Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms
Nathalie Bertrand, Bastien Thomas, Josef Widder

To cite this version:
Nathalie Bertrand, Bastien Thomas, Josef Widder. Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms. Concur 2021 - International Conference on Concurrency Theory, Aug 2021, Paris, France. pp.1-17, 10.4230/LIPIcs.CONCUR.2021.15. hal-03480241

HAL Id: hal-03480241
https://inria.hal.science/hal-03480241
Submitted on 14 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms

Nathalie Bertrand
Univ Rennes, Inria, CNRS, IRISA, France

Bastien Thomas
Univ Rennes, Inria, CNRS, IRISA, France

Josef Widder
Informal Systems, Austria

Abstract
Distributed algorithms typically run over arbitrary many processes and may involve unboundedly many rounds, making the automated verification of their correctness challenging. Building on domain theory, we introduce a framework that abstracts infinite-state distributed systems that represent distributed algorithms into finite-state guard automata. The soundness of the approach corresponds to the Scott-continuity of the abstraction, which relies on the assumption that the distributed algorithms are layered. Guard automata thus enable the verification of safety and liveness properties of distributed algorithms.

1 Introduction
Under the umbrella of parameterized verification, the verification of systems formed of an arbitrary number of agents executing the same code, has attracted quite some attention in the recent years, see for instance [18, 9]. Application examples range from distributed algorithms (e.g., for clock synchronization [28] or robot coordination [27]), cache-coherence protocols [25, 1], to chemical or biological systems [10]. In all cases, the systems are designed to operate correctly independently of the number of agents.

More specifically, distributed algorithms are central to various emblematic applications, including telecommunications, scientific computing, and Blockchain. Automatically proving the correctness of distributed algorithms is a particularly relevant, as stated by Lamport: “Model-checking algorithms prior to submitting them for publication should become the norm” [22]. The task, that the verification community has started to address, is quite challenging, since it aims at validating at once all instances of the algorithm for arbitrarily many processes.

Distributed algorithms with threshold guards are omni-present in solutions for consensus and agreement problems. Typically, these guards also are parameterized, e.g., if the number of processes in a distributed system is \( n \), then it is natural to require that certain actions are taken only if a majority of processes is ready to do so; this results in a parameterized threshold expression of \( n/2 \). Due to Blockchain and other current applications these kinds of distributed algorithm enjoy recent attention from the algorithm design community as well as the verification community. the algorithm design community has been studying them for a
Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms

long time, (see e.g., [11]) and typically provides hand-written proofs based on mathematical models without formal semantics.

For computer-aided verification the first challenge is to develop appropriate modeling formalisms that maintain all behaviors of the original algorithms on the one hand, and on the other hand are abstract and succinct to allow for efficient verification. Several approaches towards efficient verification have recently been proposed.

The threshold automata framework [20] targets asynchronous distributed algorithms with threshold guards and reductions (similar to [23, 17]) have been used to show that SMT-based bounded model checking is complete [19]. Later this framework was generalized and generalizations were analyzed regarding decidability [21], and complexity [5]. The current paper also targets threshold distributed algorithms, yet eventually provides an even coarser abstraction to represent their behaviors, thus reducing the overall verification complexity. Moreover, the semantics of distributed algorithms and the soundness of the abstraction rely on domain theory concepts, thus providing a solid mathematical framework to our work. Last but not least, our approach can handle infinite behaviours, in contrast to the threshold automata framework.

The logical fragment of the IVy toolset has also been shown to allow to model threshold guards by axiomising their semantics as quorum systems [7]. For instance, the reason for waiting for quorums of more than $n/2$ messages is that any two such quorums must intersect at one sender. IVy allows to express these quorum axioms and reduce verification to decidable fragments. Similar intuitions underlie verification results in the heard-of model (HO model) [13]. This computational model for distributed algorithms already targets a high level of abstractions that are sound for communication closed distributed algorithms [12]. Here a consensus logic was introduced in [16] that could be used for deductive verification and cut-off results where provided in [24] that reduce the parameterized verification problem to small finite instances. Compared to this line of work, the distributed algorithms we target share some similarities with these round-based communication closed models. Recently, a threshold automata framework for round-based algorithms was introduced that also uses a small counterexample property for verification in [29]. In contrast, we use domain theory, and particularly Scott continuity to be able to reason on infinite behaviors and thus to capture algorithms that do not necessarily terminate.

Other less related verification frameworks also target distributed algorithms with quite different techniques such as event B [26], array systems [4] or logic and automata theory [3].

Contributions

Using basic domain theory concepts, we provide a rigorous framework to model and verify (asynchronous) distributed algorithms. Our methodology applies to distributed algorithms that are structured in layers (that can be seen as a fine-grain notion of rounds), and may consist of countably many layers, thus capturing round-based distributed algorithms (with no a priori bound on the number of rounds).

- In Section 2, we define partially ordered transition systems, which serve to express the semantics our models.
- Section 3 introduces the low-level model of layered distributed systems to represent threshold based distributed algorithms. The state-space of layered distributed systems being infinite (and even not necessarily finitely representable), we provide several abstraction steps, up to a so-called guard abstraction. The soundness of each step is justified by the Scott-continuity of the corresponding abstraction. Some steps are also complete, and thus do not introduce spurious behaviors.
Finally, towards practical verification, we define in Section 4 the guard automaton, a finite-state abstraction of (cyclic) layered distributed systems. It overapproximates the set of infinite behaviors of distributed algorithms, and thus enabling the verification of safety as well as liveness properties. Its construction can be automated with the help of an SMT solver, paving the way to the automated verification of round-based threshold distributed algorithms.

2 A Fistful of Domain Theory

2.1 Mathematical Preliminaries

This section presents mathematical notions as well as notations that are used throughout the paper. In particular, it introduces partially ordered sets and Scott topology. The interested reader is referred to [2] for an thorough introduction to domain theory.

Sets and multisets. A multiset over a set $X$ is an element of $\mathbb{N}^X$. Addition and inclusion over multisets are defined in a natural way. For $\xi, \xi' \in \mathbb{N}^X$ two multisets, $\xi + \xi' \in \mathbb{N}^X$ is the multiset such that for every $x \in X$, $(\xi + \xi')(x) = \xi(x) + \xi'(x)$. We write $\xi \in \xi'$ if for every $x \in X$, $\xi(x) \leq \xi'(x)$. Standard sets can be seen as special cases of multisets with the canonical bijection between the set of subsets of $X$ ($2^X$) and the set of functions from $X$ to $\{0, 1\}$.

Sequences. For $X$ a set and $n \in \mathbb{N}$ a natural number, a sequence of elements of $X$ of length $n$ is some $u \in X^{[0,\ldots,n-1]}$. Its length is $|u| = n$ and for $i < n$, $u(i) \in X$ denotes the letter at index $i$. $X^* = \bigcup_{n \in \mathbb{N}} X^{[0,\ldots,n-1]}$ (resp. $X^+ = \bigcup_{n > 0} X^{[0,\ldots,n-1]}$) denotes the set of all finite (resp. finite and non-empty) sequences of elements of $X$. Moreover, $X^\infty = X^* \cup X^+$ is the set of finite or infinite sequences of $X$. For $u \in X^*$ a finite sequence and $v \in X^\infty$ a finite or infinite sequence, we write $u \cdot v$ for the concatenation of $u$ and $v$. For $u$ and $w$ two sequences, we write $u < w$ and say that $u$ is a prefix of $w$ if either $w$ is finite and there exists $v \in X^\infty$ such that $u \cdot v = w$ or $u = w$. For $w$ a sequence and $i \leq |w|$, $w_i$ is the prefix of length $i$.

