A Short Counterexample Property for Safety and Liveness Verification of Fault-Tolerant Distributed Algorithms

Igor Konnov Marijana Lazić Helmut Veith * Josef Widder

TU Wien, Austria

{konnov, lazic, veith, widder}@forsyte.at

Abstract

Distributed algorithms have many mission-critical applications ranging from embedded systems and replicated databases to cloud computing. Due to asynchronous communication, process faults, or network failures, these algorithms are difficult to design and verify. Many algorithms achieve fault tolerance by using threshold guards that, for instance, ensure that a process waits until it has received an acknowledgment from a majority of its peers. Consequently, domain-specific languages for fault-tolerant distributed systems offer language support for threshold guards.

We introduce an automated method for model checking of safety and liveness of threshold-guarded distributed algorithms in systems where the number of processes and the fraction of faulty processes are parameters. Our method is based on a *short counterexample property*: if a distributed algorithm violates a temporal specification (in a fragment of LTL), then there is a counterexample whose length is bounded and independent of the parameters. We prove this property by (i) characterizing executions depending on the structure of the temporal formula, and (ii) using commutativity of transitions to accelerate and shorten executions. We extended the ByMC toolset (Byzantine Model Checker) with our technique, and verified liveness and safety of 10 prominent fault-tolerant distributed algorithms, most of which were out of reach for existing techniques.

Categories and Subject Descriptors F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.4.5 [*Software*]: Operating systems: Fault-tolerance, Verification

Keywords Parameterized model checking, Byzantine faults, fault-tolerant distributed algorithms, reliable broadcast

* We dedicate this article to the memory of Helmut Veith, who passed away tragically after we finished the first draft together. In addition to contributing to this work, Helmut initiated our long-term research program on verification of fault-tolerant distributed algorithms, which made this paper possible.

Supported by: the Austrian Science Fund (FWF) through the National Research Network RiSE (S11403 and S11405), project PRAVDA (P27722), and Doctoral College LogiCS (W1255-N23); and by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

Copyright is held by the owner/author(s).

POPL'17, January 15–21, 2017, Paris, France
ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009860

1. Introduction

Distributed algorithms have many applications in avionic and automotive embedded systems, computer networks, and the internet of things. The central idea is to achieve dependability by replication, and to ensure that all correct replicas behave as one, even if some of the replicas fail. In this way, the correct operation of the system is more reliable than the correct operation of its parts. Fault-tolerant algorithms typically have been used in applications where highest reliability is required because human life is at risk (e.g., automotive or avionic industries), and even unlikely failures of the system are not acceptable. In contrast, in more mainstream applications like replicated databases, human intervention to restart the system from a checkpoint was often considered to be acceptable, so that expensive fault tolerance mechanisms were not used in conventional applications. However, new application domains such as cloud computing provide a new motivation to study fault-tolerant algorithms: with the huge number of computers involved, faults are the norm [53] rather than an exception, so that fault tolerance becomes an economic necessity; and so does the correctness of fault tolerance mechanisms. Hence, design, implementation, and verification of distributed systems constitutes an active research area [7, 23, 41, 42, 48, 57, 67]. Although distributed algorithms show complex behavior, and are difficult to understand for human engineers, there is only very limited tool support to catch logical errors in fault-tolerant distributed algorithms at design time.

The state of the art in the design of fault-tolerant systems is exemplified by the recent work on Paxos-like distributed algorithms like Raft [54] or M^{2}PAXOS [57]. The designers encode these algorithms in TLA+ [65], and use the TLC model checker to automatically find bugs in small instances, i.e., in distributed systems containing, e.g., three processes. Large distributed systems (e.g., clouds) need guarantees for *all* numbers of processes. These guarantees are typically given using hand-written mathematical proofs. In principle, these proofs could be encoded and machine-checked using the TLAPS proof system [16], PVS [49], Isabelle [15], Coq [48], Nuprl [60], or similar systems; but this requires human expertise in the proof checkers and in the application domain, and a lot of effort.

Ensuring correctness of the implementation is an open challenge: As the implementations are done by hand [54, 57], the connection between the specification and the implementation is informal, such that there is no formal argument about the correctness of the implementation. To address the discrepancy between design, implementation, and verification, Drăgoi et al. [23] introduced a domain-specific language PSync which is used for two purposes: (i) it compiles into running code, and (ii) it is used for verification. Their verification approach [24], requires a developer to provide invariants, and similar verification conditions. While this approach requires less human intervention than writing machine-checkable proofs, coming up with invariants of distributed systems requires considerable

```
case class EchoMsg extends Message

class ReliableBroadcastOnce
      extends DSLProtocol {
 val n = ALL.size          // nr. processes
 val t = ALL.size / 3 - 1 // max. faults
 var accept: Boolean = False

 UPON RECEIVING START WITH v DO {
  IF v == 1 THEN // check the initial value
  SEND EchoMsg TO ALL
 }
 UPON RECEIVING EchoMsg TIMES t + 1 DO {
  SEND EchoMsg TO ALL // >= 1 correct
 }
 UPON RECEIVING EchoMsg TIMES n - t DO {
  accept = True // almost all correct
 }
}
```

Figure 1. Code example of a distributed algorithm in DISTAL [7]. A distributed system consists of n processes, at most $t < n/3$ of which are Byzantine faulty. The correct ones execute the code, and no assumptions is made about the faulty processes.

human ingenuity. The Mace [41] framework is based on a similar idea, and is an extension to C++. While being fully automatic, their approach to correctness is light-weight in that it uses a tool that explores random walks to find (not necessarily all) bugs, rather than actually verifying systems.

In this paper we focus on automatic verification methods for programming constructs that are typical for fault-tolerant distributed algorithms. Figure 1 is an example of a distributed algorithm in the domain-specific language DISTAL [7]. It encodes the core of the reliable broadcast protocol from [64], which is used as building block of many fault-tolerant distributed systems. Line 13 and Line 16 use so-called "threshold guards" that check whether a given number of messages from distinct senders arrived at the receiver. As threshold guards are the central algorithmic idea for fault tolerance, domain-specific languages such as DISTAL or PSync have constructs for them (see [23] for an overview of domain-specific languages and formalization frameworks for distributed systems). For instance, the code in Figure 1 works for systems with n processes among which t can fail, with $t < n/3$ as required for Byzantine fault tolerance [56]. In such systems, waiting for messages from $n - t$ processes ensures that if all correct processes send messages, then faulty processes cannot prevent progress. Similarly, waiting for $t + 1$ messages ensures that at least one message was sent by a correct process. Konnov et al. [42] introduced an automatic method to verify safety of algorithms with threshold guards. Their method is parameterized in that it verifies distributed algorithms for all values of parameters (n and t) that satisfy a resilience condition ($t < n/3$). This work bares similarities to the classic work on reduction for parallel programs by Lipton [50]. Lipton proves statements like "all P operations on a semaphore are left movers with respect to operations on other processes." He proves that given a run that ends in a given state, the same state is reached by the run in which the P operation has been moved. Konnov et al. [42] do a similar analysis for threshold-guarded operations, in which they analyze the relation between statements from Figure 1 like "`send EchoMsg`" and "`UPON RECEIVING EchoMsg TIMES t + 1`" in order to determine which statements are movable. From this, they develop an offline partial order reduction that together with acceleration [6, 44] reduced reachability checking to complete bounded model checking using SMT. In this way, they automatically check safety of fault-tolerant algorithms.

However, for fault-tolerant distributed algorithms liveness is as important as safety: This comes from the celebrated impossibility result by Fischer, Lynch, and Paterson [32] that states that a fault-tolerant consensus algorithm cannot ensure both safety and liveness in asynchronous systems. It is folklore that designing a safe fault-tolerant distributed algorithm is trivial: *just do nothing*; e.g., by never committing transactions, one cannot commit them in inconsistent order. Hence, a technique that verifies only safety may establish the "correctness" of a distributed algorithm that never does anything useful. To achieve trust in correctness of a distributed algorithm, we need tools that verify both safety and liveness.

As exemplified by [31], liveness verification of parameterized distributed and concurrent systems is still a research challenge. Classic work on parameterized model checking by German and Sistla [35] has several restrictions on the specifications ($\forall i.\ \phi(i)$) and the computational model (rendezvous), which are incompatible with fault-tolerant distributed algorithms. In fact, none of the approaches (e.g., [18, 26, 27, 59]) surveyed in [9] apply to the algorithms we consider. More generally, in the parameterized case, going from safety to liveness is not straightforward. There are systems where safety is decidable and liveness is not [28].

Contributions. We generalize the approach by Konnov et al. [42, 44] to liveness by presenting a framework and a model checking tool that takes as input a description of a distributed algorithm (in our variant [36] of Promela [39]) and specifications in a fragment of linear temporal logic. It then shows correctness for all parameter values (e.g., n and t) that satisfy the required resilience condition (e.g., $t < n/3$), or reports a counterexample:

1. As in the classic result by Vardi and Wolper [66], we observe that it is sufficient to search for counterexamples that have the form of a lasso, i.e., after a finite prefix an infinite loop is entered. Based on this, we analyze specifications automatically, in order to enumerate possible shapes of lassos depending on temporal operators **F** and **G** and evaluations of threshold guards.
2. We automatically do offline partial order reduction using the algorithm's description. For this, we introduce a more refined mover analysis for threshold guards and temporal properties. We extend Lipton's reduction method [50] (re-used and extended by many others [19, 22, 25, 34, 44, 47]), so that we maintain invariants, which allows us to go beyond reachability and verify specifications with the temporal operators **F** and **G**.
3. By combining acceleration [6, 44] with Points 1 and 2, we obtain a short counterexample property, that is, that infinite executions (which may potentially be counterexamples) have "equivalent" representatives of bounded length. The bound depends on the process code and is independent of the parameters. The equivalence is understood in terms of temporal logic specifications that are satisfied by the original executions and the representatives, respectively. We show that the length of the representatives increases mildly compared to reachability checking in [42]. This implies a so-called completeness threshold [46] for threshold-based algorithms and our fragment of LTL.
4. Consequently, we only have to check a reasonable number of SMT queries that encode parameterized and bounded-length representatives of executions. We show that if the parameterized system violates a temporal property, then SMT reports a counterexample for one of the queries. We prove that otherwise the specification holds for all system sizes.
5. Our theoretical results and our implementation push the boundary of liveness verification for fault-tolerant distributed algorithms. While prior results [40] scale just to two out of ten benchmarks from [42], we verified safety and liveness of all ten. These benchmarks originate from distributed algorithms [11, 12, 14, 21, 37, 52, 61, 63, 64] that constitute the core of important services such as replicated state machines.

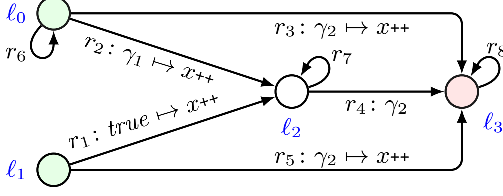

Figure 2. The threshold automaton corresponding to Figure 1 with $\gamma_1 : x \geq (t+1) - f$ and $\gamma_2 : x \geq (n-t) - f$ over parameters n, t, and f, representing the number of processes, the upper bound on the faulty processes (used in the code), and the actual number of faulty processes. The negative number $-f$ in the threshold is used to model the environment, and captures that at most f of the received messages may have been sent by faulty processes.

From a theoretical viewpoint, we introduce new concepts and conduct extensive proofs (the proofs can be found in [43]) for Points 1 and 2. From a practical viewpoint, we have built a complete framework for model checking of fault-tolerant distributed algorithms that use threshold guards, which constitute the central programming paradigm for dependable distributed systems.

2. Representation of Distributed Algorithms

2.1 Threshold Automata

As internal representation in our tool, and in the theoretical work of this paper, we use *threshold automata* (TA) defined in [44]. The TA that corresponds to the DISTAL code from Figure 1 is given in Figure 2. The threshold automaton represents the local control flow of a single process, where arrows represent local transitions that are labeled with $\varphi \mapsto \texttt{act}$: Expression φ is a threshold guard and the action act may increment a shared variable.

Example 2.1. The TA from Figure 2 is quite similar to the code in Figure 1: if `START` is called with $v = 1$ this corresponds to the initial local state ℓ_1, while otherwise a process starts in ℓ_0. Initially a process has not sent any messages. The local state ℓ_2 in Figure 2 captures that the process has sent `EchoMsg` and `accept` evaluates to false, while ℓ_3 captures that the process has sent `EchoMsg` and `accept` evaluates to true. The syntax of Figure 1, although checking how many messages of some type are received, hides bookkeeping details and the environment, e.g., message buffers. For our verification technique, we need to make such issues explicit: The shared variable x stores the number of correct processes that have sent `EchoMsg`. Incrementing x models that `EchoMsg` is sent when the transition is taken. Then, execution of Line 9 corresponds to the transition r_1. Executing Line 13 is captured by r_2: the check whether $t+1$ messages are received is captured by the fact that r_2 has the guard γ_1, that is, $x \geq (t+1) - f$. Intuitively, this guard checks whether sufficiently many processes have sent `EchoMsg` (i.e., increased x), and takes into account that at most f messages may have been sent by faulty processes. Namely, if we observe the guard in the equivalent form $x + f \geq t+1$, then we notice that it evaluates to true when the total number of received `EchoMsg` messages from correct processes (x) and potentially received messages from faulty processes (at most f), is at least $t+1$, which corresponds to the guard of Line 13. Transition r_4 corresponds to Line 16, r_3 captures that Line 9 and Line 16 are performed in one protocol step, and r_5 captures Line 13 and Line 16. ◁

While the example shows that the code in a domain-specific language and a TA are quite close, it should be noted that in reality, things are slightly more involved. For instance, the DISTAL runtime takes care of the bookkeeping of sent and received messages (waiting queues at different network layers, buffers, etc.), and just triggers the high-level protocol when a threshold guard evaluates to true. This typically requires counting the number of received messages. While these local counters are present in the implementation, they are abstracted in the TA. For the purpose of this paper we do not need to get into the details. Discussions on data abstraction and automated generation of TAs from code similar to DISTAL can be found in [45].

We recall the necessary definitions introduced in [44]. A threshold automaton is a tuple $\mathsf{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, RC)$ whose components are defined as follows: The *local states* and the *initial states* are in the finite sets $\mathcal{L}$ and $\mathcal{I} \subseteq \mathcal{L}$, respectively. For simplicity, we identify local states with natural numbers, i.e., $\mathcal{L} = \{1, \ldots, |\mathcal{L}|\}$. *Shared variables* and *parameter variables* range over $\mathbb{N}_0$ and are in the finte sets Γ and Π, respectively. The *resilience condition RC* is a formula over parameter variables in linear integer arithmetic, and the *admissible parameters* are $\mathbf{P}_{RC} = \{\mathbf{p} \in \mathbb{N}_0^{|\Pi|} : \mathbf{p} \models RC\}$. After an example for resilience conditions, we will conclude the definition of a threshold automaton by defining $\mathcal{R}$ as the finite set of rules.

