Liveness in Broadcast Networks

Peter Chini, Roland Meyer, and Prakash Saivasan

TU Braunschweig
{p.chini, roland.meyer, p.saivasan}@tu-bs.de

Abstract. We study two liveness verification problems for broadcast networks, a system model of identical clients communicating via message passing. The first problem is liveness verification. It asks whether there is a computation such that one of the clients visits a final state infinitely often. The complexity of the problem has been open since 2010 when it was shown to be P-hard and solvable in EXPSPACE. We close the gap by a polynomial-time algorithm. The algorithm relies on a characterization of live computations in terms of paths in a suitable graph, combined with a fixed-point iteration to efficiently check the existence of such paths. The second problem is fair liveness verification. It asks for a computation where all participating clients visit a final state infinitely often. We adjust the algorithm to also solve fair liveness in polynomial time.

1 Introduction

Broadcast networks are a general model for capturing the behavior of client crowds [14] (or client networks). These consist of an arbitrary number of identical clients that run concurrently and communicate via a mechanism like shared memory or message passing. Such networks appear in various applications. For instance, in distributed algorithms, when a large group of clients needs to elect a unique leader. Or in cache-coherence protocols when coherence among shared data needs to be guaranteed. Proving that a client network satisfies a certain specification or correctness condition seems a hard task: Errors may occur only for a particular number of clients which is not known a priori. Hence, automated verification needs to reason about all those numbers at the same time.

Despite this difficulty, verification of client networks has led to a successful line of research [24,20,15,16,12,6,9,8,7,4,2]. Many of the considered verification tasks, especially for broadcast networks, admit algorithms that are surprisingly efficient. However, there is a gap in the complexity for checking liveness: Repeated reachability in broadcast networks is known to be decidable in EXPSPACE [9] but the only known lower bound is P-hardness [8]. We close this gap by showing that liveness can be checked in polynomial time.

Broadcast networks were introduced to verify ad hoc networks [9,24]. They consist of an arbitrary number of identical finite-state automata, communicating via passing messages. We refer to these automata as clients. Using the name client twice is intentional: An automaton makes visible the interaction of a single concrete client with its environment, providing transitions for sending and
receiving messages. The interplay between several communicating clients is captured by the transition relation of the broadcast network. When a client sends message \(a\) by taking a corresponding send transition, at the same time, there is a number of clients that receive \(a\) by taking a corresponding receive transition. A sequence of such synchronized transitions is called a computation.

In a broadcast network, the number of receiving clients can change during a computation. This means that a sending client can only transport its message to clients that are within its range. The idea for this model stems from mobile clients that might be in the range of a sender for a certain time \([24,9]\). Usually, the model is referred to as reconfigurable broadcast networks, since the communication topology is reconfigured after each send. Since we only consider reconfigurable broadcast networks, we drop the term “reconfigurable”.

Different variants of safety verification in broadcast networks have been studied in earlier works. Technically, two variants of reachability were considered, differing in the number of clients that need to reach an unsafe/final state: In the coverability problem, the question is whether at least one participating client can reach such a state. The problem is known to be solvable in polynomial time \([8]\).

In the synchronization problem, all clients need to visit a final state at the same time. At first sight, the problem seems harder than coverability, but is also solvable in polynomial time by a fixed-point algorithm \([18]\). Both problems remain in \(\mathbb{P}\) even if the reconfigurable topology is slightly restricted \([2]\). Usually, such constraints lead to undecidability results \([9,2]\).

The focus of our work is on liveness verification. Instead of only describing unsafe states, liveness properties are capable of expressing good events that should keep happening during a computation. An example are conditions of the form “Each request in a system is followed by a response”. Liveness verification amounts to checking repeated reachability, testing whether a final state can be reached infinitely many times.

In the setting of broadcast networks, repeated reachability was studied before in \([9]\). The authors consider the repeated coverability problem, a generalization of the coverability problem where at least one client needs to visit a final state infinitely many times. The problem is known to be solvable in \(\text{EXPS}^{\text{SPACE}}\) by a reduction to repeated coverability in Petri Nets \([17,13]\). However, the only known lower bound is its \(\mathbb{P}\)-hardness \([8]\). Closing this gap remained an open problem.