Closures and bounds for partially ordered sets. Let $(X, \sqsubseteq)$ be a partially ordered set, and $\xi \in X$. The upward-closure of $\xi$ is $\uparrow \xi = \{ x \in X \mid \exists x' \in \xi, x' \sqsubseteq x \}$, and $\xi$ is upward-closed if $\uparrow \xi = \xi$. Dually, one defines the downward-closure $\downarrow \xi$ and downward-closed sets. An element $x \in X$ is an upper-bound of $\xi$ if for any element $x' \in \xi$, $x' \sqsubseteq x$. We write $\text{ub}(\xi)$ for the set of upper-bounds of $\xi$. If it exists (it is then unique), the greatest element of $\xi$ is $x \in X$ such that $x \in \xi$ and $x \in \text{ub}(\xi)$. Dually, one defines the notion of least element by reversing the order. If it exists, the least upper bound of $\xi$ is the least element of $\text{ub}(\xi)$, and we denote it by $\bigcup \xi$. Finally $\xi$ is directed if it is non-empty and if for every two elements $x, x' \in \xi$, $\text{ub}(\{x, x'\}) \cap \xi \neq \emptyset$; intuitively, any finite subset of $\xi$ has an upper-bound in $\xi$. An interesting particular case of directed case are completely ordered sets which are called chains in this context.

Directed Complete Partially ordered sets (DCPO). A DCPO is a partially ordered set $(X, \sqsubseteq)$ such that any directed subset $\xi \subseteq X$ has a (unique) least upper bound. These partially ordered sets are particularly important in semantics of programming languages.

The Scott Topology on DCPO. Directed complete partial orders are naturally equipped with the Scott topology. A subset $\xi$ of a DCPO $(X, \sqsubseteq)$ is Scott-closed if it is downward-closed and if for any directed subset $\xi' \subseteq \xi, \bigcup \xi' \in \xi$. A subset is Scott-open if its complement in $X$ is Scott-closed. Functions that are continuous for the Scott topology are called Scott-continuous. A function $f : X \to Y$ is monotonous if for any $x, x' \in X$, if $x \sqsubseteq x'$ then $f(x) \sqsubseteq f(x')$. A Scott-continuous function is always monotonous. A function $f : X \to Y$ is Scott-continuous if and only if for any directed subset $\xi \subseteq X$, $f(\bigcup(\xi)) = \bigcup(f(\xi))$. In
this paper, a partial function \( f : X \rightarrow Y \) is called \textit{Scott-continuous} if its domain \( \text{dom}(f) \) is Scott-closed and if for any directed subset \( \xi \subset \text{dom}(f) \), \( f(\bigcup \xi) = \bigcup f(\xi) \).

### 2.2 Partially Ordered Transition Systems

Building on domain theory, this section introduces a generic model for distributed transition systems, that will capture the semantics of distributed algorithms. An ordering naturally appears on sets of sent messages – that can only grow – and the asynchrony requires the order to be partial only.

▶ **Definition 1.** A partially ordered transition system (POTS) is a tuple \( \mathcal{O} = (X, \preceq, A) \) where:

- \( (X, \preceq) \) forms a DCPO.
- \( A \) is a set of partial functions, called actions, from \( X \) to itself and such that for every \( a \in A \) and every \( x \in \text{dom}(a) \), \( x \preceq a(x) \).

▶ **Definition 2.** A schedule is a (finite or infinite) sequence of actions: \( \sigma = (a_t)_{t \in T} \), with \( T \in \mathbb{N} \). A schedule \( \sigma = (a_t)_{t \in T} \) is applicable at \( x \in X \) if there exists a sequence \( (x_t)_{t \in T+1} \) with \( x_0 = x \), and for every \( t < T \), \( x_t \in \text{dom}(a_t) \) and \( a_t(x_t) = x_{t+1} \). In this case, we write \( \text{configs}(x, \sigma) \) for the sequence \( (x_t)_{t \in T+1} \), and \( x \cdot \sigma \) for \( \bigcup \{ x_t \mid t < T+1 \} \).

The above definition uses the convention that \( \infty + 1 = \infty \). Note that if \( \sigma \) is applicable at \( x \), then the sequence \( (x_t)_{t \in T+1} \) is unique. Moreover, the least upper bound \( \bigcup \{ x_t \mid t < T+1 \} \) exists because for any \( t < T \), \( x_t \in x_{t+1} \) and \( \{ x_t \mid t < T+1 \} \) is therefore a chain. When \( \sigma = (a_t)_{t \in T} \) is finite, \( x \cdot \sigma = x \cdot a_0 \cdots a_{T-1} \) denotes the last element of the monotonous sequence \( \text{configs}(x, \sigma) \). In particular, for \( a \in A \) and \( x \in \text{dom}(a) \), \( x \cdot a = a(x) \). When \( \sigma_t \in A^t \) is defined as the prefix of length \( t \) of \( \sigma \), \( x_t = x \cdot \sigma_t \) and it follows: \( x \cdot \sigma = \bigcup \{ x \cdot \sigma_t \mid t < T, t \in \mathbb{N} \} \).

The following lemma will be useful throughout the paper:

▶ **Lemma 3.** For \( x \in X \), the set \( \text{App}(x) \) of schedules applicable at \( x \) is Scott-closed for the prefix ordering and the function: \( \left[ x \cdot _{-} \right] : \text{App}(x) \rightarrow X \) is Scott-continuous.

▶ **Definition 4.** An abstraction between POTS \( \mathcal{O} = (X, \preceq, A) \) and \( \mathcal{O'} = (X', \preceq, A') \) consists of:

- a set abstraction \( \text{ab}_X : X \rightarrow X' \) which is a Scott-continuous function;
- a monoid abstraction \( \text{ab}_A : A \rightarrow A' \) which is a monoid morphism (with slight abuse of notation, \( \text{ab}_A \) also denotes its Scott-continuous extension \( \overline{A} \rightarrow \overline{A'} \));
- both such that for every \( a \in A \) and every \( x \in \text{dom}(a) \), \( \text{ab}_A(a) \in A' \) is applicable at \( \text{ab}_X(x) \in X' \) and \( \text{ab}_X(x \cdot a) = \text{ab}_X(x) \cdot \text{ab}_A(a) \).

The last condition of the definition of abstraction translates into the commutativity of the diagram in Figure 1a. The soundness of the abstraction for any (possibly infinite) schedule is stated in the following proposition and illustrated on Figure 1b.

▶ **Proposition 5.** Let \( (\text{ab}_X, \text{ab}_A) \) be an abstraction between \( \mathcal{O} = (X, \preceq, A) \) and \( \mathcal{O'} = (X', \preceq, A') \), \( x \in X \) be an element, and \( \sigma \in \overline{A} \) a schedule. If \( \sigma \) is applicable at \( x \), then \( \text{ab}_A(\sigma) \) is applicable at \( \text{ab}_X(x) \) and \( \text{ab}_X(x \cdot \sigma) = \text{ab}_X(x) \cdot \text{ab}_A(\sigma) \).

The proof of this proposition is by transfinite induction on the length of schedules: showing that the result holds for finite schedules is easy, and continuity arguments (such as Lemma 3) are then used to extend to infinite schedules.
3 Layered Distributed Systems and their Abstractions

This section introduces a low-level model for distributed algorithms, whose semantics will be expressed as a POTS. The model is structured in layers, thus restricting the application to algorithms with a specific shape. However, many distributed algorithms from the literature fall in this class, and minor modifications of other algorithms make them amenable to our techniques. The restriction to layered models is used several times in the theoretical developments that follow.