Example 2.2. The admissible parameters and resilience conditions are motivated by fault-tolerant distributed algorithms: Let n be the number of processes, t be the assumed number of faulty processes, and in a run, f be the actual number of faults. For these parameters, the famous result by Pease, Shostak and Lamport [56] states that agreement can be solved iff the resilience condition $n > 3t \wedge t \geq f \geq 0$ is satisfied. Given such constraints, the set $\mathbf{P}_{RC}$ is infinite, and in Section 2.2 we will see that this results in an infinite state system. ◁

A *rule* is a tuple $(\mathit{from}, \mathit{to}, \varphi^{\leq}, \varphi^{>}, \mathbf{u})$, where *from* and *to* are from $\mathcal{L}$, and capture from which local state to which a process moves via that rule. A rule can only be executed if $\varphi^{\leq}$ and $\varphi^{>}$ are true; both are conjunction of guards. Each guard consists of a shared variable $x \in \Gamma$, coefficients $a_0, \ldots, a_{|\Pi|} \in \mathbb{Z}$, and parameter variables $p_1, \ldots, p_{|\Pi|} \in \Pi$ so that $x \geq a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i$ is a *lower guard* and $x < a_0 + \sum_{i=1}^{|\Pi|} a_i \cdot p_i$ is an *upper guard*. Then, Φ^{rise} and Φ^{fall} are the sets of lower and upper guards.[1] Rules may increase shared variables using an update vector $\mathbf{u} \in \mathbb{N}_0^{|\Gamma|}$ that is added to the vector of shared variables. Finally, $\mathcal{R}$ is the finite set of rules.

Example 2.3. A rule corresponds to an edge in Figure 2. The pair $(\mathit{from}, \mathit{to})$ encodes the edge while $(\varphi^{\leq}, \varphi^{>}, \mathbf{u})$ encodes the edge label. For example, rule r_2 would be $(\ell_0, \ell_2, \gamma_1, \top, 1)$. Thus, a rule corresponds to a (guarded) statement from Figure 1 (or combined statements as discussed in Example 2.1). ◁

The above definition of TAs is quite general. It allows loops, increase of shared variables in loops, etc. As has been observed in [44], if one does not restrict increases on shared variables, the resulting systems may produce runs that visit infinitely many states, and there is little hope for a complete verification method. Hence, Konnov et al. [42] analyzed the TAs of the benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64]: They observed that some states have self-loops (corresponding to busy-waiting for messages to arrive) and in the case of failure detector based algorithms [61] there are loops that consist of at most two rules. None of the rules in loops increase shared variables. In our theory, we allow more general TAs than actually found in the benchmarks. In more detail, we make the following assumption:

[1] Compared to [42], we use the more intuitive notation of Φ^{rise} and Φ^{fall}: lower guards can only change from false to true (rising), while upper guards can only change from true to false (falling); cf. Proposition 5.1.

Threshold automata for fault-tolerant distributed algorithms. As in [44], we assume that if a rule r is in a loop, then $r.\mathbf{u} = \mathbf{0}$. In addition, we use the restriction that all the cycles of a TA are simple, i.e., between any two locations in a cycle there exists exactly one node-disjoint directed path (nodes in cycles may have self-loops). We conjecture that this restriction can be relaxed as in [42], but this is orthogonal to our work.

Example 2.4. In the TA from Figure 2 we use the shared variable x as the number of correct processes that have sent a message. One easily observes that the rules that update x do not belong to loops. Indeed, all the benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64] share this structure. This is because at the algorithmic level, all these algorithms are based on the reliable communication assumption (no message loss and no spurious message generation/duplication), and not much is gained by resending the same message. In these algorithms a process checks whether sufficiently many processes (e.g., a majority) have sent a message to signal that they are in some specific local state. Consequently, a receiver would ignore duplicate messages from the same sender. In our analysis we exploit this characteristic of distributed algorithms with threshold guards, and make the corresponding assumption that processes do not send (i.e., increase x) from within a loop. Similarly, as a process cannot make the sending of a message undone, we assume that shared variables are never decreased. So, while we need these assumptions to derive our results, they are justified by our application domain. $\triangleleft$

2.2 Counter Systems

A threshold automaton models a single process. Now the question arises how we define the composition of multiple processes that will result in a distributed system. Classically, this is done by parallel composition and interleaving semantics: A state of a distributed system that consists of n processes is modeled as n-dimensional vector of local states. The transition to a successor state is then defined by non-deterministically picking a process, say i, and changing the ith component of the n-dimensional vector according to the local transition relation of the process. However, for our domain of threshold-guarded algorithms, we do not care about the precise n-dimensional vector so that we use a more efficient encoding: It is well-known that the system state of specific distributed or concurrent systems can be represented as a counter system [2, 44, 51, 59]: instead of recording for some local state ℓ, which processes are in ℓ, we are only interested in *how many processes are in* ℓ. In this way, we can efficiently encode transition systems in SMT with linear integer arithmetics. Therefore, we formalize the semantics of the threshold automata by counter systems.

Fix a threshold automaton TA, a function (expressible as linear combination of parameters) $N\colon \mathbf{P}_{RC} \to \mathbb{N}_0$ that determines the number of modeled processes, and admissible parameter values $\mathbf{p} \in \mathbf{P}_{RC}$. A counter system $\mathsf{Sys}(\mathsf{TA})$ is defined as a transition system (Σ, I, R), with configurations Σ and I and transition relation R defined below.

Definition 2.5. *A configuration $\sigma = (\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$ consists of a vector of* counter values $\sigma.\boldsymbol{\kappa} \in \mathbb{N}_0^{|\mathcal{L}|}$*, a vector of* shared variable values $\sigma.\mathbf{g} \in \mathbb{N}_0^{|\Gamma|}$*, and a vector of* parameter values $\sigma.\mathbf{p} = \mathbf{p}$*. The set Σ contains all configurations. The initial configurations are in set I, and each initial configuration σ satisfies $\sigma.\mathbf{g} = \mathbf{0}$, $\sum_{i\in\mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = N(\mathbf{p})$, and $\sum_{i\notin\mathcal{I}} \sigma.\boldsymbol{\kappa}[i] = 0$.*

Example 2.6. The safety property from Example 2.2, refers to an initial configuration that satisfies resilience condition $n > 3t \wedge t \geq f \geq 0$, e.g., $4 > 3 \cdot 1 \wedge 1 \geq 0 \geq 0$ such that $\sigma.\mathbf{p} = (4, 1, 0)$. In our encodings we typically have N is the function $(n, t, f) \mapsto n - f$. Further, $\sigma.\boldsymbol{\kappa}[\ell_0] = N(\mathbf{p}) = n - f = 4$ and $\sigma.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$, and the shared variable $\sigma.\mathbf{g} = 0$. $\triangleleft$

A *transition* is a pair $t = (\mathit{rule}, \mathit{factor})$ of a rule and a non-negative integer called the *acceleration factor*. For $t = (\mathit{rule}, \mathit{factor})$ we write $t.\mathbf{u}$ for $\mathit{rule}.\mathbf{u}$, etc. A transition t is *unlocked* in σ if $\forall k \in \{0, \ldots, t.\mathit{factor} - 1\}.\ (\sigma.\boldsymbol{\kappa}, \sigma.\mathbf{g} + k \cdot t.\mathbf{u}, \sigma.\mathbf{p}) \models t.\varphi^{\leq} \wedge t.\varphi^{>}$. A transition t is *applicable (or enabled)* in σ, if it is unlocked, and $\sigma.\boldsymbol{\kappa}[t.\mathit{from}] \geq t.\mathit{factor}$, or $t.\mathit{factor} = 0$.

Example 2.7. This notion of applicability contains acceleration and is central for our approach. Intuitively, the value of the factor corresponds to how many times the rule is executed by different processes. In this way, we can subsume steps by an arbitrary number of processes into one transition. Consider Figure 2. If for some k, k processes are in location ℓ_1, then in classic modeling it takes k transitions to move these processes one-by-one to ℓ_2. With acceleration, however, these k processes can be moved to ℓ_2 in one step, independently of k. In this way, the bounds we compute will be independent of the parameter values. However, assuming x to be a shared variable and f being a parameter that captures the number of faults, our (crash-tolerant) benchmarks include rules like "$x < f \mapsto x{+}{+}$" for local transition to a special "crashed" state. The above definition ensures that at most $f - x$ of these transitions are accelerated into one transition (whose factor thus is at most $f - x$). This precise treatment of threshold guards is crucial for fault-tolerant distributed algorithms. The central contribution of this paper is to show how acceleration can be used to shorten schedules while maintaining specific temporal logic properties. $\triangleleft$

Definition 2.8. *The configuration σ' is the result of applying the enabled transition t to σ, if*

1. $\sigma'.\mathbf{g} = \sigma.\mathbf{g} + t.\mathit{factor} \cdot t.\mathbf{u}$
2. $\sigma'.\mathbf{p} = \sigma.\mathbf{p}$
3. *if* $t.\mathit{from} \neq t.\mathit{to}$ *then* $\sigma'.\boldsymbol{\kappa}[t.\mathit{from}] = \sigma.\boldsymbol{\kappa}[t.\mathit{from}] - t.\mathit{factor}$, $\sigma'.\boldsymbol{\kappa}[t.\mathit{to}] = \sigma.\boldsymbol{\kappa}[t.\mathit{to}] + t.\mathit{factor}$, *and* $\forall \ell \in \mathcal{L} \setminus \{t.\mathit{from}, t.\mathit{to}\}.\ \sigma'.\boldsymbol{\kappa}[\ell] = \sigma.\boldsymbol{\kappa}[\ell]$.
4. *if* $t.\mathit{from} = t.\mathit{to}$ *then* $\sigma'.\boldsymbol{\kappa} = \sigma.\boldsymbol{\kappa}$.

In this case we use the notation $\sigma' = t(\sigma)$.

Example 2.9. Let us again consider Figure 2 with $n = 4$, $t = 1$, and $f = 1$. We consider the initial configuration where $\sigma.\boldsymbol{\kappa}[\ell_1] = n - f = 3$ and $\sigma.\boldsymbol{\kappa}[\ell_i] = 0$, for $\ell_i \in \mathcal{L} \setminus \{\ell_0\}$. The guard of rule r_5, $\gamma_2\colon x \geq (n - t) - f = 2$, initially evaluates to false because $x = 0$. The guard of rule r_1 is true, so that any transition (r_1, factor) is unlocked. As $\sigma.\boldsymbol{\kappa}[\ell_1] = 3$, all transitions (r_1, factor), for $0 \leq \mathit{factor} \leq 3$ are applicable. If the transition $(r_1, 2)$ is applied to the initial configuration, we obtain that $x = 2$ so that, after the application, γ_2 evaluates to true. Then r_5 is unlocked and the transitions $(r_5, 1)$ and $(r_5, 0)$ are applicable as $\sigma.\boldsymbol{\kappa}[\ell_1] = 1$. Since γ_2 checks for greater or equal, once it becomes true it remains true. Such monotonic behavior is given for all guards, as has already been observed in [44, Proposition 7], and is a crucial property. $\triangleleft$

The transition relation R is defined as follows: Transition (σ, σ') belongs to R iff there is a rule $r \in \mathcal{R}$ and a factor $k \in \mathbb{N}_0$ such that $\sigma' = t(\sigma)$ for $t = (r, k)$. A *schedule* is a sequence of transitions. For a schedule τ and an index $i : 1 \leq i \leq |\tau|$, by $\tau[i]$ we denote the ith transition of τ, and by τ^i we denote the prefix $\tau[1], \ldots, \tau[i]$ of τ. A schedule $\tau = t_1, \ldots, t_m$ is *applicable* to configuration σ_0, if there is a sequence of configurations $\sigma_1, \ldots, \sigma_m$ with $\sigma_i = t_i(\sigma_{i-1})$ for $1 \leq i \leq m$. A schedule $t_1, \ldots, t_m$ where $t_i.\mathit{factor} = 1$ for $0 < i \leq m$ is called *conventional*. If there is a $t_i.\mathit{factor} > 1$, then a schedule is *accelerated*. By $\tau \cdot \tau'$ we denote the concatenation of two schedules τ and τ'.

We will reason about schedules in Section 6 for our mover analysis, which is naturally expressed by swapping neighboring transitions in a schedule. To reason about temporal logic properties, we need to reason about the configurations that are "visited" by a schedule. For that we now introduce paths.

A finite or infinite sequence $\sigma_0, t_1, \sigma_1, \ldots, t_{k-1}, \sigma_{k-1}, t_k, \ldots$ of alternating configurations and transitions is called a *path*, if for every transition t_i, $i \in \mathbb{N}$, in the sequence, holds that t_i is enabled in σ_{i-1}, and $\sigma_i = t_i(\sigma_{i-1})$. For a configuration σ_0 and a finite schedule τ applicable to σ_0, by $\mathsf{path}(\sigma_0, \tau)$ we denote $\sigma_0, t_1, \sigma_1, \ldots, t_{|\tau|}, \sigma_{|\tau|}$ with $\sigma_i = t_i(\sigma_{i-1})$, for $1 \le i \le |\tau|$. Similarly, if τ is an infinite schedule applicable to σ_0, then $\mathsf{path}(\sigma_0, \tau)$ represents an infinite sequence $\sigma_0, t_1, \sigma_1, \ldots, t_{k-1}, \sigma_{k-1}, t_k, \ldots$ where $\sigma_i = t_i(\sigma_{i-1})$, for all $i > 0$.

The evaluation of the threshold guards solely defines whether certain rules are unlocked. As was discussed in Example 2.9, along a path, the evaluations of guards are monotonic. The set of upper guards that evaluate to false and lower guards that evaluate to true — called the context — changes only finitely many times. A schedule can thus be understood as an alternating sequence of schedules without context change, and context-changing transitions. We will recall the definitions of context etc. from [42] in Section 5. We say that a schedule τ is *steady* for a configuration σ, if every configuration of $\mathsf{path}(\sigma, \tau)$ has the same context.

Due to the resilience conditions and admissible parameters, our counter systems are in general infinite state. The following proposition establishes an important property for verification.

Proposition 2.10. *Every (finite or infinite) path visits finitely many configurations.*

Proof. By Definition 2.8(3), if a transition t is applied to a configuration σ, then the sum of the counters remains unchanged, that is, $\sum_{\ell \in \mathcal{L}} \sigma.\boldsymbol{\kappa}[\ell] = \sum_{\ell \in \mathcal{L}} t(\sigma).\boldsymbol{\kappa}[\ell]$. By repeating this argument, the sum of the counters remains stable in a path. By Definition 2.8(2) the parameter values also remain stable in a path.

By Definition 2.8(1), it remains to show that in each path eventually the shared variable $\mathbf{g}$ stop increasing. Let us fix a rule $r = (\mathit{from}, \mathit{to}, \varphi^{\le}, \varphi^{>}, \mathbf{u})$ that increases $\mathbf{g}$. By the definition of a transition, applying some transition (r, factor) decreases $\boldsymbol{\kappa}[r.\mathit{from}]$ by factor. As by assumption on TAs, r is not in a cycle, $\boldsymbol{\kappa}[r.\mathit{from}]$ is increased only finitely often, namely, at most $N(\mathbf{p})$ times. As there are only finitely many rules in a TA, the proposition follows. □

3. Verification Problems: Parameterized Reachability vs. Safety & Liveness.

In this section we will discuss the verification problems for fault-tolerant distributed algorithms. A central challenge is to handle resilience conditions precisely.

