S., Fernndez, A.: Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In: DISC. pp. 34–49. Springer (1999)
26. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1-2), 134–152 (1997)
27. Lime, D., Roux, O.H., Seidner, C., Traonouez, L.M.: Romeo: A parametric model-checker for Petri nets with stopwatches. In: TACAS. pp. 54–57. Springer (2009)
28. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. Laboratory for Computer Science, Massachusetts Institute of Technology (1988)
29. McMillan, K.L.: Ivy, URL: https://microsoft.github.io/ivy/, accessed: December 2020
30. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: CAV. pp. 190–202. Springer (2020)
31. Roscoe, A.W.: Understanding concurrent systems. Springer Science & Business Media (2010)
32. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying safety of synchronous fault-tolerant algorithms by bounded model checking. In: TACAS. pp. 357–374. Springer (2019)
33. Tran, T.H., Konnov, I., Widder, J.: FORTE2021-FD, URL: https://github.com/banhday/forte2021-fd, accessed: April 2021
34. Tran, T.H., Konnov, I., Widder, J.: Specifications of the Chandra and Toueg failure detector in TLA$^+$, FAST, and Ivy, URL: https://zenodo.org/record/4687714#.YHcBeBKxVH4, accessed: April 2021
35. Tran, T.H., Konnov, I., Widder, J.: Cutoffs for symmetric point-to-point distributed algorithms. In: NETYS. pp. 329–346. Springer (2020)
36. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In: PODC. pp. 347–356 (2019)
37. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Correct Hardware Design and Verification Methods, pp. 54–66. Springer (1999)