3.1 Layered Distributed Transition Systems

This section introduces Layered Distributed Transition Systems (LDTSs) as a model for distributed algorithms, such as the Phase King algorithm [8]. A simplified version of the algorithm is provided in Algorithm 1. This algorithm operates in rounds, each consisting of three steps:

- Broadcast a message \((\ell, m)\) to all process where \(\ell\) is the round index (line 3)
- Receive the messages \((\ell, \_\_)\) sent in this round (line 4)
- Update the process variables according to the received messages (lines 5 to 12)

In general, such a series of three instructions, indexed by \(\ell \in \mathbb{N}\), is called a layer and it refines the classical notion of rounds: for instance, in Ben-Or’s consensus algorithm [6], each round comprises two layers. Note that layers are assumed to be communication-closed [17, 14]: the update instruction at layer \(\ell\) only depends on received messages from the same layer.

Distributed algorithms run over a finite set of processes, and at every point in time, the local state of a process is defined by the valuation of its local variables. In this paper, the contents of a sent message is not particularly relevant as it can be deduced from the local state of its sender. Therefore, the communications can be encoded by guards that prevent a process from taking a transition if a condition on the state of other processes is not met. Formally, the syntax of layered distributed transition systems is as follows:

> **Definition 6.** A layered distributed transition system (LDTS) is a tuple \(\mathcal{D} = (P, S, \text{guard})\) where:

- \(P\) is a finite set of processes
- \(S\) is a set of states partitioned in layers: \(S = \bigcup_{\ell \in \mathbb{N}} S_\ell\).
  For \(\bot\) a new element, set \(S^\bot = S \cup \{\bot\}\) and for \(\ell \in \mathbb{N}\), \(S_\ell^\bot = S_\ell \cup \{\bot\}\).

\(\Box\) Figure 1 \((ab_X, ab_A)\) forms an abstraction between the POTS \((X, \leq, A)\) and \((X', \leq, A')\).
The set $S^\ell$ is partially ordered with $s \leq s'$ if $s = 1$, or $s = s'$. 

\[ \text{guard} : S^2 \to 2^{[P \to S^\ell]} \] associates to each pair of states a guard.

Additionally, the following layered hypothesis is imposed: 
For $\ell \in \mathbb{N}$, $s \in S_\ell$ and $s' \in S$, $\text{guard}(s, s') \in 2^{[P \to S^\ell]}$, and if $s' \notin S_{\ell+1}$, then $\text{guard}(s, s') = \emptyset$.

Intuitively, for $\ell \in \mathbb{N}$, $S_\ell$ is the set of states a process can be in at layer $\ell$, and $\bot$ is used to represent that a process has not reached that layer yet. Although trivial, the ordering on $S^\ell$ shows sufficient to represent the semantics of distributed algorithms. Moreover, the guards correspond to a condition on messages received from other processes. Having $x \in \text{guard}(s, s')$ with $x(p) = \bot$ means that there are no conditions on the messages received from process $p$, so that a process in state $s$ can go to $s'$ even if it has not received any message from $p$.

To define the semantics of LDTS, recall that the system \textit{a priori} runs fully asynchronously, so that processes may be in different layers\(^1\). However, messages may be received by processes even if the sender has later reached a layer. This means that the state of each process at each layer should be recorded in the semantics of a LDTS. An agglomeration of local states is called a configuration. A full configuration additionally stores the messages received by each process, as formalized below:

\begin{definition}
Let $D = (P, S, \text{guard})$ be an LDTS. A full configuration of $D$ is a pair $c^\ell = (\text{state}(c^\ell), \text{received}(c^\ell))$ where

- $\text{state}(c^\ell) : P \to S^\ell$ is such that for every $p \in P$ and $\ell \in \mathbb{N}$
  - if $\ell < \lfloor \text{state}(c^\ell)(p) \rfloor$, then $\text{state}(c^\ell)(p)(\ell) \in S_\ell$ and the latter is the state of $p$ in $\ell$;
  - if $\ell \geq \lfloor \text{state}(c^\ell)(p) \rfloor$, then $\text{state}(c^\ell)(p)(\ell) = \bot$.

- $\text{received}(c^\ell) : P \to P \to \mathbb{N} \to S^\ell$ such that for every $p \in P$, $\text{received}(c^\ell)(p) \in \text{state}(c^\ell)$.

The set of full configurations is denoted $C^\ell$. It is partially ordered with $\subseteq$ defined by $c^\ell \subseteq c'^\ell$ if $\text{state}(c^\ell) \subseteq \text{state}(c'^\ell)$ pointwise with the prefix ordering on $S^\ell$ and $\text{received}(c^\ell) \subseteq \text{received}(c'^\ell)$ pointwise.
\end{definition}

\(^1\) Synchronous systems can also be represented by LDTS, as illustrated with the Phase King algorithm.
Note that $S^i$ is a DCPO since each of its directed subsets is finite. $O^f$ is isomorphic to the Cartesian product $[(P^f, \rightarrow) \times (S^f, <)] \times [(P^f \times N, \rightarrow) \times (S^f, \leq)]$ and is therefore a DCPO too.

At a full configuration $c_f^j \in C_f^j$, two types of actions may happen, corresponding to receptions and internal transitions. First, a process $p \in P$ may receive a message that was sent in layer $\ell \in N$ by a process $p' \in P$; this action is denoted $\text{rec}(p, \ell, p')$. Second, a process $p \in P$ may move from a state $s \in S_\ell$ to state $s' \in S_{\ell+1}$, denoted $\text{tr}(p, s, s')$. The effect of actions on full configurations is formally defined as follows:

- **Definition 8.** The set of actions of an LDTS $D = (P, S, \text{guard})$ is
  \[
  A_f^J = \{ \text{rec}(p, p', \ell) | p, p' \in P, \ell \in N \} \cup \bigcup_{\ell \in N} \{ \text{tr}(p, s, s') | p \in P, s \in S_\ell, s' \in S_{\ell+1} \}.
  \]

For $c_f^j \in C_f^j$ and $\text{rec}(p, p', \ell) \in A_f^J$, the full configuration $c_f'^j = \text{rec}(p, p', \ell) (c_f^j)$ is defined by:
- $\text{state}(c_f'^j) = \text{state}(c_f^j)$
- $\text{received}(c_f'^j)(p)(p')(\ell) = \text{state}(c_f^j)(p')(\ell)$ and $\text{received}(c_f'^j)$ equals $\text{received}(c_f^j)$ elsewhere.

For $c_f^j \in C_f^j$ and $\text{tr}(p, s, s') \in A_f^J$, writing $\ell = |\text{state}(c_f^j)(p)| - 1$, then $\text{tr}(p, s, s')$ is enabled at $c_f^j \in C_f^j$ if: $\ell \leq \infty$, $\text{state}(c_f^j)(p)(\ell) = s$ and $\text{received}(c_f^j)(p)(\_)(\ell) \in \text{guard}(s)(s')$. In this case, the full configuration $c_f'^j = \text{tr}(p, s, s')(c_f^j)$ is defined with:
- $\text{state}(c_f'^j)(p) = \text{state}(c_f^j)(p) \cdot s'$ and $\text{state}(c_f'^j)$ equals $\text{state}(c_f^j)$ elsewhere.
- $\text{received}(c_f'^j) = \text{received}(c_f^j)$

Note that the reception actions are always enabled. So defined, the semantics of an LDTS is a POTS $O_D^f = (C_f^j, \leq, A_f^J)$; in particular, the notions of schedules and abstractions apply.