Example 3.1. The safety property (unforgeability) of [64] expressed in terms of Figure 2 means that no process should ever enter ℓ_3 if initially all processes are in ℓ_0, given that $n > 3t \wedge t \ge f \ge 0$. We can express this in the counter system: under the resilience condition $n > 3t \wedge t \ge f \ge 0$, given an initial configuration σ, with $\sigma.\boldsymbol{\kappa}[\ell_0] = n - f$, to verify safety, we have to establish the absence of a schedule τ that satisfies $\sigma' = \tau(\sigma)$ and $\sigma'.\boldsymbol{\kappa}[\ell_3] > 0$.

In order to be able to answer this question, we have to deal with these resilience conditions precisely: Observe that ℓ_3 is unreachable, as all outgoing transitions from ℓ_0 contain guards that evaluate to false initially, and since all processes are in ℓ_0 no process ever increases x. A slight modification of $t \ge f$ to $t+1 \ge f$ in the resilience condition changes the result, i.e., one fault too many breaks the system. For example, if $n = 4$, $t = 1$, and $f = 2$, then the new resilience condition holds, but as the guard $\gamma_1 : x \ge (t+1) - f$ is now initially true, then one correct process can fire the rule r_2 and increase x. Now when $x = 1$, the guard $\gamma_2 : x \ge (n-t) - f$ becomes true, so that the process can fire the rule r_4 and reach the state ℓ_3. This tells us that unforgeability is not satisfied in the system where the resilience condition is $n > 3t \wedge t + 1 \ge f \ge 0$. ◁

$$\begin{aligned}
\psi &::= \mathit{pform} \mid \mathsf{G}\,\psi \mid \mathsf{F}\,\psi \mid \psi \wedge \psi \\
\mathit{pform} &::= \mathit{cform} \mid \mathit{gform} \vee \mathit{cform} \\
\mathit{cform} &::= \bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0 \mid \bigwedge_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] = 0 \mid \mathit{cform} \wedge \mathit{cform} \\
\mathit{gform} &::= \mathit{guard} \mid \neg \mathit{gform} \mid \mathit{gform} \wedge \mathit{gform}
\end{aligned}$$

Table 1. The syntax of $\mathsf{ELTL_{FT}}$-formulas: *pform* defines propositional formulas, and ψ defines temporal formulas. We assume that $\mathit{Locs} \subseteq \mathcal{L}$ and $\mathit{guard} \in \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$.

This is the verification question studied in [42], which can be formalized as follows:

Definition 3.2 (Parameterized reachability)**.** *Given a threshold automaton TA and a Boolean formula B over* $\{\boldsymbol{\kappa}[i] = 0 \mid i \in \mathcal{L}\}$, *check whether there are parameter values* $\mathbf{p} \in \mathbf{P}_{RC}$, *an initial configuration* $\sigma_0 \in I$ *with* $\sigma_0.\mathbf{p} = \mathbf{p}$ *and a finite schedule* τ *applicable to* σ_0 *such that* $\tau(\sigma_0) \models B$.

As shown in [42], if such a schedule exists, then there is also a schedule of bounded length. In this paper, we do not limit ourselves to reachability, but consider specifications of *counterexamples to safety and liveness* of FTDAs from the literature. We observe that such specifications use a simple subset of linear temporal logic that contains only the temporal operators F and G.

Example 3.3. Consider a liveness property from the distributed algorithms literature called correctness [64]:

$$\mathsf{G}\,\mathsf{F}\,\psi_{\mathrm{fair}} \rightarrow (\boldsymbol{\kappa}[\ell_0] = 0 \rightarrow \mathsf{F}\,\boldsymbol{\kappa}[\ell_3] \neq 0). \quad (1)$$

Formula ψ_{fair} expresses the reliable communication assumption of distributed algorithms [32]. In this example, $\psi_{\mathrm{fair}} \equiv \boldsymbol{\kappa}[\ell_1] = 0 \wedge (x \ge t+1 \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_1] = 0) \wedge (x \ge n-t \rightarrow \boldsymbol{\kappa}[\ell_0] = 0 \wedge \boldsymbol{\kappa}[\ell_2] = 0)$. Intuitively, $\mathsf{G}\,\mathsf{F}\,\psi_{\mathrm{fair}}$ means that all processes in ℓ_1 should eventually leave this state, and if sufficiently many messages of type x are sent (γ_1 or γ_2 holds true), then all processes eventually receive them. If they do so, they have to eventually fire rules r_1, r_2, r_3, or r_4 and thus leave locations ℓ_0, ℓ_1, and ℓ_2. Our approach is based on possible shapes of *counterexamples*. Therefore, we consider the negation of the specification (1), that is, $\mathsf{G}\,\mathsf{F}\,\psi_{\mathrm{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathsf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. In the following we define the logic that can express such counterexamples. ◁

The fragment of LTL limited to F and G was studied in [29, 46]. We further restrict it to the logic that we call *Fault-Tolerant Temporal Logic* ($\mathsf{ELTL_{FT}}$), whose syntax is shown in Table 1. The formulas derived from *cform* — called counter formulas — restrict counters, while the formulas derived from *gform* — called guard formulas — restrict shared variables. The formulas derived from *pform* are propositional formulas. The temporal operators F and G follow the standard semantics [5, 17], that is, for a configuration σ and an infinite schedule τ, it holds that $\mathsf{path}(\sigma, \tau) \models \varphi$, if:

1. $\sigma \models \varphi$, when φ is a propositional formula,
2. $\exists \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathsf{F}\,\psi$,
3. $\forall \tau', \tau'' : \tau = \tau' \cdot \tau''.\ \mathsf{path}(\tau'(\sigma), \tau'') \models \psi$, when $\varphi = \mathsf{G}\,\psi$.

To stress that the formula should be satisfied by *at least one path*, we prepend $\mathsf{ELTL_{FT}}$-formulas with the existential path quantifier E. We use the shorthand notation *true* for a valid propositional formula, e.g., $\bigwedge_{i \in \emptyset} \boldsymbol{\kappa}[i] = 0$. We also denote with $\mathsf{ELTL_{FT}}$ the set of all formulas that can be written using the logic $\mathsf{ELTL_{FT}}$.

We will reason about invariants of the finite subschedules, and consider a propositional formula ψ. Given a configuration σ, a finite schedule τ applicable to σ, and ψ, by $\mathsf{Cfgs}(\sigma, \tau) \models \psi$ we denote

```
algorithm parameterized_model_checking(TA, φ): // see Def. 3.4
 G := cut_graph (φ) /* Sect. 4 */
 H := threshold_graph(TA) /* Sect. 5 */
 for each ≺ in topological_orderings(G ∪ H) do // e.g., using [13]
  check_one_order(TA, φ, G, H, ≺) /* Sect. 6–7 */
  if SMT_sat() then report the SMT model as a counterexample
```

Figure 3. A high-level description of the verification algorithm. For details of `check_one_order`, see Section 7.2 and Figure 10.

that ψ holds in every configuration σ' visited by the path $\mathsf{path}(\sigma, \tau)$. In other words, for every prefix τ' of τ, we have that $\tau'(\sigma) \models \psi$.

Definition 3.4 (Parameterized unsafety & non-liveness)**.** *Given a threshold automaton TA and an $\mathsf{ELTL_{FT}}$ formula ψ, check whether there are parameter values $\mathbf{p} \in \mathbf{P}_{RC}$, an initial configuration $\sigma_0 \in I$ with $\sigma_0.\mathbf{p} = \mathbf{p}$, and an infinite schedule τ of Sys(TA) applicable to σ_0 such that $\mathsf{path}(\sigma_0, \tau) \models \psi$.*

Complete bounded model checking. We solve this problem by showing how to reduce it to bounded model checking while guaranteeing completeness. To this end, we have to construct a bounded-length encoding of infinite schedules. In more detail:

- We observe that if $\mathsf{path}(\sigma_0, \tau) \models \psi$, then there is an initial state σ and two finite schedules ϑ and ρ (of unknown length) that can be used to construct an infinite (lasso-shaped) schedule $\vartheta \cdot \rho^\omega$, such that $\mathsf{path}(\sigma, \vartheta \cdot \rho^\omega) \models \psi$ (Section 4.1).
- Now given ϑ and ρ, we prove that we can use a ψ-specific reduction, to cut ϑ and ρ into subschedules $\vartheta_1, \dots, \vartheta_m$ and $\rho_1, \dots, \rho_n$, respectively so that the subschedules satisfy subformulas of ψ (Sections 4.2, 4.3 and 5).
- We use an offline partial order reduction, specific to the subformulas of ψ, and acceleration to construct representative schedules $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ that satisfy the required $\mathsf{ELTL_{FT}}$ formulas that are satisfied ϑ_i and ρ_j, respectively for $1 \le i \le m$ and $1 \le j \le n$. Moreover, $\mathsf{rep}[\vartheta_i]$ and $\mathsf{rep}[\rho_j]$ are fixed sequences of rules, where bounds on the lengths of the sequences are known (Section 6).
- These fixed sequence of rules can be used to encode a query to the SMT solver (Section 7.1). We ask whether there is an applicable schedule in the counter system that satisfies the sequence of rules and ψ (Section 7.3). If the SMT solver reports a contradiction, there exists no counterexample.

Based on these theoretical results, our tool implements the high-level verification algorithm from Figure 3 (in the comments we give the sections that are concerned with the respective step):

4. Shapes of Schedules that Satisfy $\mathsf{ELTL_{FT}}$

We characterize all possible shapes of lasso schedules that satisfy an $\mathsf{ELTL_{FT}}$-formula φ. These shapes are characterized by so-called *cut points*: We show that every lasso satisfying φ has a fixed number of cut points, one cut point per a subformula of φ that starts with $\mathbf{F}$. The configuration in the cut point of a subformula $\mathbf{F}\,\psi$ must satisfy ψ, and all configurations between two cut points must satisfy certain propositional formulas, which are extracted from the subformulas of φ that start with $\mathbf{G}$. Our notion of a cut point is motivated by extreme appearances of temporal operators [29].

Example 4.1. Consider the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathbf{E}\,\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, where $a, \dots, e$ are propositional formulas, whose structure is not of interest in this section. Formula φ is satisfiable by certain paths that have lasso shapes, i.e., a path consists of a finite prefix and a loop, which is repeated infinitely. These lassos may differ in the actual occurrences of the propositions and the start of the loop: For instance, at some point, a holds, and since then b always holds, then d holds at some point, then e holds at some point, then the loop is entered, and c holds infinitely often inside the loop. This is the case (a) shown in Figure 4, where the configurations in the cut points A, B, C, and D must satisfy the propositional formulas a, d, e, and c respectively, and the configurations between A and F must satisfy the propositional formula b. This example does not restrict the propositions between the initial state and the cut point A, so that this lasso shape, for instance, also captures the path where b holds from the beginning. There are 20 different lasso shapes for φ, five of them are shown in the figure. We construct lasso shapes that are sufficient for finding a path satisfying an $\mathsf{ELTL_{FT}}$ formula. In this example, it is sufficient to consider lasso shapes (a) and (b), since the other shapes can be constructed from (a) and (b) by unrolling the loop several times. ◁

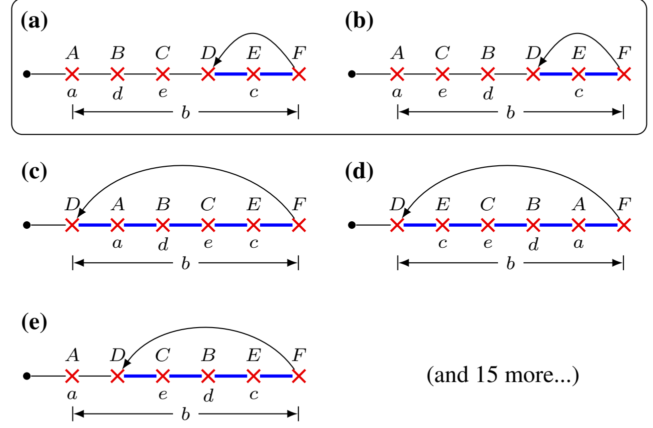

Figure 4. The shapes of lassos that satisfy the formula $\mathbf{E}\,\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$. The crosses show cut points for: (A) formula $\mathbf{F}\,(a \wedge \mathbf{F}\,d \wedge \mathbf{F}\,e \wedge \mathbf{G}\,b \wedge \mathbf{G}\,\mathbf{F}\,c)$, (B) formula $\mathbf{F}\,d$, (C) formula $\mathbf{F}\,e$, (D) loop start, (E) formula $\mathbf{F}\,c$, and (F) loop end.

4.1 Restricting Schedules to Lassos

In the seminal paper [66], Vardi and Wolper showed that if a finite-state transition system M *violates* an LTL formula — which requires *all paths* to satisfy the formula — then there is a path in M that (i) violates the formula and (ii) has lasso shape. As our logic $\mathsf{ELTL_{FT}}$ specifies counterexamples to the properties of fault-tolerant distributed algorithms, we are interested in this result in the following form: if the transition system *satisfies* an ELTL formula — which requires *one path* to satisfy the formula — then M has a path that (i) *satisfies* the formula and (ii) has lasso shape.

As observed above, counter systems are infinite state. Consequently, one cannot apply the results of [66] directly. However, using Proposition 2.10, we show that a similar result holds for counter systems of threshold automata and $\mathsf{ELTL_{FT}}$:

Proposition 4.2. *Given a threshold automaton TA and an $\mathsf{ELTL_{FT}}$ formula φ, if Sys(TA) $\models \mathbf{E}\,\varphi$, then there are an initial configuration $\sigma_1 \in I$ and a schedule $\tau \cdot \rho^\omega$ with the following properties:*

1. *the path satisfies the formula: $\mathsf{path}(\sigma_1, \tau \cdot \rho^\omega) \models \varphi$,*
2. *application of ρ forms a cycle: $\rho^k(\tau(\sigma_1)) = \tau(\sigma_1)$ for $k \ge 0$.*

Although in [43] we use Büchi automata to prove Proposition 4.2, we do not use Büchi automata in this paper. Since $\mathsf{ELTL_{FT}}$ uses only the temporal operators $\mathbf{F}$ and $\mathbf{G}$, we found it much easier to reason about the structure of $\mathsf{ELTL_{FT}}$ formulas directly (in the spirit of [29]) and then apply path reductions, rather than constructing the synchronous product of a Büchi automaton and of a counter system and then finding proper path reductions.

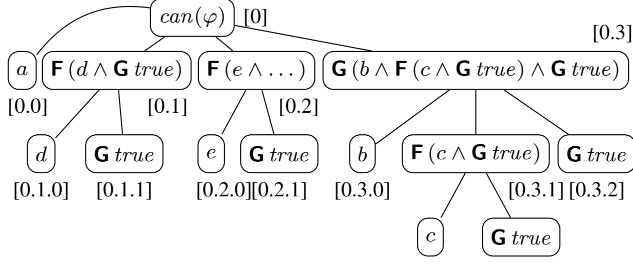

Figure 5. A canonical syntax tree of the $\mathsf{ELTL_{FT}}$ formula $\varphi \equiv \mathsf{F}\,(a \wedge \mathsf{F}\, d \wedge \mathsf{F}\, e \wedge \mathsf{G}\, b \wedge \mathsf{G}\,\mathsf{F}\, c)$ considered in Example 4.1. The labels $[w]$ denote identifiers of the tree nodes.