We consider two variants of repeated reachability. The first is called liveness verification problem. It is equivalent to repeated coverability considered in \([9]\). The second variant is the fair liveness verification problem. It asks for a computation where all clients that run infinitely often meet a final state infinitely many times. We contribute an algorithm, solving both problems in polynomial time. Hereby closing the gap for repeated coverability. Furthermore, we observe a phenomenon that has been discovered before in the related leader-contributor model \([20,14,12]\): Safety and liveness verification are of the same complexity.

Our results can also be applied for efficient model checking against LTL \([21]\) in broadcast networks. If the LTL formula is given as an automaton \([25]\), we construct the product with the clients and run our algorithm.
At the heart of our algorithm is a fixed-point iteration that terminates in polynomial time. It is inspired by ideas presented in [18]. To obtain the iteration, we rely on an efficient representation of computations. To this end, we first prove a characterization of witnesses for liveness verification in terms of paths in a graph. Since the graph is of exponential size, we cannot immediately apply a path finding algorithm. Instead, we show that a path exists if and only if there is one in a particular normal form. Paths in normal form can then be found efficiently by the fixed-point iteration.

Related Work We already discussed related literature considering safety and liveness verification of broadcast networks with reconfigurable communication topology. There are various studies where the topology obeys certain restrictions. In [2], decidability and undecidability results for reachability problems were proven for a topology that can only change within a certain distance each reconfiguration. The case when communication is fixed along a given graph was studied in [1]. Topologies that approximate the notion of bounded diameter were considered in [11]. The authors of [15] studied safety and liveness verification under the assumption that a sent message is always received by all clients. Networks with communication failures were considered in [11]. Reachability in probabilistic broadcast networks was studied in [3]. In [4], a variant of broadcast networks was considered where the clients follow a certain local strategy.

The leader-contributor model is closely related to broadcast networks. Instead of only clients, it has a fixed leader and an arbitrary number of identical contributors that communicate via a shared memory. The model was introduced in [20]. The case when leader and all contributors are finite-state automata was considered in [16] and the corresponding reachability problem was proven to be NP-complete. In [7], the authors provided a fine-grained view for the reachability problem proving it fixed-parameter tractable. Liveness verification for this model was studied in [12]. The authors show that repeated reachability is NP-complete. This proves that the complexities of safety and liveness verification coincide.

Client networks with shared memory and randomized scheduler were considered in [6]. A communication mechanism for client networks called rendez-vous was studied in [19]. For a survey of parameterized verification we refer to [5].

2 Broadcast Networks

We introduce the model of broadcast networks of interest in this paper. Our presentation avoids an explicit characterization of the communication topology in terms of graphs. A broadcast network is a concurrent system consisting of an arbitrary but finite number of identical clients that communicate by passing messages to each other. Formally, it is a pair $\mathcal{N} = (D, P)$. The domain $D$ is a finite set of messages that can be used for communication. A message $a \in D$ can either be sent, $!a$, or received, $?a$. The set $\text{Ops}(D) = \{!a, ?a \mid a \in D\}$ captures the communication operations a client can perform. For modeling the identical clients, we abstract away the internal behavior and focus on the communication
with others via \( Ops(D) \). With this, the clients are given in the form of a finite state automaton \( P = (Q, I, \delta) \), where \( Q \) is a finite set of states, \( I \subseteq Q \) is a set of initial states, and \( \delta \subseteq Q \times Ops(D) \times Q \) is the transition relation. We extend \( \delta \) to words in \( Ops(D)^* \) and write \( q \xrightarrow{w} q' \) instead of \( (q, w, q') \in \delta \).