- **Example 9.** Consider the Phase King algorithm run by three correct processes and a Byzantine one. The Byzantine process is not represented explicitly ($P = \{p_0, p_1, p_2\}$ only contains correct processes) but the guards of the LDTS account for the messages it may send. Also, the King is chosen at each round non-deterministically, abstracting process ids.

  A correct process in layer $\ell$ may be in one of four states $S_\ell = \{v_0, v_1, k_0, k_1\}$, where $k_x$ (resp. $v_x$) represents that the local value of $v$ is $x \in \{0, 1\}$ and that the process is currently King (resp. not King). A full configuration, say $c_f^j$, is depicted top-left of Figure 2. The sequence states process $p_0$ went through so far is $\text{state}(c_f^j)(p_0) = v_0 \cdot k_1 \cdot v_1$. Also, $\text{received}(c_f^j)(p_0)(p_2)(0) = v_1$ represents that process $p_0$ received the message that process $p_2$ was in state $v_1$ at layer 0. In contrast, $p_0$ does not know the state of $p_2$ at layer 2 (represented by a blank space instead of 1 for commodity). Thus, in $c_f^j$, the message sent by process $p_2$ at layer 2 has yet to be received by $p_0$. The action $\text{rec}(p_0, p_2, 2)$ corresponding to this reception is therefore enabled at $c_f^j$. The resulting configuration $c_f^j * \text{rec}(p_0, p_2, 2)$ would be identical to $c_f^j$ except for $\text{received}(c_f^j * \text{rec}(p_0, p_2, 2))(p_0)(p_2)(2) = \text{state}(c_f^j)(p_2)(2) = v_1$ instead of 1. The reception $\text{rec}(p_0, p_1, 2)$ can also happen at $c_f^j * \text{rec}(p_0, p_2, 2)$. The resulting configuration $c_f'^j = c_f^j * \text{rec}(p_0, p_2, 2) * \text{rec}(p_0, p_1, 2)$ coincides with $c_f^j$ except for

\[
\begin{align*}
\text{received}(c_f'^j)(p_0) &= p_1 : v_1 \cdot v_1 \cdot k_1 \\
\text{received}(c_f'^j)(p_2) &= p_1 : v_0 \cdot v_1 \cdot v_1
\end{align*}
\]

Now $p_0$ has received more than $\frac{n}{2} + t$ messages in $\{v_1, k_1\}$ so that it updates its value to 1 in the next round. Therefore, the action $\text{tr}(p_0, v_1, v_1)$ is enabled at $c_f'^j$ and the configuration $c_f'^j * \text{tr}(p_0, v_1, v_1)$ is equal to $c_f^j$ except for $\text{state}(c_f'^j * \text{tr}(p_0, v_1, v_1)) = v_0 \cdot k_1 \cdot v_1 \cdot v_1$. 


3.2 Abstracting Received Messages

The partially ordered transition system $O_D$ is fine-grained and rather complex to analyze, therefore the aim of the rest of this section is to define simpler POTS, that preserve or overapproximate the semantics of $O_D$. The successive steps are represented in Figure 2.

<table>
<thead>
<tr>
<th>Full Configuration</th>
<th>Succinct Abstraction</th>
<th>Succinct Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>state : $C^f \rightarrow C^*$ Prop. 10</td>
<td>$p_0$</td>
</tr>
<tr>
<td>received($p_0$)</td>
<td>$p_0$</td>
<td>$k_1$</td>
</tr>
<tr>
<td>$v_0$</td>
<td>$k_1$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$k_1$</td>
<td>$v_0$</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_0 &gt; 0$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$k_1 &gt; 0$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$v_1 &gt; 0$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$2(v_0 + k_0 + f) &gt; n + 2t$</td>
<td>$T$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$2(v_1 + k_0 + f) &gt; n + 2t$</td>
<td>$T$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$2(v_1 + k_1 + f) &gt; n + 2t$</td>
<td>$\cdots$</td>
<td>$\cdots$</td>
</tr>
<tr>
<td>$v_0 + k_0 + v_1 + k_1 + f \geq n$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

| Guard Abstraction eval$_C : C \rightarrow 2^C$ Prop. 21 |

Counter Abstraction count : $C^* \rightarrow C$ Prop. 17

<table>
<thead>
<tr>
<th>Counter Configuration, $n = 4, t = 1, f = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0 : 1$</td>
</tr>
<tr>
<td>$k_0 : 0$</td>
</tr>
<tr>
<td>$v_1 : 2$</td>
</tr>
<tr>
<td>$k_1 : 0$</td>
</tr>
</tbody>
</table>

Figure 2: An illustration of the successive abstractions.

The information of messages received by each process is used to check enabledness of transitions. However, the received messages necessarily form a subset of the sent messages. Using the notion of abstraction, this section proves that received messages can be forgotten without losing any information. Instead, it suffices to require the existence of a subset of sent messages that would enable a transition. Changing views from received messages to sent ones is often implicit [21, 20] and without restrictions it may introduce spurious counter-examples (see Example 13). By imposing that each message appears in at most one guard, the layering hypothesis guarantees that the abstraction is complete (Theorem 12). This abstraction is then used to provide a characterization of reachable configurations (Theorem 15), including those reachable via an infinite schedule.

A succinct configuration is an element of $C^* = P \rightarrow S^*$. For $c_0 \in C^*$, $p \in P$, $\ell < |c_0(p)|$ and $s \in S$, $c_0(p)(\ell) = s$ means that process $p$ is/was in state $s$ at layer $\ell$. As before, if $\ell \geq |c_0(p)|$, then $c_0(p)(\ell) = \perp$, representing that process $p$ has not reached layer $\ell$ yet. So-defined, the projection $\text{state} : C^f \rightarrow C^*$ abstracts $C^f$ into $C^*$, so that the reception actions become useless. The set of succinct actions is then $A^* = \bigcup_{p \in P} \{|[p : s \rightarrow s']| p \in P, s \in S_\ell, s' \in S_{\ell+1}\}$ and the monoid morphism $\text{simpl} : A^f \rightarrow A^{f*}$ is defined by ignoring reception actions. Formally:

- for $\text{rec}(p, p', \ell) \in A^f$, $\text{simpl}(\text{rec}(p, p', \ell)) = \varepsilon$;
- for $\text{tr}(p, s, s') \in A^f$, $\text{simpl}(\text{tr}(p, s, s')) = \{p : s \rightarrow s'\}$.

One can define enabledness of a succinct action, and its effect. For a succinct configuration $c_0 \in C^*$ and a succinct action $[p : s \rightarrow s'] \in A^*$, writing $\ell = |c_0(p)| - 1$, then $[p : s \rightarrow s']$ is enabled at $c_0$ if $\ell < \infty$, $c_0(p)(\ell) = s$ and $c_0(\perp)(\ell) \in \text{guard}(s)(s')$. In this case, $([p : s \rightarrow s'](c_0))(p) = c_0(p) \cdot s'$ and $([p : s \rightarrow s'](c_0)) = c_0 \cdot s'$ for any other process.
The first two conditions of enabledness are analogous to the case of the full semantics (see Definition 8). The last condition however replaces the guard of the edge with its upper state such that:

when a process receives messages enabling a transition, no earlier transition required these.

**Example 13.** Consider the succinct configuration \( c^s \) in the top right of Figure 2. It is obtained by applying \( \text{state} \) to the full configuration \( c^f \) on the left. In Example 9, the full schedule \( \sigma^f = \text{rec}(p_0,p_2,2) \cdot \text{rec}(p_0,p_1,2) \cdot \text{tr}(p_0,v_1,v_1) \) is shown to be applicable at \( c^f \). Therefore, Proposition 10 implies that \( \text{simp}(\sigma^f) = [p_0 : v_1 \rightarrow v_1] \) is applicable at \( c^s \).