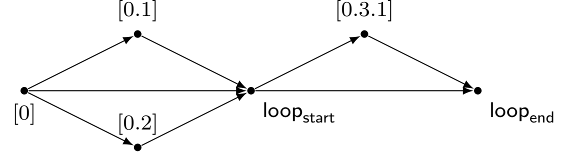

Figure 6. The cut graph of the canonical syntax tree in Figure 5

Although Proposition 4.2 guarantees counterexamples of lasso shape, it is not sufficient for model checking: (i) counter systems are infinite state, so that state enumeration may not terminate, and (ii) Proposition 4.2 does not provide us with bounds on the length of the lassos needed for bounded model checking. In the next section, we show how to split a lasso schedule in finite segments and to find constraints on lasso schedules that satisfy an $\mathsf{ELTL_{FT}}$ formula. In Section 6 we then construct shorter (bounded length) segments.

4.2 Characterizing Shapes of Lasso Schedules

We now construct a cut graph of an $\mathsf{ELTL_{FT}}$ formula: Cut graphs constrain the orders in which subformulas that start with the operator F are witnessed by configurations. The nodes of a cut graph correspond to cut points, while the edges constrain the order between the cut points. Using cut points, we give necessary and sufficient conditions for a lasso to satisfy an $\mathsf{ELTL_{FT}}$ formula in Theorems 4.12 and 4.13. Before defining cut graphs, we give the technical definitions of canonical formulas and canonical syntax trees.

Definition 4.3. *We inductively define canonical $\mathsf{ELTL_{FT}}$ formulas:*

- *if p is a propositional formula, then the formula $p \wedge \mathsf{G}\, true$ is a canonical formula of rank 0,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 1$, then the formula $p \wedge \mathsf{F}\psi_1 \wedge \dots \wedge \mathsf{F}\psi_k \wedge \mathsf{G}\, true$ is a canonical formula of rank 1,*
- *if p is a propositional formula and formulas $\psi_1, \dots, \psi_k$ are canonical formulas (of any rank) for some $k \geq 0$, and ψ_{k+1} is a canonical formula of rank 0 or 1, then the formula $p \wedge \mathsf{F}\psi_1 \wedge \dots \wedge \mathsf{F}\psi_k \wedge \mathsf{G}\psi_{k+1}$ is a canonical formula of rank 2.*

Example 4.4. Let p and q be propositional formulas. The formulas $p \wedge \mathsf{G}\, true$ and $true \wedge \mathsf{F}\,(q \wedge \mathsf{G}\, true) \wedge \mathsf{G}\,(p \wedge \mathsf{G}\, true)$ are canonical, while the formulas p, $\mathsf{F}\, q$, and $\mathsf{G}\, p$ are not canonical. Continuing Example 4.1, the canonical version of the formula $\mathsf{F}\,(a \wedge \mathsf{F}\, d \wedge \mathsf{F}\, e \wedge \mathsf{G}\, b \wedge \mathsf{G}\,\mathsf{F}\, c)$ is the formula $\mathsf{F}\,(a \wedge \mathsf{F}\,(d \wedge \mathsf{G}\, true) \wedge \mathsf{F}\,(e \wedge \mathsf{G}\, true) \wedge \mathsf{G}\,(b \wedge \mathsf{F}\,(c \wedge \mathsf{G}\, true) \wedge \mathsf{G}\, true))$. ◁

We will use formulas in the following canonical form in order to simplify presentation.

Observation 1. *The properties of canonical $\mathsf{ELTL_{FT}}$ formulas:*

1. *Every canonical formula consists of canonical subformulas of the form $p \wedge \mathsf{F}\psi_1 \wedge \dots \wedge \mathsf{F}\psi_k \wedge \mathsf{G}\psi_{k+1}$ for some $k \geq 0$, for a propositional formula p, canonical formulas $\psi_1, \dots, \psi_k$, and a formula ψ_{k+1} that is either canonical, or equals to $true$.*
2. *If a canonical formula contains a subformula $\mathsf{G}(\dots \wedge \mathsf{G}\psi)$, then ψ equals $true$.*

Proposition 4.5. *There is a function $can : \mathsf{ELTL_{FT}} \rightarrow \mathsf{ELTL_{FT}}$ that produces for each formula $\varphi \in \mathsf{ELTL_{FT}}$ an equivalent canonical formula $can(\varphi)$.*

For an $\mathsf{ELTL_{FT}}$ formula, there may be several equivalent canonical formulas, e.g., $p \wedge \mathsf{F}\,(q \wedge \mathsf{G}\, true) \wedge \mathsf{F}\,(p \wedge \mathsf{G}\, true) \wedge \mathsf{G}\, true$ and $p \wedge \mathsf{F}\,(p \wedge \mathsf{G}\, true) \wedge \mathsf{F}\,(q \wedge \mathsf{G}\, true) \wedge \mathsf{G}\, true$ differ in the order of F-subformulas. With the function can we fix one such a formula.

Canonical syntax trees. The canonical syntax tree of the formula introduced in Example 4.1 is shown in Figure 5. With $\mathbb{N}_0^*$ we denote the set of all finite words over natural numbers — these words are used as node identifiers.

Definition 4.6. *The* canonical syntax tree *of a formula $\varphi \in \mathsf{ELTL_{FT}}$ is the set $\mathcal{T}(\varphi) \subseteq \mathsf{ELTL_{FT}} \times \mathbb{N}_0^*$ constructed inductively as follows:*

1. *The tree contains the root node labeled with the canonical formula $can(\varphi)$ and id 0, that is, $\langle can(\varphi), 0\rangle \in \mathcal{T}(\varphi)$.*
2. *Consider a tree node $\langle \psi, w\rangle \in \mathcal{T}(\varphi)$ such that for some canonical formula $\psi' \in \mathsf{ELTL_{FT}}$ one of the following holds: (a) $\psi = \psi' = can(\varphi)$, or (b) $\psi = \mathsf{F}\psi'$, or (c) $\psi = \mathsf{G}\psi'$.*
   If ψ' is $p \wedge \mathsf{F}\psi_1 \wedge \dots \wedge \mathsf{F}\psi_k \wedge \mathsf{G}\psi_{k+1}$ for some $k \geq 0$, then the tree $\mathcal{T}(\varphi)$ contains a child node for each of the conjuncts of ψ', that is, $\langle p, w.0\rangle \in \mathcal{T}(\varphi)$, as well as $\langle \mathsf{F}\psi_i, w.i\rangle \in \mathcal{T}(\varphi)$ and $\langle \mathsf{G}\psi_j, w.j\rangle \in \mathcal{T}(\varphi)$ for $1 \leq i \leq k$ and $j = k+1$.

Observation 2. *The canonical syntax tree $\mathcal{T}(\varphi)$ of an $\mathsf{ELTL_{FT}}$ formula φ has the following properties:*

- *Every node $\langle \psi, w\rangle$ has the unique identifier w, which encodes the path to the node from the root.*
- *Every intermediate node is labeled with a temporal operator F or G over the conjunction of the formulas in the children nodes.*
- *The root node is labeled with the formula φ itself, and φ is equivalent to the conjunction of the root's children formulas, possibly preceded with a temporal operator F or G.*

The temporal formulas that appear under the operator G have to be dealt with by the loop part of a lasso. To formalize this, we say that a node with id $w \in \mathbb{N}_0^*$ is *covered* by a G-node, if w can be split into two words $u_1, u_2 \in \mathbb{N}_0^*$ with $w = u_1.u_2$, and there is a formula $\psi \in \mathsf{ELTL_{FT}}$ such that $\langle \mathsf{G}\,\psi, u_1\rangle \in \mathcal{T}(\varphi)$.

Cut graphs. Using the canonical syntax tree $\mathcal{T}(\varphi)$ of a formula φ, we capture in a so-called *cut graph* the possible orders in which formulas $\mathsf{F}\,\psi$ should be witnessed by configurations of a lasso-shaped path. We will then use the occurrences of the formula ψ to cut the lasso into bounded finite schedules.

Example 4.7. Figure 6 shows the cut graph of the canonical syntax tree in Figure 5. It consists of tree node ids for subformulas starting with F, and two special nodes for the start and the end of the loop. In the cut graph, the node with id 0 precedes the node with id 0.1, since at least one configuration satisfying $(a \wedge \mathsf{F}\,(d \wedge \dots) \wedge \dots)$ should occur on a path before (or at the same moment as) a state satisfying $(d \wedge \dots)$. Similarly, the node with id 0 precedes the node with id 0.2. The nodes with ids 0.1 and 0.2 do not have to precede each other, as the formulas d and e can be satisfied in either order. Since the nodes with the ids 0, 0.1, and 0.2 are not covered by a G-node, they both precede the loop start. The loop start precedes the node with id 0.3.1, as this node is covered by a G-node. ◁

Definition 4.8. *The* cut graph *$\mathcal{G}(\varphi)$ of an $\mathsf{ELTL_{FT}}$ formula is a directed acyclic graph $(\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$ with the following properties:*

1. *The set of nodes* $\mathcal{V}_\mathcal{G} = \{\mathsf{loop}_\mathsf{start}, \mathsf{loop}_\mathsf{end}\} \cup \{w \in \mathbb{N}_0^* \mid \exists \psi.\ \langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)\}$ *contains the tree ids that label* $\mathbf{F}$*-formulas and two special nodes* $\mathsf{loop}_\mathsf{start}$ *and* $\mathsf{loop}_\mathsf{end}$, *which denote the start and the end of the loop respectively.*
2. *The set of edges* $\mathcal{E}_\mathcal{G}$ *satisfies the following constraints:*
   (a) *Each tree node* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)$ *that is* not *covered by a* $\mathbf{G}$*-node precedes the loop start, i.e.,* $(w, \mathsf{loop}_\mathsf{start}) \in \mathcal{E}_\mathcal{G}$.
   (b) *For each tree node* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{T}(\varphi)$ *covered by a* $\mathbf{G}$*-node:*
      - *the loop start precedes* w, *i.e.,* $(\mathsf{loop}_\mathsf{start}, w) \in \mathcal{E}_\mathcal{G}$, *and*
      - w *precedes the loop end, i.e.,* $(w, \mathsf{loop}_\mathsf{end}) \in \mathcal{E}_\mathcal{G}$.
   (c) *For each pair of tree nodes* $\langle \mathbf{F}\psi_1, w\rangle, \langle \mathbf{F}\psi_2, w.i\rangle \in \mathcal{T}(\varphi)$ *not covered by a* $\mathbf{G}$*-node, we require* $(w, w.i) \in \mathcal{E}_\mathcal{G}$.
   (d) *For each pair of tree nodes* $\langle \mathbf{F}\psi_1, w_1\rangle, \langle \mathbf{F}\psi_2, w_2\rangle \in \mathcal{T}(\varphi)$ *that are both covered by a* $\mathbf{G}$*-node, we require either* $(w_1, w_2) \in \mathcal{E}_\mathcal{G}$, *or* $(w_2, w_1) \in \mathcal{E}_\mathcal{G}$ *(but not both).*

Definition 4.9. *Given a lasso* $\tau \cdot \rho^\omega$ *and a cut graph* $\mathcal{G}(\varphi) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, *we call a function* $\zeta : \mathcal{V}_\mathcal{G} \to \{0, \dots, |\tau| + |\rho| - 1\}$ *a* cut function, *if the following holds:*

- $\zeta(\mathsf{loop}_\mathsf{start}) = |\tau|$ *and* $\zeta(\mathsf{loop}_\mathsf{end}) = |\tau| + |\rho| - 1$,
- *if* $(v, v') \in \mathcal{E}_\mathcal{G}$, *then* $\zeta(v) \le \zeta(v')$.

We call the indices $\{\zeta(v) \mid v \in \mathcal{V}_\mathcal{G}\}$ the *cut points*. Given a schedule τ and an index $k : 0 \le k < |\tau| + |\rho|$, we say that the index k *cuts* τ into π' and π'', if $\tau = \pi' \cdot \pi''$ and $|\pi'| = k$.

Informally, for a tree node $\langle \mathbf{F}\,\psi, w\rangle \in \mathcal{T}(\varphi)$, a cut point $\zeta(w)$ witnesses satisfaction of $\mathbf{F}\,\psi$, that is, the formula ψ holds at the configuration located at the cut point. It might seem that Definitions 4.8 and 4.9 are too restrictive. For instance, assume that the node $\langle \mathbf{F}\,\psi, w\rangle$ is not covered by a $\mathbf{G}$-node, and there is a lasso schedule $\tau \cdot \rho^\omega$ that satisfies the formula φ at a configuration σ. It is possible that the formula ψ is witnessed only by a cut point inside the loop. At the same time, Definition 4.9 forces $\zeta(w) \le \zeta(\mathsf{loop}_\mathsf{start})$. We show that this problem is resolved by unwinding the loop K times for some $K \ge 0$, so that there is a cut function for the lasso with the prefix $\tau \cdot \rho^K$ and the loop ρ:

Proposition 4.10. *Let* φ *be an* $\mathsf{ELTL_{FT}}$ *formula,* σ *be a configuration and* $\tau \cdot \rho^\omega$ *be a lasso schedule applicable to* σ *such that* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ *holds. There is a constant* $K \ge 0$ *and a cut function* ζ *such that for every* $\langle \mathbf{F}\psi, w\rangle \in \mathcal{G}(\mathcal{T}(\varphi))$ *if* $\zeta(w)$ *cuts* $(\tau \cdot \rho^K) \cdot \rho$ *into* π' *and* π'', *then* ψ *is satisfied at the cut point, that is,* $\mathsf{path}(\pi'(\sigma), \pi'' \cdot \rho^\omega) \models \psi$.

Proof sketch. The detailed proof is given in [43]. We will present the required constant $K \ge 0$ and the cut function ζ. To this end, we use extreme appearances of $\mathbf{F}$-formulas (cf. [29, Sec. 4.3]) and use them to find ζ. An extreme appearance of a formula $\mathbf{F}\,\psi$ is the furthest point in the lasso that still witnesses ψ. There might be a subformula that is required to be witnessed in the prefix, but in $\tau \cdot \rho^\omega$ it is only witnessed by the loop. To resolve this, we replace τ by a a longer prefix $\tau \cdot \rho^K$, by unrolling the loop ρ several times; more precisely, K times, where K is the number of nodes that should precede the lasso start. In other words, if all extreme appearances of the nodes happen to be in the loop part, and they appear in the order that is against the topological order of the graph $\mathcal{G}(\mathcal{T}(\varphi))$, we unroll the loop K times (the number of nodes that have to be in the prefix) to find the prefix, in which the nodes respect the topological order of the graph. In the unrolled schedule we can now find extreme appearances of the required subformulas in the prefix. □

We show that to satisfy an $\mathsf{ELTL_{FT}}$ formula, a lasso should (i) satisfy propositional subformulas of $\mathbf{F}$-formulas in the respective cut points, and (ii) maintain the propositional formulas of $\mathbf{G}$-formulas from some cut point on. This is formalized as a witness.

In the following definition, we use a short-hand notation for propositional subformulas: given an $\mathsf{ELTL_{FT}}$-formula ψ and its canonical form $can(\psi) = \psi_0 \wedge \mathbf{F}\,\psi_1 \wedge \dots \wedge \mathbf{F}\,\psi_k \wedge \mathbf{G}\,\psi_{k+1}$, we use the notation $prop(\psi)$ to denote the formula ψ_0.