During a communication phase in \( \mathcal{N} \), one client sends a message that is received by a number of other clients. This induces a change of the current state in each client participating in the communication. We use \( \text{configurations} \) to display the current states of the clients. A configuration is a tuple \( c = (q_1, \ldots, q_k) \in Q^k \), \( k \in \mathbb{N} \). We use \( \text{Set}(c) \) to denote the set of client states occurring in \( c \). To access the components of \( c \), we use \( c[i] = q_i \). As the number of clients in the system is arbitrary but fixed, we define the set of all configurations to be \( CF = \bigcup_{k \in \mathbb{N}} Q^k \).

The set of \( \text{initial configurations} \) is given by \( CF_0 = \bigcup_{k \in \mathbb{N}} I^k \). The communication is modeled by a transition relation among configurations. Let \( c' = (q'_1, \ldots, q'_k) \) be another configuration with \( k \) clients and \( a \in D \) a message. We have a transition \( c \xrightarrow{a} \mathcal{N} \ c' \) if the following conditions hold: (1) there is a sender, an \( i \in [1..k] \) such that \( q_i \xrightarrow{a} q'_i \), (2) there is a number of receivers, a set \( R \subseteq [1..k] \setminus \{i\} \) such that \( q_j \xrightarrow{a} q'_j \) for each \( j \in R \), and (3) all other clients stay idle, for all \( j \notin R \cup \{i\} \) we have \( q_j = q'_j \). We use \( \text{idx}(c) = R \cup \{i\} \) to denote the indices of clients that contributed to the transition. We extend the transition relation to words \( w \in D^* \) and write \( c \xrightarrow{w} \mathcal{N} \ c' \). Such a sequence of consecutive transitions is called a \( \text{computation} \) of \( \mathcal{N} \). Note that all configurations appearing in a computation have the same number of clients. We write \( c \xrightarrow{*} \mathcal{N} \ c' \) if there is a word \( w \in D^* \) with \( c \xrightarrow{w} \mathcal{N} \ c' \). If \( |w| \geq 1 \), we also use \( c \xrightarrow{+} \mathcal{N} \ c' \). Where appropriate, we skip \( \mathcal{N} \) in the index. We are interested in infinite computations, infinite sequences \( \pi = c_0 \rightarrow c_1 \rightarrow \ldots \) of consecutive transitions. We call such a computation \( \text{initialized} \), if \( c_0 \in CF_0 \). Moreover, we use \( \text{Inf}(\pi) = \{i \in \mathbb{N} \mid \exists^\infty j : i \in \text{idx}(c_j \rightarrow c_{j+1})\} \) to denote the set of clients that participate in the computation infinitely often. We let \( \text{Fin}(\pi) = \{i \in \mathbb{N} \mid \exists^\infty j : c_j[i] \in F\} \) represent the set of clients that visit final states infinitely often.

### 3 Liveness

We consider the \( \text{liveness verification problem} \) for broadcast networks. Given a broadcast network \( \mathcal{N} = (D, P) \) with \( P = (Q, I, \delta) \) and a set of final states \( F \subseteq Q \), the problem asks whether there is an infinite initialized computation \( \pi \) in which at least one client visits a final state from \( F \) infinitely often, \( \text{Fin}(\pi) \neq \emptyset \).

**Input:** A broadcast network \( \mathcal{N} = (D, P) \) and final states \( F \subseteq Q \).

**Question:** Is there an initialized computation \( \pi \) with \( \text{Fin}(\pi) \neq \emptyset \)?

The liveness verification problem was introduced in \cite{9} as repeated coverability. In this section, we show the following:

**Theorem 1.** The liveness verification problem is \( \mathcal{P} \)-complete.
The definition of the finite graph underlying our algorithm is inspired by the powerset construction for the determinization of finite state automata \([23]\). The nodes keep track of sets of states \(S\) that a client may be in. Different from finite state automata, however, we have a parameterized system. So for each state \(s \in S\), there is not only one client in that state but arbitrarily (but finitely) many. As a consequence, a transition from \(s\) to \(s'\) may have two effects. Some of the clients in \(s\) change their state to \(s'\) while others stay in \(s\). In that case, the set of states is updated to \(S' = S \cup \{s'\}\). Alternatively, all clients may change their state to \(s'\), in which case we get \(S' = (S \setminus \{s\}) \cup \{s'\}\).