Propositions 10 and 5 entail that the succinct abstraction is sound in the sense that it does not remove any existing behavior, and properties that hold on every execution of the succinct model also hold on the full semantics. However, in general, abstractions are not complete and they may introduce new behaviors (for instance, schedules without any reception actions may be applicable in the simplification but not in the full model). Nevertheless, the succinct abstraction is complete: there always exists an applicable full schedule corresponding to each applicable succinct schedule.

**Theorem 12.** Let \( \sigma^s \in \overline{A^*} \) be a succinct schedule applicable at an initial configuration \( c^s \in C^s \). Then, there exists a full schedule \( \sigma^f \in \overline{A'^*} \) applicable at a full configuration \( c^f \in C^f \) such that: \( \text{state}(c^f) = c^s \), \( \text{simp}(\sigma^f) = \sigma^s \), and \( \text{state}(c^f \cdot \sigma^f) = c^s \cdot \sigma^s \).

To prove Theorem 12 one transforms each action \([p : s \rightarrow s']\) into a finite schedule of the form \( (\text{rec}(p,p_0,t)) \cdot \text{tr}(p,s,s') \), carefully choosing the receptions to ensure that the last transition is enabled. To do so, the difficulties are twofold. First, the full schedule \( (\text{rec}(p,p_0,t)) \cdot \text{tr}(p,s,s') \) not only depends on \([p : s \rightarrow s']\), but also on the current configuration. Therefore one cannot define a trivial abstraction. Second, this method requires a way to control the buffers of received messages throughout the schedule. Indeed, one should avoid that a process receives too many messages to take a transition, as ‘un-receiving’ messages in impossible. This is where the layered structure comes into play, and ensures that when a process receives messages enabling a transition, no earlier transition required these.

**Example 13.** As explained, the layering assumption is crucial in Theorem 12. Consider the non layered distributed transition system with four states \( a, b, c, x \), and two processes \( p, p' \). Let \( c^f \) be the initial full configuration with \( \text{state}(c^f)(p) = a \) and \( \text{state}(c^f)(p') = x \). Intuitively, in this counterexample, the guards are set such that the first transition \( \text{tr}(p,a,b) \) is enabled only if \( \text{received}(c^f)(p)(p') = x \) while the next transition \( \text{tr}(p,b,c) \) requires \( \text{received}(c^f)(p)(p') = 1 \neq x \). Process \( p \) would thus have to ‘forget’ that it received a message from \( p' \) in order to take the second transition, which is impossible in the full semantics.

In contrast, the succinct semantics does not record whether \( p \) has already received the message from \( p' \) when approaching the second transition. The succinct schedule \([p : a \rightarrow b] \cdot [p : b \rightarrow c] \) is therefore applicable at \( \text{state}(c^f) \) which would contradict Theorem 12 for unlayered distributed transition systems. Imposing that each message appears at most in one guard along the execution of a process, the layered hypothesis prevents this type of counterexamples.

The advantage of the succinct semantics over the full one is that the guards can only become true during an execution. This monotony property, combined with the layered
Guard Automata for the Verification of Safety and Liveness of Distributed Algorithms

hypothesis, entail the possibility to check that a configuration is reachable \textit{a posteriori}, simply by verifying that the guards of the transitions that are taken are verified in the last configuration. In particular, this avoids building explicitly the schedule at all intermediate configurations. This is formally stated in the following definition and theorem.

\begin{itemize}
  \item \textbf{Definition 14.} A succinct configuration \(c^* \in C^*\) is coherent if for any \(p \in P\) and \(\ell \in \mathbb{N}\), if \(c^*(p)(\ell) = s \neq \perp\) and \(c^*(p)(\ell + 1) = s' \neq \perp\), then \(c^*(\_)(\ell) \in \text{guard}(s, s')\).
  \item \textbf{Theorem 15.} Let \(c^*, c'^* \in C^*\) be two succinct configurations such that \(c^*\) is coherent. Then the following statements are equivalent:
    \begin{itemize}
      \item \(c^* \subseteq c'^*\) and \(c'^*\) is coherent.
      \item There exists a (possibly infinite) schedule \(\sigma^* \in A^*\) applicable at \(c^*\) such that \(c^* \ast \sigma^* = c'^*\).
    \end{itemize}
\end{itemize}

\subsection{Counter Abstraction}

The theory presented so far dealt with a fixed set \(P\) of processes. As an advantage, the guards of the edges could be any condition on the set of received messages, but as a drawback, it is impossible to represent \textit{parameterised} systems where the number of processes is not fixed. To remedy this downside, this section introduces \textit{layered threshold automata} (LTA). While this model is syntactically similar to threshold automata [20], its semantics in terms of a POTS is novel. Natural abstractions between the semantics of LDTS and LTA can then be presented, proving that LTA form a faithful representation of distributed algorithms, in contrast to unrestricted threshold automata.

\begin{itemize}
  \item \textbf{Definition 16.} A \textit{Layered Threshold Automaton} (LTA) is a tuple \(T = (R, S, \text{guard})\) where:
    \begin{itemize}
      \item \(R\) is a set of parameters
      \item \(S\) is a set of states partitioned into layers: \(S = \bigcup_{i=0}^{\infty} S_i\), with \(S_0\) the set of initial states.
      \item \(\text{guard} : S^2 \to \text{PA}(S \cup R)\) associates a guard, in Presburger arithmetic over free variables in \(S \cup R\), to each pair of states. The layered hypothesis assumes that for \(\ell \in \mathbb{N}\), \(s \in S_\ell\), and \(s' \in S\), \(\text{guard}(s, s') \in \text{PA}(S_\ell \cup R)\) and if \(s' \notin S_{\ell+1}\), \(\text{guard}(s, s') = \text{false}\).
    \end{itemize}

The guards are monotonous, i.e. for any guard \(g \in \text{guard}(S^2)\), for any valuation \(\rho \in \mathbb{N}^R\), \(\kappa, \kappa' \in \mathbb{N}^S\), if \(\kappa \leq \kappa'\) when ordered pointwise and if \(\rho \cdot \kappa = g\), then \(\rho \cdot \kappa' = g\) as well.

The set of parameters \(R\) typically includes the number \(n\) of processes and an upper bound \(t\) on the number of faulty processes. Intuitively, the guards represent the conditions on sent messages for taking the corresponding transition. The monotony assumption therefore requires that guards in the algorithms concern received messages only, which may be any subset of the sent messages.

In the remainder of this section, \(T = (R, S, \text{guard})\) is a fixed LTA. A configuration \(c\) of \(T\) is defined by:
    \begin{itemize}
      \item a parameter valuation \(\text{param}(c) : R \to \mathbb{N}\) that remains constant during an execution;
      \item a counting mapping \(\kappa(c) : S \to \mathbb{N}\) where \(\kappa(c)(s) = k\) means that \(k\) processes have visited the state \(s\);
      \item flow counters \(\text{flow}(c) : (\bigcup_{\ell \in \mathbb{N}} S_\ell \times S_{\ell+1}) \to \mathbb{N}\) where \(\text{flow}(c)(s, s') = k\) means that \(k\) processes moved from \(s\) to \(s'\).
    \end{itemize}

Moreover, processes that leave a state must have entered it, therefore, configurations should also verify the following flow conditions:
    \begin{itemize}
      \item \textbf{in:} for every \(\ell \in \mathbb{N} \setminus \{0\}\) and every \(s \in S_\ell\), \(\sum_{s' \in S_{\ell-1}} \text{flow}(c)(s', s) = \kappa(c)(s)\)
      \item \textbf{out:} for every \(\ell \in \mathbb{N}\) and every \(s \in S_\ell\), \(\sum_{s' \in S_{\ell+1}} \text{flow}(c)(s, s') \leq \kappa(c)(s)\).
    \end{itemize}
The set $C$ of all configurations is equipped with the natural order $\sqsubseteq$ defined by $c \sqsubseteq c'$ if $\text{param}(c) = \text{param}(c')$, $\kappa(c) \leq \kappa(c')$ and $\text{flow}(c) \leq \text{flow}(c')$.