Definition 4.11. *Given a configuration* σ, *a lasso* $\tau \cdot \rho^\omega$ *applicable to* σ, *and an* $\mathsf{ELTL_{FT}}$ *formula* φ, *a cut function* ζ *of* $\mathcal{G}(\mathcal{T}(\varphi))$ *is a* witness *of* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$, *if the three conditions hold:*

(C1) *For* $can(\varphi) \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$:
  (a) $\sigma \models \psi_0$, *and*
  (b) $\mathsf{Cfgs}(\sigma, \tau \cdot \rho) \models prop(\psi_{k+1})$.

(C2) *For* $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ *with* $\zeta(v) < |\tau|$, *if* $\zeta(v)$ *cuts* $\tau \cdot \rho$ *into* π' *and* π'' *and* $\psi \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$, *then:*
  (a) $\pi'(\sigma) \models \psi_0$, *and*
  (b) $\mathsf{Cfgs}(\pi'(\sigma), \pi'') \models prop(\psi_{k+1})$.

(C3) *For* $\langle \mathbf{F}\psi, v\rangle \in \mathcal{T}(\varphi)$ *with* $\zeta(v) \ge |\tau|$, *if* $\zeta(v)$ *cuts* $\tau \cdot \rho$ *into* π' *and* π'' *and* $\psi \equiv \psi_0 \wedge \mathbf{F}\psi_1 \wedge \dots \wedge \mathbf{F}\psi_k \wedge \mathbf{G}\psi_{k+1}$, *then:*
  (a) $\pi'(\sigma) \models \psi_0$, *and*
  (b) $\mathsf{Cfgs}(\tau(\sigma), \rho) \models prop(\psi_{k+1})$.

Conditions (a) require that propositional formulas hold in a configuration, while conditions (b) require that propositional formulas hold on a finite suffix. Hence, to ensure that a cut function constitutes a witness, one has to check the configurations of a *fixed number of finite* paths (between the cut points). This property is crucial for the path reduction (see Section 6). Theorems 4.12 and 4.13 show that the existence of a witness is a sound and complete criterion for the existence of a lasso satisfying an $\mathsf{ELTL_{FT}}$ formula.

Theorem 4.12 (Soundness)**.** *Let* σ *be a configuration,* $\tau \cdot \rho^\omega$ *be a lasso applicable to* σ, *and* φ *be an* $\mathsf{ELTL_{FT}}$ *formula. If there is a witness of* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$, *then the lasso* $\tau \cdot \rho^\omega$ *satisfies* φ, *that is* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$.

Theorem 4.13 (Completeness)**.** *Let* φ *be an* $\mathsf{ELTL_{FT}}$ *formula,* σ *be a configuration and* $\tau \cdot \rho^\omega$ *be a lasso applicable to* σ *such that* $\mathsf{path}(\sigma, \tau \cdot \rho^\omega) \models \varphi$ *holds. There is a witness of* $\mathsf{path}(\sigma, (\tau \cdot \rho^K) \cdot \rho^\omega) \models \varphi$ *for some* $K \ge 0$.

Theorem 4.12 is proven for subformulas of φ by structural induction on the intermediate nodes of the canonical syntax tree. In the proof of Theorem 4.13 we use Proposition 4.10 to prove the points of Definition 4.11. (The detailed proofs are given in [43].)

4.3 Using Cut Graphs to Enumerate Shapes of Lassos

Proposition 4.2 and Theorem 4.13 suggest that in order to find a schedule that satisfies an $\mathsf{ELTL_{FT}}$ formula φ, it is sufficient to look for lasso schedules that can be cut in such a way that the configurations at the cut points and the configurations between the cut points satisfy certain propositional formulas. In fact, the cut points as defined by cut functions (Definition 4.9) are *topological orderings* of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$. Consequently, by enumerating the topological orderings of the cut graph $\mathcal{G}(\mathcal{T}(\varphi))$ we can enumerate the *lasso shapes*, among which there is a lasso schedule satisfying φ (if φ holds on the counter system). These shapes differ in the order, in which $\mathbf{F}$-subformulas of φ are witnessed. For this, one can use fast generation algorithms, e.g., [13].

Example 4.14. Consider the cut graph in Figure 6. The ordering of its vertices $0, 0.1, 0.2, \mathsf{loop}_\mathsf{start}, 0.3.1, \mathsf{loop}_\mathsf{end}$ corresponds to the lasso shape (a) shown in Figure 4, while the ordering $\mathsf{loop}_\mathsf{start}, 0, 0.2, 0.1, \mathsf{loop}_\mathsf{start}, 0.3.1, \mathsf{loop}_\mathsf{end}$ corresponds to the lasso shape (b). These are the two lasso shapes that one has to analyze, and they are the result of our construction using the cut graph. The other 18 lasso shapes in the figure are not required, and not constructed by our method. ◁

From this observation, we conclude that given a topological ordering $v_1, \dots, v_{|\mathcal{V}_\mathcal{G}|}$ of the cut graph $\mathcal{G}(\mathcal{T}(\varphi)) = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, one has to look for a lasso schedule that can be written as an alternating

sequence of configurations σ_i and schedules τ_j:

$$\sigma_0, \tau_0, \sigma_1, \tau_1, \ldots, \sigma_\ell, \tau_\ell, \ldots, \sigma_{|\mathcal{V}_\mathcal{G}|-1}, \tau_{|\mathcal{V}_\mathcal{G}|}, \sigma_{|\mathcal{V}_\mathcal{G}|}, \quad (2)$$

where $v_\ell = \mathsf{loop_{start}}$, $v_{|\mathcal{V}_\mathcal{G}|} = \mathsf{loop_{end}}$, and $\sigma_\ell = \sigma_{|\mathcal{V}_\mathcal{G}|}$. Moreover, by Definition 4.11, the sequence of configurations and schedules should satisfy ***(C1)–(C3)***, e.g., if a node v_i corresponds to the formula $\mathbf{F}\,(\psi_0 \wedge \cdots \wedge \mathbf{G}\,\psi_{k+1})$ and this formula matches Condition ***(C2)***, then the following should hold:

1. Configuration σ_i satisfies the propositional formula: $\sigma_i \models \psi_0$.
2. All configurations visited by the schedule $\tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}$ from the configuration σ_i satisfy the propositional formula $prop(\psi_{k+1})$. Formally, $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models prop(\psi_{k+1})$.

One can write an SMT query for the sequence (2) satisfying Conditions ***(C1)–(C3)***. However, this approach has two problems:

1. The order of rules in schedules $\tau_0, \ldots, \tau_{|\mathcal{V}_\mathcal{G}|}$ is not fixed. Non-deterministic choice of rules complicates the SMT query.
2. To guarantee completeness of the search, one requires a bound on the length of schedules $\tau_0, \ldots, \tau_{|\mathcal{V}_\mathcal{G}|}$.

For reachability properties these issues were addressed in [42] by showing that one only has to consider specific orders of the rules; so-called representative schedules. To lift this technique to $\mathsf{ELTL_{FT}}$, we are left with two issues:

1. The shortening technique applies to steady schedules, i.e., the schedules that do not change evaluation of the guards. Thus, we have to break the schedules $\tau_0, \ldots, \tau_{|\mathcal{V}_\mathcal{G}|}$ into steady schedules. This issue is addressed in Section 5.
2. The shortening technique preserves state reachability, e.g., after shortening of τ_i, the resulting schedule still reaches configuration σ_{i+1}. But it may violate an invariant such as $\mathsf{Cfgs}(\sigma_i, \tau_i \cdot \ldots \cdot \tau_{|\mathcal{V}_\mathcal{G}|}) \models prop(\psi_{k+1})$. This issue is addressed in Section 6.

5. Cutting Lassos with Threshold Guards

We introduce threshold graphs to cut a lasso into steady schedules, in order to apply the shortening technique of Section 6. Then, we combine the cut graphs and threshold graphs to cut a lasso into smaller finite segments, which can be first shortened and then checked with the approach introduced in Section 4.3.

Given a configuration σ, its context $\omega(\sigma)$ is the set that consists of the lower guards unlocked in σ and the upper guards locked in σ, i.e., $\omega(\sigma) = \Omega^{\mathrm{rise}} \cup \Omega^{\mathrm{fall}}$, where $\Omega^{\mathrm{rise}} = \{g \in \Phi^{\mathrm{rise}} \mid \sigma \models g\}$ and $\Omega^{\mathrm{fall}} = \{g \in \Phi^{\mathrm{fall}} \mid \sigma \not\models g\}$. As discussed in Example 2.9 on page 4, since the shared variables are never decreased, the contexts in a path are monotonically non-decreasing:

Proposition 5.1 (Prop. 3 of [42])**.** *If a transition t is enabled in a configuration σ, then $\omega(\sigma) \subseteq \omega(t(\sigma))$.*

Example 5.2. Continuing Example 2.9, which considers the TA in Figure 2. Both threshold guards γ_1 and γ_2 are false in the initial state σ. Thus, $\omega(\sigma) = \emptyset$. The transition $t = (r_1, 1)$ unlocks the guard γ_1, i.e., $\omega(t(\sigma)) = \{\gamma_1\}$. ◁

As the transitions of the counter system $\mathsf{Sys}(\mathsf{TA})$ never decrease shared variables, the loop of a lasso schedule must be steady:

Proposition 5.3. *For each configuration σ and a schedule $\tau \cdot \rho^\omega$, if $\rho^k(\tau(\sigma)) = \tau(\sigma)$ for $k \geq 0$, then the loop ρ is steady for $\tau(\sigma)$, that is, $\omega(\rho(\tau(\sigma))) = \omega(\tau(\sigma))$.*

In [42], Proposition 5.1 was used to cut a finite path into segments, one per context. We introduce threshold graphs and their topological orderings to apply this idea to lasso schedules.

Definition 5.4. *A* threshold graph *is $\mathcal{H}(\mathit{TA}) = (\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ such that:*

- *The vertices set $\mathcal{V}_\mathcal{H}$ contains the threshold guards and the special node $\mathsf{loop_{start}}$, i.e., $\mathcal{V}_\mathcal{H} = \Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}} \cup \{\mathsf{loop_{start}}\}$.*
- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{rise}}$ to a guard $g_2 \in \Phi^{\mathrm{rise}}$, if g_2 cannot be unlocked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \models g_2$ implies $\sigma \models g_1$.*
- *There is an edge from a guard $g_1 \in \Phi^{\mathrm{fall}}$ to a guard $g_2 \in \Phi^{\mathrm{fall}}$, if g_2 cannot be locked before g_1, i.e., $(g_1, g_2) \in \mathcal{E}_\mathcal{H}$, if for each configuration $\sigma \in \Sigma$, $\sigma \not\models g_2$ implies $\sigma \not\models g_1$.*

Note that the conditions in Definition 5.4 can be easily checked with an SMT solver, for all configurations.

Example 5.5. The threshold graph of the TA in Figure 2 has the vertices $\mathcal{V}_\mathcal{H} = \{\gamma_1, \gamma_2, \mathsf{loop_{start}}\}$ and the edges $\mathcal{E}_\mathcal{H} = \{(\gamma_1, \gamma_2)\}$. ◁

Similar to Section 4.3, we consider a topological ordering $g_1, \ldots, g_\ell, \ldots, g_{|\mathcal{V}_\mathcal{H}|}$ of the vertices of the threshold graph. The node $g_\ell = \mathsf{loop_{start}}$ indicates the point where a loop should start, and thus by Proposition 5.3, after that point the context does not change. Thus, we consider only the subsequence $g_1, \ldots, g_{\ell-1}$ and split the path $\mathsf{path}(\sigma, \tau \cdot \rho)$ of a lasso schedule $\tau \cdot \rho^\omega$ into an alternating sequence of configurations σ_i and schedules τ_0 and $t_j \cdot \tau_j$, for $1 \leq j < \ell$, ending up with the loop ρ (starting in $\sigma_{\ell-1}$ and ending in $\sigma_\ell = \sigma_{\ell-1}$):

$$\sigma_0, \tau_0, \sigma_1, (t_1 \cdot \tau_1), \ldots, \sigma_{\ell-2}, (t_{\ell-1} \cdot \tau_{\ell-1}), \sigma_{\ell-1}, \rho, \sigma_\ell \quad (3)$$

In this sequence, the transitions $t_1, \ldots, t_{\ell-1}$ change the context, and the schedules $\tau_0, \tau_1, \ldots, \tau_{\ell-1}, \rho$ are steady. Finally, we interleave a topological ordering of the vertices of the cut graph with a topological ordering of the vertices of the threshold graph. More precisely, we use a topological ordering of the vertices of the union of the cut graph and the threshold graph. We use the resulting sequence to cut a lasso schedule following the approach in Section 4.3 (cf. Equation (2)). By enumerating all such interleavings, we obtain all lasso shapes. Again, the lasso is a sequence of steady schedules and context-changing transitions.

Example 5.6. Continuing Example 1 given on page 5, we consider the lasso shapes that satisfy the $\mathsf{ELTL_{FT}}$ formula $\mathbf{G}\,\mathbf{F}\,\psi_{\mathrm{fair}} \wedge \boldsymbol{\kappa}[\ell_0] = 0 \wedge \mathbf{G}\,\boldsymbol{\kappa}[\ell_3] = 0$. Figure 7 shows the lasso shapes that have to be inspected by an SMT solver. In case (a), both threshold guards γ_1 and γ_2 are eventually changed to true, while the counter $\kappa[\ell_3]$ is never increased in a fair execution. For $n = 3t$, this is actually a counterexample to the correctness property explained in Example 1. In cases (b) and (c) at most one threshold guard is eventually changed to true, so these lasso shapes cannot produce a counterexample. ◁

In the following section, we will show how to shorten steady schedules, while maintaining Conditions ***(C1)–(C3)*** of Definition 4.11, required to satisfy the $\mathsf{ELTL_{FT}}$ formula.

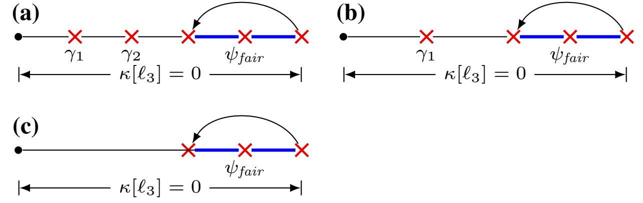

Figure 7. The shapes of lassos to check the correctness property in Example 3.3. Recall that γ_1 and γ_2 are the threshold guards, defined as $x \geq t + 1 - f$ and $x \geq n - t - f$ respectively.

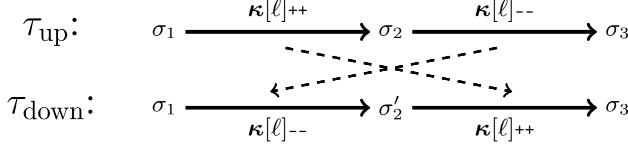

Figure 8. Changing the order of transitions can violate $\mathsf{ELTL_{FT}}$ formulas. If $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then for the upper schedule τ_{up} holds that $\mathsf{Cfgs}(\sigma_1, \tau_{\text{up}}) \models \boldsymbol{\kappa}[\ell] > 0$, while for the lower one this is not the case, because $\sigma_2' \not\models \boldsymbol{\kappa}[\ell] > 0$.

6. The Short Counterexample Property

Our verification approach focuses on counterexamples, and as discussed in Section 3, negations of specifications are expressed in $\mathsf{ELTL_{FT}}$. In the case of reachability properties, counterexamples are finite schedules reaching a bad state from an initial state. An efficient method for finding counterexamples to reachability can be found in [42]. It is based on the short counterexample property. Namely, it was proven that for each threshold automaton, there is a constant d such that if there is a schedule that reaches a bad state, then there must also exist an accelerated schedule that reaches that state in at most d transitions (i.e., d is the diameter of the counter system). The proof in [42] is based on the following three steps:

1. each finite schedule (which may or may not be a counterexample), can be divided into a few steady schedules,
2. for each of these steady schedules they find a representative, i.e., an accelerated schedule of bounded length, with the same starting and ending configurations as the original schedule,
3. at the end, all these representatives are concatenated in the same order as the original steady schedules.

This result guarantees that the system is correct if no counterexample to reachability properties is found using bounded model checking with bound d. In this section, we extend the technique from Point 2 from reachability properties to $\mathsf{ELTL_{FT}}$ formulas. The central result regarding Point 2 is the following proposition which is a specialization of [42, Prop. 7]:

Proposition 6.1. *Let* $\mathit{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, \mathit{RC})$ *be a threshold automaton. For every configuration* σ *and every steady schedule* τ *applicable to* σ*, there exists a steady schedule* $\mathsf{srep}[\sigma, \tau]$ *with the following properties:* $\mathsf{srep}[\sigma, \tau]$ *is applicable to* σ*,* $\mathsf{srep}[\sigma, \tau](\sigma) = \tau(\sigma)$*, and* $|\mathsf{srep}[\sigma, \tau]| \le 2 \cdot |\mathcal{R}|$.

We observe that the proposition talks about the first configuration σ and the last one $\tau(\sigma)$, while it ignores intermediate configurations. However, for $\mathsf{ELTL_{FT}}$ formulas, one has to consider all configurations in a schedule, and not just the first and the last one.

Example 6.2. Figure 8 shows the result of swapping transitions. The approaches by [50] and [42] are only concerned with the first and last configurations: they use the property that after swapping transitions, σ_3 is still reached from σ_1. The arguments used in [42, 50] do not care about the fact that the resulting path visits a different intermediate state (σ_2' instead of σ_2). However, if $\sigma_1.\boldsymbol{\kappa}[\ell] = 1$, then $\sigma_2.\boldsymbol{\kappa}[\ell] > 0$, while $\sigma_2'.\boldsymbol{\kappa}[\ell] = 0$. Hence, swapping transitions may change the evaluation of $\mathsf{ELTL_{FT}}$ formulas, e.g., $\mathbf{G}\,(\boldsymbol{\kappa}[\ell] > 0)$. ◁

Another challenge in verification of $\mathsf{ELTL_{FT}}$ formulas is that counterexamples to liveness properties are infinite paths. As discussed in Section 4, we consider infinite paths of lasso shape $\vartheta \cdot \rho^\omega$. For a finite part of a schedule, $\vartheta \cdot \rho$, satisfying an $\mathsf{ELTL_{FT}}$ formula, we show the existence of a new schedule, $\vartheta' \cdot \rho'$, of bounded length satisfying the same formula as the original one. Regarding the shortening, our approach uses a similar idea as the one from [42]. We follow modified steps from reachability analysis:

1. We split $\vartheta \cdot \rho$ into several steady schedules, using cut points introduced in Sections 4 and 5. The cut points depend not only on threshold guards, but also on the $\mathsf{ELTL_{FT}}$ formula φ representing the negation of a specification we want to check. Given such a steady schedule τ, each configuration of τ satisfies a set of propositional subformulas of φ, which are covered by the operator $\mathbf{G}$ in φ.
2. For each of these steady schedules we find a representative, that is, an accelerated schedule of bounded length that satisfies the necessary propositional subformulas as in the original schedule (i.e., not just that starting and ending configurations coincide).
3. We concatenate the obtained representatives in the original order.

In [43], we present the mathematical details for obtaining these representative schedules, and prove different cases that taken together establish our following main theorem:

Theorem 6.3. *Let* $\mathit{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, \mathit{RC})$ *be a threshold automaton, and let* $\mathit{Locs} \subseteq \mathcal{L}$ *be a set of locations. Let* σ *be a configuration, let* τ *be a steady conventional schedule applicable to* σ*, and let* ψ *be one of the following formulas:*

$$\bigvee_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] \neq 0, \quad \text{or} \quad \bigwedge_{\ell \in \mathit{Locs}} \boldsymbol{\kappa}[\ell] = 0.$$

If all configurations visited by τ *from* σ *satisfy* ψ*, i.e.,* $\mathsf{Cfgs}(\sigma, \tau) \models \psi$*, then there is a steady representative schedule* $\mathsf{repr}[\psi, \sigma, \tau]$ *with the following properties:*

a) The representative is applicable, and ends in the same final state: $\mathsf{repr}[\psi, \sigma, \tau]$ *is applicable to* σ*, and* $\mathsf{repr}[\psi, \sigma, \tau](\sigma) = \tau(\sigma)$*,*
b) The representative has bounded length: $|\mathsf{repr}[\psi, \sigma, \tau]| \le 6 \cdot |\mathcal{R}|$*,*
c) The representative maintains the formula ψ*. In other words,* $\mathsf{Cfgs}(\sigma, \mathsf{repr}[\psi, \sigma, \tau]) \models \psi$*,*
d) The representative is a concatenation of three representative schedules srep *from Proposition 6.1: there exist* τ_1*,* τ_2 *and* τ_3*, (possibly empty) subschedules of* τ*, such that* $\tau_1 \cdot \tau_2 \cdot \tau_3$ *is applicable to* σ*, and it holds that* $(\tau_1 \cdot \tau_2 \cdot \tau_3)(\sigma) = \tau(\sigma)$*, and* $\mathsf{repr}[\psi, \sigma, \tau] = \mathsf{srep}[\sigma, \tau_1] \cdot \mathsf{srep}[\tau_1(\sigma), \tau_2] \cdot \mathsf{srep}[(\tau_1 \cdot \tau_2)(\sigma), \tau_3]$.

Our approach is slightly different in the case when the formula ψ has a more complex form: $\bigwedge_{1 \le m \le n} \bigvee_{\ell \in \mathit{Locs}_m} \boldsymbol{\kappa}[\ell] \neq 0$, for $\mathit{Locs}_m \subseteq \mathcal{L}$, where $1 \le m \le n$ and $n \in \mathbb{N}$. In this case, our proof requires the schedule τ to have sufficiently large counter values. To ensure that there is an infinite schedule with sufficiently large counter values, we first prove that if a counterexample exists in a small system, there also exists one in a larger system, that is, we consider configurations where each counter is multiplied with a constant *finite multiplier* μ. For resilience conditions that do not correspond to parameterized systems (i.e., fix the system size to, e.g., $n = 4$) or pathological threshold automata, such multipliers may not exist. However, all our benchmarks have multipliers, and existence of multipliers can easily be checked using simple queries to SMT solvers in preprocessing. This additional restriction leads to slightly smaller bounds on the lengths of representative schedules:

Theorem 6.4. *Fix a threshold automaton* $\mathit{TA} = (\mathcal{L}, \mathcal{I}, \Gamma, \Pi, \mathcal{R}, \mathit{RC})$ *that has a finite multiplier* μ*, and a configuration* σ*. For an* $n \in \mathbb{N}$*, fix sets of locations* $\mathit{Locs}_m \subseteq \mathcal{L}$ *for* $1 \le m \le n$*. If* $\psi = \bigwedge_{1 \le m \le n} \bigvee_{\ell \in \mathit{Locs}_m} \boldsymbol{\kappa}[\ell] \neq 0$*, then for every steady conventional schedule* τ*, applicable to* σ*, with* $\mathsf{Cfgs}(\sigma, \tau) \models \psi$*, there exists a schedule* $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ *with the following properties:*

a) The representative is applicable and ends in the same final state: $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]$ *is a steady schedule applicable to* $\mu\sigma$*, and* $\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau](\mu\sigma) = \mu\tau(\mu\sigma)$*,*
b) The representative has bounded length: $|\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]| \le 4 \cdot |\mathcal{R}|$*,*
c) The representative maintains the formula ψ*. In other words,* $\mathsf{Cfgs}(\mu\sigma, \mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau]) \models \psi$*,*

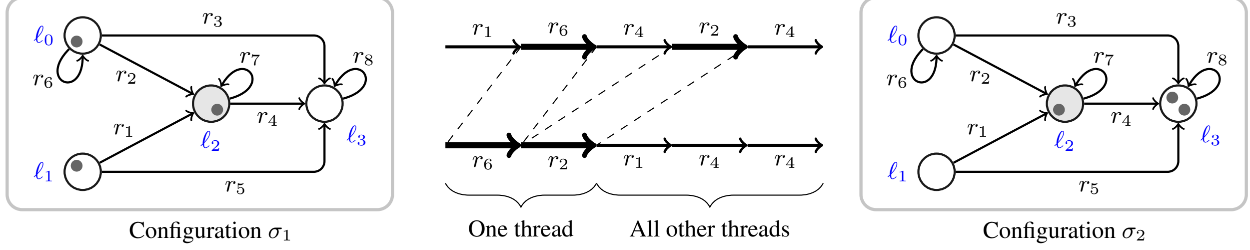

Figure 9. Example of constructing a representative schedule by moving a thread to the beginning. The number of dots in the local states correspond to counter values, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = \sigma_1.\boldsymbol{\kappa}[\ell_1] = \sigma_1.\boldsymbol{\kappa}[\ell_2] = 1$ and $\sigma_1.\boldsymbol{\kappa}[\ell_3] = 0$.

d) *The representative is a concatenation of two representative schedules* srep *from Proposition 6.1:*
$\mathsf{repr}_{\wedge\vee}[\psi, \mu\sigma, \mu\tau] = \mathsf{srep}[\mu\sigma, \tau] \cdot \mathsf{srep}[\tau(\mu\sigma), (\mu-1)\tau]$.

The main technical challenge for proving Theorems 6.3 and 6.4 is that we want to swap transitions and maintain $\mathsf{ELTL_{FT}}$ formulas at the same time. As discussed in Example 6.2, simply applying the ideas from the reachability analysis in [42, 50] is not sufficient.

We address this challenge by more refined swapping strategies depending on the property ψ of Theorem 6.3. For instance, the intuition behind $\bigvee_{\ell \in Locs} \boldsymbol{\kappa}[\ell] \neq 0$ is that in a given distributed algorithm, there should always be at least one process in one of the states in $Locs$. Hence, we would like to consider individual processes, but in the context of counter systems. Therefore, we introduce a mathematical notion we call a *thread*, which is a schedule that can be executed by an individual process. A thread is then characterized depending on whether it starts in $Locs$, ends in $Locs$, or visits $Locs$ at some intermediate step. Based on this characterization, we show that $\mathsf{ELTL_{FT}}$ formulas are preserved if we move carefully chosen threads to the beginning of a steady schedule (intuitively, this corresponds to τ_1, and τ_2 from Theorem 6.3). Then, we replace the threads, one by one, by their representative schedules from Proposition 6.1, and append another representative schedule for the remainder of the schedule. In this way, we then obtain the representative schedules in Theorem 6.3(d).

Example 6.5. We consider the TA in Figure 2, and show how a schedule $\tau = (r_1, 1), (r_6, 1), (r_4, 1), (r_2, 1), (r_4, 1)$ applicable to σ_1, with $\tau(\sigma_1) = \sigma_2$ can be shortened. Figure 9 follows this example where τ is the upper schedule. Assume that $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$, and that we want to construct a shorter schedule that produces a path that satisfies the same formula.

In our theory, subschedule $(r_1, 1), (r_4, 1)$ is a thread of σ_1 and τ for two reasons: (1) the counter of the starting local state of $(r_1, 1)$ is greater than 0, i.e., $\sigma_1.\boldsymbol{\kappa}[\ell_0] = 1$, and (2) it is a sequence of rules in the control flow of the threshold automaton, i.e., it starts from ℓ_0, then uses $(r_1, 1)$ to go to local state ℓ_2 and then $(r_4, 1)$ to arrive at ℓ_3. The intuition of (2) is that a thread corresponds to a process that executes the threshold automaton. Similarly, $(r_6, 1), (r_2, 1)$ and $(r_4, 1)$ are also threads of σ_1 and τ. In fact, we can show that each schedule can be decomposed into threads. Based on this, we analyze which local states are visited when a thread is executed.

Our formula $\mathsf{Cfgs}(\sigma_1, \tau) \models \boldsymbol{\kappa}[\ell_2] \neq 0$ talks about ℓ_2. Thus, we are interested in a thread that ends at ℓ_2, because after executing this thread, intuitively there will always be at least one process in ℓ_2, i.e., the counter $\boldsymbol{\kappa}[\ell_2]$ will be nonzero, as required. Such a thread will be moved to the beginning. We find that thread $(r_6, 1), (r_2, 1)$ meets this requirement. Similarly, we are also interested in a thread that starts from ℓ_2. Before we execute such a thread, at least one process must always be in ℓ_2, i.e., $\boldsymbol{\kappa}[\ell_2]$ will be nonzero. For this, we single out the thread $(r_4, 1)$, as it starts from ℓ_2.