An action over $C$ is an element of $A = \bigcup_{\ell \in \mathbb{N}} A_\ell$ where for $\ell \in \mathbb{N}$, $A_\ell = \{(s \rightarrow s') | s \in S_\ell, s' \in S_{\ell+1}\}$. For $c \in C$, an action $[s \rightarrow s'] \in A_\ell$ is enabled at $c$ if:

- $\sum_{c \in S_{\ell+1}} \text{flow}(c)(s, s') < \kappa(c)(s)$, and
- $\text{param}(c), \kappa(c) = \text{guard}(s, s')$, written $c = \text{guard}(s, s')$ for short.

In so, the successor configuration $[s \rightarrow s'](c) = c' \in C$ is defined by:

- $\text{param}(c') = \text{param}(c)$
- $\text{flow}(c') = \text{flow}(c) + \mathbb{I}_{(s,s')}$ where $\mathbb{I}_{(s,s')}(s, s') = 1$ and $\mathbb{I}_{(s,s')}(c) = 0$ elsewhere.
- $\kappa(c') = \kappa(c) + \mathbb{I}_s$ where $\mathbb{I}_s(s') = 1$ and $\mathbb{I}_s(s') = 0$ elsewhere.

One can easily check that configuration $c'$ verifies the flow conditions.

The semantics of the LTA $T$ is defined as the POTS $O_T = (C, \sqsubseteq, A)$.

For $\rho \in \mathbb{N}^R$, the set of configurations that have $\rho$ as parameters and $n$ processes initially is $C_\rho = \{c \in C | \text{param}(c) = \rho, \text{ and } \sum_{s \in S} \kappa(c)(s) = \rho(n)\}$. Let $O_T^\rho = (C_\rho, \sqsubseteq, A)$ denote the POTS restricted to these configurations.

There is a strong link between LTA and LDTS. More precisely, fix a valuation $\rho \in \mathbb{N}^R$. Consider $P_\rho$ as a set of $\rho(n)$ processes, and the LCTS $D_\rho = (P_\rho, S, \text{guard}_\rho)$ where the function $\text{guard}_\rho : \mathbb{N} \times S_\ell \times S_{\ell+1} \rightarrow 2^{(P_\rho \setminus S_\ell)}$ is defined for every $\ell \in \mathbb{N}$, $s \in S_\ell$ and $s' \in S_{\ell+1}$ by:

$$\text{guard}_\rho(s, s') = \{x \in P \rightarrow S_\ell | C_\rho, \{s \mapsto |x^{-1}(\{s\})|\} = \text{guard}(s, s')\}.$$

Let $C_\rho^\ast = P_\rho \rightarrow S_\ell^\ast$ denote the set of succinct configurations of $D_\rho$. Consider $c^\ast \in C_\rho^\ast$ and define $\text{count}_{C_\rho^\ast}(c^\ast) \in C_\rho$ with:

- $\text{param}(\text{count}_{C_\rho^\ast}(c^\ast)) = \rho$
- for $\ell \in \mathbb{N}$ and $s \in S_\ell$: $\kappa(\text{count}_{C_\rho^\ast}(c^\ast))(s) = |\{p \in P_\rho | c^\ast(p)(\ell) = s\}|
- For $\ell \in \mathbb{N}$, $s \in S_\ell$ and $s' \in S_{\ell+1}$:

$$\text{flow}(\text{count}_{C_\rho^\ast}(c^\ast))(s, s') = \left\{\begin{array}{ll}
  \{p \in P_\rho | c^\ast(p)(\ell + 1) = s' \}
\end{array}\right\}.$$

Let $A^\ast_\rho = \bigcup_{\ell \in \mathbb{N}} \{(p : s \rightarrow s') | p \in P_\rho, s \in S_\ell, s' \in S_{\ell+1}\}$ denotes the set of succinct actions of $D_\rho$. Define a monoid morphism $\text{count}_{A^\ast_\rho} : A^\ast_\rho \rightarrow A^\ast$ such that for $[p : s \rightarrow s'] \in A^\ast_\rho$, $\text{count}_{A^\ast_\rho}((\text{tr}(p, s, s')) = [s \rightarrow s']$. So defined:

- **Proposition 17.** The mappings $\text{count}_{C_\rho^\ast} : C_\rho^\ast \rightarrow C_\rho$ and $\text{count}_{A^\ast_\rho} : A^\ast_\rho \rightarrow A^\ast$ define an abstraction from the POTS $(C_\rho^\ast, \sqsubseteq, A^\ast)$ to the counter POTS $(C_\rho^\ast, \sqsubseteq, A)$. 

- **Definition 18.** Configuration $c \in C$ is said counter coherent when for every $\ell \in \mathbb{N}$, $s \in S_\ell$ and $s' \in S_{\ell+1}$, if $\text{flow}(c)((s, s')) > 0$, then $c = \text{guard}(s, s')$.

- **Theorem 19.** Let $c, c' \in C_\rho$ be two configurations such that $c$ is counter coherent. Then the following statements are equivalent:

  - $c \sqsubseteq c'$ and $c'$ is counter coherent;
  - There exists a (possibly infinite) schedule $\sigma \in \mathbb{N}$ applicable at $c$ such that $c \ast \sigma = c'$. 

\[\text{CONCUR 2021}\]
The flow conditions and the counter coherence can easily be encoded as a set of linear arithmetic formulas that do not depend on the number of processes. In particular, if the LTA is finite, then the resulting set of equations is finite as well, making the reachability problem decidable in this case (for initial and target states represented by linear arithmetic formulas). This can be used to verify not only safety properties, but also liveness properties as configurations represent potentially infinite behaviors and contain information about the whole execution. Theorem 19 differs from the threshold automata approach [20] because a schedule does not need to be explicitly built. In particular, the layering assumption implies that the order in which guards become true is irrelevant, which simplifies a lot the SMT queries. More importantly, our approach applies to infinite automata where methods based on bounding the diameter of the transition system have little chance of succeeding.

![Figure 3 Two threshold automata for the reliable broadcast algorithm [11].](image)

▶ Example 20. Theorem 19 heavily relies on the layered hypothesis. To see that, consider the non-layered model of Figure 3a. Let \( c \) be a configuration with \( \text{flow}(c)(v_0, v_1) > 0 \). Then the counter coherence would require that \( c \vDash v_1 \geq t + 1 - f \), however, this last condition may only hold because the transition was taken in the first place, resulting in spurious configurations. This can be fixed by tweaking the model in order to make it layered as seen on Figure 3b.

### 3.4 Guard Abstraction

Consider an LTA \( T = (R, S, \text{guard}) \). Even when \( S \) is finite, its configuration set \( C \) is infinite as the number of processes \( n \) is unbounded. When \( S \) is infinite, then \( C \) is infinite in two dimensions: it consists of infinitely many variables that may take infinitely many values. The guard abstraction presented here aims at partitioning these values into finitely many classes. The resulting model will however remain infinite, if \( S \) is.