Independently of the actual positions of these threads within a schedule, our condition $\boldsymbol{\kappa}[\ell_2] \neq 0$ is true *before* $(r_4, 1)$ starts, and *after* $(r_6, 1), (r_2, 1)$ ends. Hence, we move the thread $(r_6, 1), (r_2, 1)$ to the beginning, and obtain a schedule that ensures our condition in all visited configurations; cf. the lower schedule in Figure 9. Then we replace the thread $(r_6, 1), (r_2, 1)$, by a representative schedule from Proposition 6.1, and the remaining part $(r_1, 1), (r_4, 1), (r_4, 1)$, by another one. Indeed in our example, we could merge $(r_4, 1), (r_4, 1)$ into one accelerated transition $(r_4, 2)$ and obtain a schedule which is shorter than τ while maintaining $\boldsymbol{\kappa}[\ell_2] \neq 0$. ◁

7. Application of the Short Counterexample Property and Experimental Evaluation

7.1 SMT Encoding

We use the theoretical results from the previous section to give an efficient encoding of lasso-shaped executions in SMT with linear integer arithmetic. The definitions of counter systems in Section 2.2 directly tell us how to encode paths of the counter system. Definition 2.5 describes a configuration σ as tuple $(\boldsymbol{\kappa}, \mathbf{g}, \mathbf{p})$, where each component is encoded as a vector of SMT integer variables. Then, given a path $\sigma_0, t_1, \sigma_1, \dots, t_{k-1}, \sigma_{k-1}, t_k, \dots \sigma_k$ of length k, by $\boldsymbol{\kappa}^i$, $\mathbf{g}^i$, and $\mathbf{p}^i$ we denote the values of the vectors that correspond to σ_i, for $0 \le i \le k$. As the parameter values do not change, we use one copy of the variables $\mathbf{p}$ in our SMT encoding. By $\boldsymbol{\kappa}^i_\ell$, for $1 \le \ell \le |\mathcal{L}|$, we denote the ℓth component of $\boldsymbol{\kappa}^i$, that is, the counter corresponding to the number of processes in local state ℓ after the ith iteration. Definition 2.5 also gives us the constraint on the initial states, namely:

$$\mathrm{init}(0) \equiv \sum_{\ell \in \mathcal{I}} \boldsymbol{\kappa}^0_\ell = N(\mathbf{p}) \wedge \sum_{\ell \notin \mathcal{I}} \boldsymbol{\kappa}^0_\ell = 0 \wedge \mathbf{g}^0 = \mathbf{0} \wedge RC(\mathbf{p}) \quad (4)$$

Example 7.1. The TA from Figure 2 has four local states ℓ_0, ℓ_1, ℓ_2, ℓ_3 among which ℓ_0 and ℓ_1 are the initial states. In this example, $N(\mathbf{p})$ is $n - f$, and the resilience condition requires that there are less than a third of the processes faulty, i.e., $n > 3t$. We obtain $\mathrm{init}(0) \equiv \boldsymbol{\kappa}^0_0 + \boldsymbol{\kappa}^0_1 = n - f \wedge \boldsymbol{\kappa}^0_2 + \boldsymbol{\kappa}^0_3 = 0 \wedge x^0 = 0 \wedge n > 3t \wedge t \ge f \wedge f \ge 0$. The constraint is in linear integer arithmetic. ◁

Further, Definition 2.8 encodes the transition relation. A transition is identified by a rule and an acceleration factor. A rule is identified by threshold guards $\varphi^{\le}$ and $\varphi^{>}$, local states *from* and *to* between which processes are moved, and by $\mathbf{u}$, which defines the increase of shared variables. As according to Section 5 only a fixed number of transitions change the context and thus may change the evaluation of $\varphi^{\le}$ and $\varphi^{>}$, we do not encode $\varphi^{\le}$ and $\varphi^{>}$ for each rule. In fact, we check the guards $\varphi^{\le}$ and $\varphi^{>}$ against a fixed number of configurations, which correspond to the cut points defined by the threshold guards. The acceleration factor δ is indeed the only variable in a transition, and the SMT solver has to find assignments

of these factors. Then this transition from the ith to the $(i+1)$th configuration is encoded using rule $r = (\mathit{from}, \mathit{to}, \varphi^{\le}, \varphi^{>}, \mathbf{u})$ as follows:

$$\begin{aligned}
T(i,r) &\equiv \mathit{Move}(\mathit{from},\mathit{to},i) \land \mathit{IncShd}(\mathbf{u},i) \qquad (5)\\
\mathit{Move}(\ell,\ell',i) &\equiv \ell \ne \ell' \to \boldsymbol{\kappa}^i_\ell - \boldsymbol{\kappa}^{i+1}_\ell = \delta^{i+1} = \boldsymbol{\kappa}^{i+1}_{\ell'} - \boldsymbol{\kappa}^i_{\ell'}\\
&\land\ \ \ell = \ell' \to \left(\boldsymbol{\kappa}^i_\ell = \boldsymbol{\kappa}^{i+1}_\ell \land \boldsymbol{\kappa}^{i+1}_{\ell'} = \boldsymbol{\kappa}^i_{\ell'}\right)\\
&\land \bigwedge_{s \in \mathcal{L}\setminus\{\ell,\ell'\}} \boldsymbol{\kappa}^i_s = \boldsymbol{\kappa}^{i+1}_s\\
\mathit{IncShd}(\mathbf{u},i) &\equiv \mathbf{g}^{i+1} - \mathbf{g}^i = \delta^{i+1}\cdot\mathbf{u}
\end{aligned}$$

Given a schedule τ, we encode in linear integer arithmetic the paths that follow this schedule from an initial state as follows:

$$E(\tau) \equiv \mathrm{init}(0) \land T(0,r_1) \land T(1,r_2) \land \dots$$

We can now ask the SMT solver for assignments of the parameters as well as the factors $\delta^1, \delta^2, \dots$ in order to check whether a path with this sequence of rules exists. Note that some factors can be equal to 0, which means that the corresponding rule does not have any effect (because no process executes it). If τ encodes a lasso shape, and the SMT solver reports a satisfying assignment, this assignment is a counterexample. If the SMT solver reports unsat on all lassos discussed in Section 5, then there does not exists a counterexample and the algorithm is verified.

Example 7.2. In Example 3.3 we have seen the fairness requirement ψ_{fair}, which is a property of a configuration that can be encoded as $\mathrm{fair}(i) \equiv \boldsymbol{\kappa}^i_1 = 0 \land (x^i \ge t+1 \to \boldsymbol{\kappa}^i_0 = 0 \land \boldsymbol{\kappa}^i_1 = 0) \land (x^i \ge n-t \to \boldsymbol{\kappa}^i_0 = 0 \land \boldsymbol{\kappa}^i_2 = 0)$, which is a formula in linear integer arithmetic. Then, e.g., $\mathrm{fair}(5)$ encodes that the fifth configuration satisfies the predicate. Such state formulas can be added as conjunct to the formula $E(\tau)$ that encodes a path. ◁

As discussed in Sections 4 and 5 we have to encode lassos of the form $\vartheta\cdot\rho^\omega$ starting from an initial configuration σ. We immediately obtain a finite representation by encoding the fixed length execution $E(\vartheta\cdot\rho)$ as above, and adding the constraint that applying ρ returns to the start of the lasso loop, that is, $\vartheta(\sigma) = \rho(\vartheta(\sigma))$. In SMT this is directly encoded as equality of integer variables.

7.2 Generating the SMT Queries

The high-level structure of the verification algorithm is given in Figure 3 on page 6. In this section, we give the details of the procedure `check_one_order`, whose pseudo code is given in Figure 10. It receives as the input the following parameters: a threshold automaton TA, an $\mathsf{ELTL}_{\mathsf{FT}}$ formula φ, a cut graph $\mathcal{G}$ of φ, a threshold graph $\mathcal{H}$ of TA, and a topological order $\prec$ on the vertices of the graph $\mathcal{G}\cup\mathcal{H}$.

The procedure `check_one_order` constructs SMT assertions about the configurations of the lassos that correspond to the order $\prec$. As explained in Section 7.1, an ith configuration is defined by the vectors of SMT variables $(\boldsymbol{\kappa}^i, \mathbf{g}^i, \mathbf{p})$. We use two global variables: the number fn of the configuration under construction, and the number fs of the configuration that corresponds to the loop start. Thus, with the expressions $\boldsymbol{\kappa}^{\mathsf{fn}}$ and $\mathbf{g}^{\mathsf{fn}}$ we refer to the SMT variables of the configuration whose number is stored in fn.

In the pseudocode in Figure 10, we call `SMT_assert(`$\boldsymbol{\kappa}^{\mathsf{fn}}$`, `$\mathbf{g}^{\mathsf{fn}}$`, `$\mathbf{p} \models \psi$`)` to add an assertion ψ about the configuration $(\boldsymbol{\kappa}^{\mathsf{fn}}, \mathbf{g}^{\mathsf{fn}}, \mathbf{p})$ to the SMT query. Finally, the call `SMT_sat()` returns true, only if there is a satisfying assignment for the assertions collected so far. Such an assignment can be accessed with `SMT_model()` and gives the values for the configurations and acceleration factors, which together constitute a witness lasso.

The procedure `check_one_order` creates the assertions about the initial configurations. The assertions consist of: the assumptions $\mathrm{init}(0)$ about the initial configurations of the threshold automaton, the top-level propositional formula ψ_0, and the invariant propositional formula ψ_{k+1} that should hold from the initial configuration on. By writing `assume(`$\psi = \psi_0 \land \mathbf{F} \land \psi_1 \dots \mathbf{F}\,\psi_k \land \mathbf{G}\,\psi_{k+1}$`)`, we extract the subformulas of a canonical formula ψ (see Section 4.2). The procedure finds the minimal node in the order $\prec$ on the nodes of the graph $\mathcal{G}\cup\mathcal{H}$ and calls the procedure `check_node` for the initial node, the initial invariant ψ_{k+1}, and the empty context $\emptyset$.

The procedure `check_node` is called with a node v of the graph $\mathcal{G}\cup\mathcal{H}$ as a parameter. It adds assertions that encode a finite path and constraints on the configurations of this path. The finite path leads from the configuration that corresponds to the node v to the configuration that corresponds to v's successor in the order $\prec$.

```
1  variables fn, fs; // the current configuration number and the loop start
2  // Try to find a witness lasso for: a threshold automaton TA,
3  // an ELTL_FT formula φ, a cut graph 𝒢, a threshold graph ℋ, and
4  // a topological order ≺ on the nodes of 𝒢 ∪ ℋ.
5  procedure check_one_order(TA, φ, 𝒢, ℋ, ≺):
6    fn := 0; fs := 0;
7    SMT_start(); // start (or reset) the SMT solver
8    assume(can(φ) = ψ_0 ∧ F ψ_1 ∧ ··· ∧ F ψ_k ∧ G ψ_{k+1});
9    SMT_assert(κ^0, g^0, p ⊨ init(0) ∧ ψ_0 ∧ ψ_{k+1}); // see Equation 4
10   v_0 := min_≺(V_𝒢 ∪ V_ℋ); // the minimal node w.r.t. the linear order ≺
11   check_node(𝒢, ℋ, ≺, v_0, ψ_{k+1}, ∅);
12
13 // Try to find a witness lasso starting with the node v and the context Ω,
14 // while preserving the invariant ψ_inv.
15 recursive procedure check_node(𝒢, ℋ, ≺, v, ψ_inv, Ω):
16   if not SMT_sat() then:
17     return no_witness;
18   case (a) v ∈ V_𝒢 \ {loop_start, loop_end}:
19     find ψ s.t. ⟨F ψ, v⟩ ∈ 𝒯(φ); // v labels a formula in the syntax tree
20     assume(ψ = ψ_0 ∧ F ψ_1 ∧ ··· ∧ F ψ_k ∧ G ψ_{k+1});
21     SMT_assert(κ^fn, g^fn, p ⊨ ψ_0);
22     push_segment(ψ_inv ∧ ψ_{k+1});
23     v' := min_≺(V_𝒢 ∪ V_ℋ) ∩ {w : v ≺ w}; // the next node after v
24     check_node(𝒢, ℋ, ≺, v', ψ_inv ∧ ψ_{k+1}, Ω);
25   case (b) v ∈ V_ℋ \ {loop_start, loop_end}: // v is a threshold guard
26     if v ∈ Φ^rise then: // v is an unlocking guard, e.g., x ≥ t + 1 − f
27       push_segment(ψ_inv); // one rule unlocks v
28       SMT_assert(κ^fn, g^fn, p ⊨ v); // v is unlocked
29       push_segment(ψ_inv); // execute all unlocked rules
30       v' := min_≺(V_𝒢 ∪ V_ℋ) ∩ {w : v ≺ w}; // the next node after v
31       check_node(𝒢, ℋ, ≺, v', ψ_inv, Ω ∪ {v});
32     else: /* v ∈ Φ^fall, e.g., x < f, similar to the locking case: use ¬v */
33   case (c) v = loop_start:
34     fs := fn; // the loop starts at the current configuration
35     push_segment(ψ_inv); // execute all unlocked rules
36     v' := min_≺(V_𝒢 ∪ V_ℋ) ∩ {w : v ≺ w}; // the next node after v
37     check_node(𝒢, ℋ, ≺, v', ψ_inv, Ω);
38   case (d) v = loop_end:
39     SMT_assert(κ^fn = κ^fs ∧ g^fn = g^fs); // close the loop
40     if SMT_sat() then:
41       return witness(SMT_model())
42
43 // Encode a segment of rules as prescribed by [42] and Theorems 6.3–6.4.
44 procedure push_segment(ψ_inv):
45   // find the number of schedules to repeat in (d) of Theorems 6.3, 6.4
46   nrepetitions := compute_repetitions(ψ_inv);
47   r_1, ..., r_k := compute_rules(Ω); // use sschema_Ω from [42]
48   for _ from 1 to nrepetitions:
49     for j from 1 to k:
50       SMT_assert(κ^fn, g^fn, p ⊨ ψ_inv);
51       SMT_assert(T(fn, r_j)); // modify the counters as in Equation 5
52       fn := fn + 1; // move to the next configuration
```

Figure 10. Checking one topological order with SMT.

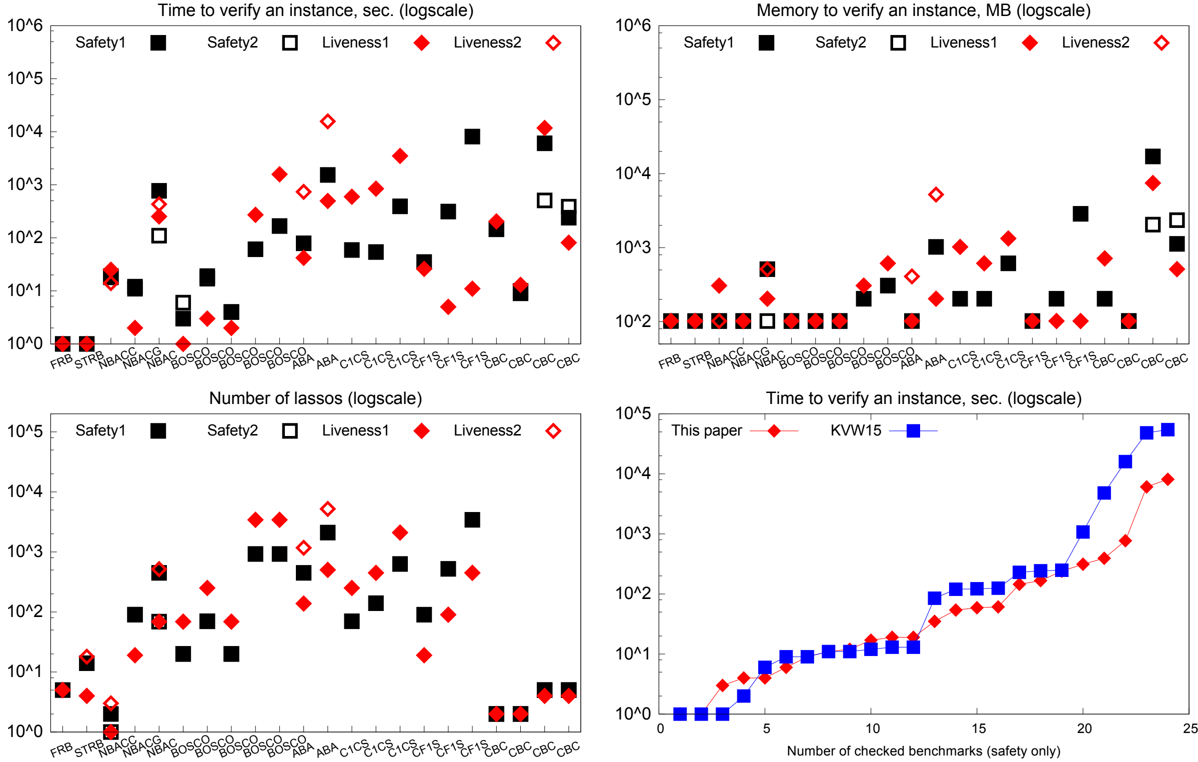

Figure 11. The plots summarize the following results of running our implementation on all benchmarks: used time in seconds (top left), used memory in megabytes (top right), the number of checked lassos (bottom left), time used both by our implementation and [42] to check *safety only* (bottom right). Several occurrences of the same benchmark correspond to different cases, such as $f > 1$, $f = 1$, and $f = 0$. Symbols ■ and □ correspond to the safety properties of each benchmark, while symbols ♦ and ♢ correspond to the liveness properties.

The constraints depend on v's origin: (a) v labels a formula $\mathbf{F}\,\psi$ in the syntax tree of φ, (b) v carries a threshold guard from the set $\Phi^{\mathrm{rise}} \cup \Phi^{\mathrm{fall}}$, (c) v denotes the loop start, or (d) v denotes the loop end. In case (a), we add an SMT assertion that the current configuration satisfies the propositional formula $prop(\psi)$ (line 21), and add a sequence of rules that leads to v's successor while maintaining the invariants ψ_{inv} of the preceding nodes and the v's invariant ψ_{k+1} (line 22). In case (b), in line 27, we add a sequence of rules, one of which should unlock (resp. lock) the threshold guard in $v \in \Phi^{\mathrm{rise}}$ (resp. $v \in \Phi^{\mathrm{fall}}$). Then, in line 29, we add a sequence of rules that leads to a configuration of v's successor. All added configurations are required to satisfy the current invariant ψ_{inv}. As the threshold guard in v is now unlocked (resp. locked), we include the guard (resp. its negation) in the current context Ω. In case (c), we store the current configuration as the loop start in the variable fs and, as in (a) and (b), add a sequence of rules leading to v's successor. Finally, in case (d), we should have reached the ending configuration that coincides with the loop start. To this end, in line 39, we add the constraint that forces the counters of both configurations to be equal. At this point, all the necessary SMT constraints have been added, and we call `SMT_sat` to check whether there is an assignment that satisfies the constraints. If there is one, we report it as a lasso witnessing the $\mathsf{ELTL_{FT}}$-formula φ that consists of: the concrete parameter values, the values of the counters and shared variables for each configuration, and the acceleration factors. Otherwise, we report that there is no witness lasso for the formula φ.

The procedure `push_segment` constructs a sequence of currently unlocked rules, as in the case of reachability [42]. However, this sequence should be repeated several times, as required by Theorems 6.3 and 6.4. Moreover, the freshly added configurations are required to satisfy the current invariant ψ_{inv}.

7.3 Experiments

We extended the tool ByMC [42] with our technique and conducted experiments[2] with the freely available benchmarks from [42]: folklore reliable broadcast (FRB) [14], consistent broadcast (STRB) [64], asynchronous Byzantine agreement (ABA) [11], condition-based consensus (CBC) [52], non-blocking atomic commitment (NBAC and NBACC [61] and NBACG [37]), one-step consensus with zero degradation (CF1S [21]), consensus in one communication step (C1CS [12]), and one-step Byzantine asynchronous consensus (BOSCO [63]). These threshold-guarded fault-tolerant distributed algorithms are encoded in a parametric extension of Promela.

Negations of the safety and liveness specifications of our benchmarks — written in $\mathsf{ELTL_{FT}}$ — follow three patterns: unsafety $\mathbf{E}\,(p \wedge \mathbf{F}\,q)$, non-termination $\mathbf{E}\,(p \wedge \mathbf{G}\,\mathbf{F}\,r \wedge \mathbf{G}\,q)$, and non-response $\mathbf{E}\,(\mathbf{G}\,\mathbf{F}\,r \wedge \mathbf{F}\,(p \wedge \mathbf{G}\,q))$. The propositions p, q, and r follow the syntax of *pform* (cf. Table 1), e.g., $p \equiv \bigwedge_{\ell \in Locs_1} \kappa[\ell] = 0$ and $q \equiv \bigvee_{\ell \in Locs_2} \kappa[\ell] \neq 0$ for some sets of locations $Locs_1$ and $Locs_2$.

The results of our experiments are summarized in Figure 11. Given the properties of the distributed algorithms found in the literature, we checked for each benchmark one or two safety properties (depicted with ■ and □) and one or two liveness properties (depicted with ♦ and ♢). For each benchmark, we display the running times and the memory used together by ByMC and the SMT

[2] The details on the experiments and the artifact are available at: `http://forsyte.at/software/bymc/popl17-artifact`

solver Z3 [20], as well as the number of exercised lasso shapes as discussed in Section 5.

For safety properties, we compared our implementation against the implementation of [42]. The results are summarized the bottom right plot in Figure 11, which shows that there is no clear winner. For instance, our implementation is 170 times faster on BOSCO for the case $n > 5t$. However, for the benchmark ABA we experienced a tenfold slowdown. In our experiments, attempts to improve the SMT encoding for liveness usually impaired safety results.

Our implementation has verified safety and liveness of all ten parameterized algorithms in less than a day. Moreover, the tool reports counterexamples to liveness of CF1S and BOSCO exactly for the cases predicted by the distributed algorithms literature, i.e., when there are not enough correct processes to reach consensus in one communication step. Noteworthy, liveness of only the two simplest benchmarks (STRB and FRB) had been automatically verified before [40].

8. Conclusions

Parameterized verification approaches the problem of verifying systems of thousands of processes by proving correctness for all system sizes. Although the literature predominantly deals with safety, parameterized verification for liveness is of growing interest, and has been addressed mostly in the context of programs that solve mutual exclusion or dining philosophers [4, 30, 31, 59]. These techniques do not apply to fault-tolerant distributed algorithms that have arithmetic conditions on the fraction of faults, threshold guards, and typical specifications that evaluate a global system state.

Parameterized verification is in general undecidable [3]. As recently surveyed by Bloem et al. [9], one can escape undecidability by restricting, e.g., communication semantics, local state space, the local control flow, or the temporal logic used for specifications. Hence, we make explicit the required restrictions. On the one hand, these restrictions still allow us to model fault-tolerant distributed algorithms and their specifications, and on the other hand, they give rise to a practical verification method. The restrictions are on the local control flow (loops) of processes (Section 2.1), as well as on the temporal operators and propositional formulas (Section 3). We conjecture that lifting these restrictions quite quickly leads to undecidability again. In addition, we justify our restrictions with the considerable number of benchmarks [11, 12, 14, 21, 37, 52, 61, 63, 64] that fit into our fragment, and with the convincing experimental results from Figure 11.

Our main technical contribution is to combine and extend several important techniques: First, we extend the ideas by Etessami et al. [29] to reason about shapes of infinite executions of lasso shape. These executions are counterexample candidates. Then we extend reductions introduced by Lipton [50] to deal with $\mathsf{ELTL}_{\mathsf{FT}}$ formulas. (Techniques that extend Lipton's in other directions can be found in [19, 22, 25, 34, 44, 47].) Our reduction is specific to threshold guards which are typical for fault-tolerant distributed algorithms and are found in domain-specific languages. Using on our reduction we apply acceleration [6, 44] in order to arrive at our short counterexample property.

Our short counterexample property implies a completeness threshold, that is, a bound b that ensures that if no lasso of length up to b is satisfying an $\mathsf{ELTL}_{\mathsf{FT}}$ formula, then there is no infinite path satisfying this formula. For linear temporal logic with the F and G operators, Kroening et al. [46] prove bounds on the completeness thresholds on the level of Büchi automata. Their bound involves the recurrence diameter of the transition systems, which is prohibitively large for counter systems. Similarly, the general method to transfer liveness with fairness to safety checking by Biere et al. [8] leads to an exponential growth of the diameter, and thus to too large values of b. Hence, we decided to conduct an analysis on the level of threshold automata, accelerated counter systems, and a fragment of the temporal logic, which allows us to exploit specifics of the domain, and get bounds that can be used in practice.

Acceleration has been applied for parameterized verification by means of regular model checking [1, 10, 58, 62]. As noted by Fisman et al. [33], to verify fault-tolerant distributed algorithms, one would have to intersect the regular languages that describe sets of states with context-free languages that enforce the resilience condition (e.g., $n > 3t$). Our approach of reducing to SMT handles resilience conditions naturally in linear integer arithmetic.

There are two reasons for our restrictions in the temporal logic: On one hand, in our benchmarks, there is no need to find counterexamples that contain a configuration that satisfies $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$ for some $\ell, \ell' \in \mathcal{L}$. One would only need such a formula to specify requirement that at least one process is at location ℓ and at least one process is at location ℓ' (the disjunction would be negated in the specification), which is unnatural for fault-tolerant distributed algorithms. On the other hand, enriching our logic with $\bigvee_{i \in Locs} \boldsymbol{\kappa}[i] = 0$ allows one to express tests for zero in the counter system, which leads to undecidability [9]. For the same reason, we avoid disjunction, as it would allow one to indirectly express test for zero: $\boldsymbol{\kappa}[\ell] = 0 \vee \boldsymbol{\kappa}[\ell'] = 0$.

The restrictions we put on threshold automata are justified from a practical viewpoint of our application domain, namely, threshold-guarded fault-tolerant algorithms. We assumed that all the cycles in threshold automata are simple (while the benchmarks have only self-loops or cycles of length 2). As our analysis already is quite involved, these restrictions allow us to concentrate on our central results without obfuscating the notation and theoretical results. Still, from a theoretical viewpoint it might be interesting to relax the restrictions on cycles in the future.

More generally, these restrictions allowed us to develop a completely automated verification technique. In general, there is a trade-off between degree of automation and generality. Our method is completely automatic, but our input language cannot compete in generality with mechanized proof methods that rely heavily on human expertise, e.g., IVY [55], Verdi [68], IronFleet [38], TLAPS [16].

References

[1] P. A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In *CAV*, LNCS, pages 305–318, 1998.

[2] F. Alberti, S. Ghilardi, and E. Pagani. Counting constraints in flat array fragments. In *IJCAR*, volume 9706 of *LNCS*, pages 65–81, 2016.

[3] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *IPL*, 15:307–309, 1986.

[4] M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting fair non-termination in multithreaded programs. In *CAV*, pages 210–226, 2012.

[5] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[6] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[7] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: a framework for implementing fault-tolerant distributed algorithms. In *DSN*, pages 1–8, 2013.

[8] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2): 160–177, 2002.

[9] R. Bloem, S. Jacobs, A. Khalimov, I. Konnov, S. Rubin, H. Veith, and J. Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[10] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, LNCS, pages 372–386, 2004.

[11] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[12] F. V. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *PaCT*, volume 2127 of *LNCS*, pages 42–50, 2001.

[13] E. R. Canfield and S. G. Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995.

[14] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[15] B. Charron-Bost and S. Merz. Formal verification of a consensus algorithm in the heard-of model. *IJSI*, 3(2–3):273–303, 2009.

[16] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, volume 6173 of *LNCS*, pages 142–148, 2010.

[17] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[18] E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: the environment abstraction framework for model checking concurrent systems. In *TACAS'08/ETAPS'08*, pages 33–47. Springer, 2008.

[19] E. Cohen and L. Lamport. Reduction in TLA. In *CONCUR*, volume 1466 of *LNCS*, pages 317–331, 1998.

[20] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.

[21] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *DSN*, pages 137–146, 2006.

[22] T. W. Doeppner. Parallel program correctness through refinement. In *POPL*, pages 155–169, 1977.

[23] C. Drăgoi, T. A. Henzinger, and D. Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

[24] C. Drăgoi, T. A. Henzinger, H. Veith, J. Widder, and D. Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

[25] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[26] E. Emerson and K. Namjoshi. Reasoning about rings. In *POPL*, pages 85–94, 1995.

[27] E. A. Emerson and V. Kahlon. Model checking guarded protocols. In *LICS*, pages 361–370. IEEE, 2003.

[28] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS*, pages 352–359. IEEE Computer Society, 1999.

[29] K. Etessami, M. Y. Vardi, and T. Wilke. First-order logic with two variables and unary temporal logic. *Inf. Comput.*, 179(2):279–295, 2002.

[30] Y. Fang, N. Piterman, A. Pnueli, and L. D. Zuck. Liveness with invisible ranking. *STTT*, 8(3):261–279, 2006.

[31] A. Farzan, Z. Kincaid, and A. Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196, 2016.

[32] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[33] D. Fisman, O. Kupferman, and Y. Lustig. On verifying fault tolerance of distributed protocols. In *TACAS*, volume 4963 of *LNCS*, pages 315–331. Springer, 2008.

[34] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291, 2005.

[35] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39:675–735, 1992.

[36] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. Tutorial on parameterized model checking of fault-tolerant distributed algorithms. In *Formal Methods for Executable Software Models*, LNCS, pages 122–171. Springer, 2014.

[37] R. Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15 (1):17–25, 2002.

[38] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *SOSP*, pages 1–17, 2015.

[39] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[40] A. John, I. Konnov, U. Schmid, H. Veith, and J. Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *FMCAD*, pages 201–209, 2013.

[41] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN PLDI*, pages 179–188, 2007.

[42] I. Konnov, H. Veith, and J. Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

[43] I. Konnov, M. Lazić, H. Veith, and J. Widder. A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms. *CoRR*, abs/1608.05327, 2016. URL `http://arxiv.org/abs/1608.05327`.

[44] I. Konnov, H. Veith, and J. Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 2016. Accepted manuscript available online: 10-MAR-2016. `http://dx.doi.org/10.1016/j.ic.2016.03.006`.

[45] I. Konnov, H. Veith, and J. Widder. What you always wanted to know about model checking of fault-tolerant distributed algorithms. In *PSI 2015, Revised Selected Papers*, volume 9609 of *LNCS*, pages 6–21. Springer, 2016.

[46] D. Kroening, J. Ouaknine, O. Strichman, T. Wahl, and J. Worrell. Linear completeness thresholds for bounded model checking. In *CAV*, volume 6806 of *LNCS*, pages 557–572, 2011.

[47] L. Lamport and F. B. Schneider. Pretending atomicity. Technical Report 44, SRC, 1989.

[48] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370, 2016.

[49] P. Lincoln and J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *FTCS*, pages 402–411, 1993.

[50] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[51] B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. I. *Acta Informatica*, 21(2): 125–169, 1984.

[52] A. Mostéfaoui, E. Mourgaya, P. R. Parvédy, and M. Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

[53] Netflix. 5 lessons we have learned using AWS. 2010. retrieved on Nov. 7, 2016. `http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html`.

[54] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC*, pages 305–320, 2014.

[55] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630, 2016.

[56] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[57] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making fast consensus generally faster. In *DSN*, pages 156–167, 2016.

[58] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV*, LNCS, pages 328–343, 2000.

[59] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0,1,\infty)$- counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. 2002.

[60] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *ECEASST*, 72, 2015.

[61] M. Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *HASE*, pages 209–214, 1997.

[62] V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science*, 149(1):79–96, 2006.

[63] Y. J. Song and R. van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *DISC*, volume 5218 of *LNCS*, pages 438–450, 2008.

[64] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

[65] TLA. TLA+ toolbox. `http://research.microsoft.com/en-us/um/people/lamport/tla/tools.html`.

[66] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 322–331, 1986.

[67] K. von Gleissenthall, N. Bjørner, and A. Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, pages 599–613, 2016.

[68] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.