Consider a set \( G \subset \text{PA}(S \cup R) \) of monotonous guards, that is, every \( g \in G \) is a linear arithmetic formulas with free variables in \( S \cup R \) such that for \( \rho \in \mathbb{N}^R \) and \( \kappa, \kappa' \in \mathbb{N}^S \), if \( \kappa \leq \kappa' \) pointwise and if \( \rho, \kappa \vDash g \), then \( \rho, \kappa' \vDash g \) as well.

Intuitively, the guard abstraction only records the valuations of the guards, not the number of processes in each state. For this idea to succeed, the valuations of the guards must converge during an execution, which is guaranteed by the following proposition.

▶ Proposition 21. The mapping \( \text{eval}_G : (C, \vDash) \to (2^G, \subseteq) \) defined by \( \text{eval}_G(c) = \{ g \in G | c \vDash g \} \) is Scott-continuous.

### 4 Guard Automata towards Practical Implementation

While Theorem 19 suffices to verify finite LTA through the counter abstraction, it falls short at capturing infinite models that arise for instance from round-based algorithms. This section introduces guard automata as a finite-state abstraction which is sound, yet, unsurprisingly, not complete in general and may introduce spurious counterexamples.
4.1 Cyclic LTA

Towards algorithmic considerations and practical implementations, the rest of the paper focuses on round-based distributed algorithms, which can be captured by cyclic LTA. Intuitively, a cyclic LTA is used to model an LTA that repeats a finite series of layers indefinitely. For $k \in \mathbb{N}_{\geq 0}$, a $k$-cyclic LTA ($k$-CLTA) is a tuple $\mathcal{T}^c = (R, S^c, \text{guard}^c)$ where:

- $R$ is a finite set of parameters.
- $S^c$ is a finite set of states partitioned into $k$ layers $S^c = S^c_0 \cup \cdots \cup S^c_{k-1}$.
- $\text{guard}^c : S^c \rightarrow \mathbb{PA}(R \cup S^c)$ is a finite set of guards such that for $\ell < k$, $s^c \in S^c_\ell$ and $s^{\ell + 1} \in S^c$, $\text{guard}^c(s^c, s^{\ell + 1}) \in \mathbb{PA}(R \cup S^c)$ and if $s^{\ell + 1} \notin S^c_{\ell+1 \mod k}$, then $\text{guard}^c(s^c, s^{\ell + 1}) = \text{false}$.

Unfolding a $k$-CLTA yields an infinite-state acyclic LTA $\text{unfold}(R, S^c, \text{guard}^c)$. Formally, $\text{unfold}(R, S^c, \text{guard}^c) = (R, S, \text{guard})$ with:

- $S = \{(s^c, \ell) \mid \ell \in \mathbb{N}, s^c \in S^c_{\ell \mod k}\}$
- For $\ell \in \mathbb{N}$, $s^c \in S^c_{\ell \mod k}$ and $s^{\ell + 1} \in S^c_{\ell + 1 \mod k}$, $\text{guard}((s^c, \ell), (s^{\ell + 1}, \ell + 1)) = \text{guard}^c(s^c, s^{\ell + 1})[s^{\ell + 1} \leftarrow (s^{\ell + 1}, \ell)$ for $s^{\ell + 1} \in S^c_{\ell \mod k}]$ meaning that any free variable $s^{\ell + 1}$ in $S^c$ that appears in $\text{guard}^c(s^c, s^{\ell + 1})$ gets replaced with $(s^{\ell + 1}, \ell)$. In any other case, $\text{guard}$ is false.

4.2 Guard Automaton

From the guard abstraction, one can construct a finite-state automaton that represents the set of reachable configurations of a cyclic LTA.

Let $\mathcal{T}^c = (R, S^c, \text{guard}^c)$ be a $k$-CLTA equipped with a finite set of guards expressed in Presburger arithmetic: $G^c = \bigcup_{\ell<k} G^c_{\ell}$ such that for $\ell < k$, $G^c_{\ell} \in \mathbb{PA}(S^c_{\ell} \cup R)$. In practice, $G^c$ will include all guards appearing in the LTA, as well as the events that need to be observed.

A CLTA can be unfolded into an infinite-state LTA, by concatenating copies of $\mathcal{T}^c$. In order for the guard abstraction to be formally defined, copies of the guards in $G^c$ for each new layer are required. For $\ell \in \mathbb{N}$ a layer index and $g^c \in G^c_{\ell \mod k}$ a guard, $\text{unfold}^G_{\ell}(g^c) = g^c[s^c \leftarrow (s^c, \ell)]$ for $s^c \in S^c_{\ell \mod k}$ denotes the guard obtained by replacing every free occurrence of a variable $s^c \in S^c_{\ell \mod k}$ in $g^c$ by $(s^c, \ell)$. The converse folding operation is defined by: $\text{fold}^G_{\ell}(g) = g[f(s^c, \ell) \leftarrow s^c]$ for $s^c \in S^c_{\ell \mod k}$. Finally, $G^c_{\ell} = \text{unfold}^G_{\ell}(G^c_{\ell \mod k})$ is the set of guards at layer $\ell$ and $G = \bigcup_{\ell<k} G^c_{\ell}$ the set of all guards.

The guard abstraction maps every configuration of $\text{unfold}(\mathcal{T}^c)$ to a set of guards that hold in that configuration. Formally, $\text{eval}^G_{\ell} : C \rightarrow 2^{G^c_{\ell}}$. A set of guards $\gamma \in 2^{G^c}$ can be represented with the sequence $\gamma_0\gamma_1\ldots$ where for $\ell \in \mathbb{N}$, $\gamma_{\ell} = \gamma \cap G^c_{\ell}$. $\text{fold}^G_{\ell}(\gamma)$ then denotes the sequence $\text{fold}^G_{\ell}(\gamma_0) \cdot \text{fold}^G_{\ell}(\gamma_1) \cdot \ldots$ in $(2^{G^c_{\ell}})^\omega$ and $\text{unfold}^G_{\ell}$ is the converse operation that applies $\text{fold}^G_{\ell}$ to the elements of layer $\ell$ in the sequence. Doing so, a configuration $c \in C$ defines a (possibly infinite) word $\gamma_0\gamma_1\ldots$ over the finite alphabet $\Sigma = \bigcup_{\ell<k} 2^{G^c_{\ell}}$ as represented in Figure 4.

![Figure 4](image-url) From a configuration to a word over the finite alphabet of the guard automaton.
For \( \ell < k \) a layer index, \( \gamma^e \in 2^{G^e} \) and \( \gamma^{ef} \in 2^{G_{\ell+1} \mod k} \) guard valuations of layer \( \ell \) and the next layer, one can use an SMT solver to check whether \( \gamma^{ef} \) is a successor \( \gamma^e \). Precisely, the SMT query asks for the existence of \( x, y, e, f \in \mathbb{N}^{S^e} \), \( y \in \mathbb{N}^{S^{ef} \mod k} \) and \( e \in \mathbb{N}^{S^e \times S^{ef} \mod k} \) such that the valuation of guards (1), flow condition (2) and counter coherence (3) are verified.

\[
\begin{align*}
\mathcal{x} &\equiv \bigwedge_{g^e \in \mathcal{G}^e} g^e &\land \bigwedge_{g^{ef} \in \mathcal{G}^{ef}} \neg g^{ef} &\land \bigwedge_{g^{ef} \in \mathcal{G}^{ef}} \neg g^{ef} \\
\mathcal{y} &\equiv \bigwedge_{g^e \in \mathcal{G}^e} g^e &\land \bigwedge_{g^{ef} \in \mathcal{G}^{ef}} \neg g^{ef} &\land \bigwedge_{g^{ef} \in \mathcal{G}^{ef}} \neg g^{ef}
\end{align*}
\]

\[\begin{align*}
e, x &\equiv \bigwedge_{s^e \in S^e_0} s^e \land \bigwedge_{s^{ef} \in S^{ef} \mod k} \left[s^e, s^{ef}\right] &> 0 &\implies \text{guard}^e(s^e, s^{ef})
\end{align*}\]

The guard automaton is a finite automaton whose language overapproximates the set of reachable configurations. It bears similarities with de Bruijn graphs [15] used e.g. in bioinformatics. If \( E_\ell \subset 2^{G^e \times 2^{G^{ef} \mod k}} \) denotes the set of all pairs \( \gamma^e, \gamma^{ef} \) that verify conditions (1) and (3), one can build the set \( E = \bigcup_{\ell \leq k} E_\ell \).

**Definition 22.** The guard automaton of \( T^c \) is \( \mathcal{G}_A(T^c) = (\Sigma, E, 2^{G^e}, \text{src}, \text{dest}, \text{label}) \) where:

- \( \Sigma \) is both the alphabet and the set of states.
- \( 2^{G^e} \subset \Sigma \) is the set of initial states.
- \( E \subset \Sigma^2 \) defined above is the set of edges, equipped with \( \text{src} : E \to \Sigma \) (resp. \( \text{dest} : E \to \Sigma \)) that defines the source state (resp. destination state) of every edge.
- \( \text{label} : E \to \Sigma \) associates a label to each edge defined by \( \text{label}(\gamma^e, \gamma^{ef}) = \gamma^e \).

An infinite run \((e_\ell)_{\ell=0}^\infty \) of the guard automaton defines a word \( \text{word}((e_\ell)_{\ell=0}^\infty) = \text{label}(e_0) \cdot \text{label}(e_1) \cdot \ldots \cdot \), and \( \mathcal{L}(\mathcal{G}_A(T^c)) \subset \Sigma^* \) denotes the language of \( \mathcal{G}_A(T^c) \).

**Example 23.** Algorithm 1 can be described by the following CLTA with \( k = 1 \). The parameters are \( R = \{ n, t, f \} \) where \( f \) denotes the actual number of Byzantine faults. States are \( S^c = \{ v_0, k_0, v_1, k_1 \} \). The guards here only depend on the next value of \( v \). For instance:

- \( \text{guard}(-, v_0) = (v_0 + k_0 + v_1 + k_1 + f = n) \)
- \( \land \left( (2(v_0 + k_0 + f) > n + 2t) \lor (2(v_0 + 2k_0 \leq n + 2t) \land (2v_1 + 2k_1 \leq n + 2t) \land (k_1 = 0)) \right) \).

Also, \( \text{guard}(-, k_0) = \text{guard}(-, v_0) \) and \( \text{guard}(-, v_1) = \text{guard}(-, k_1) \) is defined symmetrically.

A configuration \( c \) of the unfolded LTA is depicted bottom-right of Figure 2, where the array contains the valuation \( \kappa(c) \) and the arrows represent the flow. For example \( \kappa(c)(v_0, 1) = 2, \text{flow}(c)((v_0, 0), (k_1, 1)) = 1 \) and \( \text{flow}(c)((v_0, 0), (v_0, 1)) = 0 \).

The guard abstraction transforms \( c \) into the guard configuration bottom-left of Figure 2. Here, we chose the set of guards \( G^e \) to consist of \( s > 0 \) for each \( s \in S^e \) and of the guards of the LTA. The alphabet \( \Sigma \) contains e.g., \( \{ T \cdot T \cdots T \} \). SMT queries determine whether two letters may appear successively, in order to build the guard automaton. For instance, according to the first two layers of \( \text{eval}_G(c), (T \cdot T \cdots T) \) can be followed by \( (T \cdot TT \cdots T) \).

There will therefore be a transition between these two states in the guard automaton.

**Theorem 24.** Let \( c \in C \) be a configuration of \( \text{unfold}(T^c) \) and \( \text{eval}_G(c) \in 2^G \) its guard abstraction. If \( c \) is counter-coherent, then \( \text{fold}^G(\text{eval}_G(c)) \in \mathcal{L}(\mathcal{G}_A(T^c)) \).

By soundness of the guard automaton construction, a property which holds on configurations that correspond to runs of \( \mathcal{G}_A(T^c) \) also holds on the configurations of \( \text{unfold}(T^c) \).
A simple verification procedure thus consists in checking that $\mathcal{L}(\text{GA}_C(T^c))$ is included in a given language of correct configurations. At a first glance, it might seem that only safety properties can be checked. However, the guard automaton also represents configurations reachable by infinite schedules, making the verification of liveness properties feasible.

Example 25. For presentation purposes, Algorithm 1 is an overly simplified version of the Phase King algorithm [8]. The latter can be faithfully encoded by the 2-CLTA $T^c$ of Figure 5, where the updated value when there is no clear majority is not the king’s value, but rather the majority of the values received by the king. Each round consists of two layers of communication, a first in which each process broadcasts its value, and a second in which the king broadcasts what it thinks is the majority. The set of guards at the first layer is $G_0^c = \{v_0 > 0, v_1 > 0\}$ and at the second layer $G_1^c$ consists of $k_0,0 + k_1,0 > 0$, $k_0,1 + k_1,1 > 0$, $p_0 + k_0,0 + k_0,1 > 0$, $p_1 + k_1,0 + k_1,1 > 0$, $2(k_0,0 + k_1,1 + p_0 + f) > n + 2t$ and $2(k_1,0 + k_1,1 + p_1 + f) > n + 2t$.

Restricting to valuations with $\sum_{s \in S} s + f = n$ (fairness) and $k_0,0 + k_0,1 + k_1,0 + k_1,1 \leq 1$ (at most one king), the resulting guard automaton has 3 states in even layers and 11 in odd layers. Writing [formula] for the set of letters in $2^{G^c}$ for which [formula] holds, one can show:

$$\mathcal{L}(\text{GA}_C(T^c)) \subseteq \exists \neg (k_0,0 + k_1,0 > 0) \land \neg (k_0,1 + k_1,1 > 0)$$

(4)

$$\cup \Sigma^\pi [\neg (k_0,0 + k_1,0 > 0) \lor (k_0,1 + k_1,1 > 0)] [\neg (p_0 + k_0,0 + k_0,1 > 0)]$$

(5)

$$\cup \Sigma^\pi [\neg (k_0,0 + k_1,0 > 0) \lor (k_0,1 + k_1,1 > 0)] [\neg (p_1 + k_1,0 + k_1,1 > 0)]$$

(6)

Therefore, either every chosen king is Byzantine (4), or all processes agree on a value after a non-Byzantine king is chosen (5 or 6).

In general, although it is sound, the guard automaton construction is not complete: the language may contain words that correspond to no configuration of the LTA. As usual for incomplete methods, heuristics can be used to remove some spurious counterexamples.

5 Conclusion

This paper presented a methodology, based on domain theory, to represent and analyze distributed algorithms. Infinite-state models are abstracted into finite-state guard automata, on which one can check safety and liveness properties.

Optimizing and benchmarking the guard automaton implementation is on our current agenda to demonstrate the applicability of our methodology to standard distributed algorithms. A more long-term research objective is to build on the current contribution to develop a rigorous framework for the verification of randomized distributed algorithms.
References