Formally, the graph of interest is \(G = (V, \rightarrow_G)\). The vertices are tuples of sets of states, \(V = \bigcup_{k \leq |Q|} \mathcal{P}(Q)^k\). The parameter \(k\) will become clear in a moment. For defining the edges, we need some more notation. For \(S \subseteq Q\) and \(a \in D\), let

\[
\text{post}_{\gamma_a}(S) = \{r' \in Q \mid \exists r \in S : r \xrightarrow{\gamma_a} r'\}
\]

denote the set of successors of \(S\) under transitions receiving \(a\). The set of states in \(S\) where receives of \(a\) are enabled is denoted by

\[
\text{enabled}_{\gamma_a}(S) = \{r \in S \mid \text{post}_{\gamma_a}(\{r\}) \neq \emptyset\}.
\]

There is a directed edge \(V_1 \rightarrow_G V_2\) from vertex \(V_1 = (S_1, \ldots, S_k)\) to vertex \(V_2 = (S'_1, \ldots, S'_k)\) if the following three conditions are satisfied: (1) There is an index \(j \in [1..k]\), states \(s \in S_j\) and \(s' \in S'_j\), and an element \(a\) from the domain \(D\) such that \(s \xrightarrow{\gamma_a} s'\) is a send transition. (2) For each \(i \in [1..k]\) there are sets of states \(\text{Gen}_i \subseteq \text{post}_{\gamma_a}(S_i)\) and \(\text{Kill}_i \subseteq \text{enabled}_{\gamma_a}(S_i)\) such that

\[
S'_i = \begin{cases} 
(S_i \setminus \text{Kill}_i) \cup \text{Gen}_i, & \text{for } i \neq j, \\
(U_j \setminus \text{Kill}_j) \cup \text{Gen}_j \cup \{s'\}, & \text{for } i = j
\end{cases}
\]
where $U_j$ is either $S_j$ or $S_j \setminus \{s\}$. (3) For each index $i \in [1..k]$ and state $q \in \text{Kill}_i$, the intersection $\text{post}_{\text{in}}(q) \cap Gen_i$ is non-empty.

Intuitively, an edge in $G$ mimics a transition in the broadcast network without making explicit the configurations. Condition (1) requires a sender, a component $j$ that is capable of sending a message $a$. Clients receiving this message are represented by (2). The set $\text{Gen}_i$ consists of those states that are reached by clients performing a corresponding receive transition. These states are added to $S_i$. As mentioned above, states can get killed. If, during a receive transition, all clients decide to move to the target state, the original state will not be present anymore. We capture those states in the set $\text{Kill}_i$ and remove them from $S_i$. Condition (3) is needed to guarantee that each killed state is replaced by a target state. Note that for component $j$ we add $s'$ due to the send transition. Moreover, we need to distinguish whether state $s$ gets killed or not.

The following lemma relates a cycle in the constructed graph with a cyclic computation of the form $c \rightarrow^+ c$. It is crucial for our result.

**Lemma 3.** There is a cycle $\{(s_1), \ldots, (s_m)\} \rightarrow^+_G \{(s_1), \ldots, (s_m)\}$ in $G$ if and only if there is a configuration $c$ with $\text{Set}(c) = \{s_1, \ldots s_m\}$ and $c \rightarrow^+ c$.

The lemma explains the restriction of the nodes in the graph to $k$-tuples of sets of states, with $k \leq |Q|$. We explore the transitions for every possible state in $c$, and there are at most $|Q|$ different states that have to be considered. We have to keep the sets of states separately to make sure that, for every starting state, the corresponding clients perform a cyclic computation.

**Proof.** We first fix some notations that we use throughout the proof. Let $c \in Q^n$ be any configuration and $s \in \text{Set}(c)$. By $\text{Pos}_c(s) = \{i \in [1..n] \mid c[i] = s\}$ we denote the positions of $c$ storing state $s$. Given a second configuration $d \in Q^n$, we use the set $\text{Target}_c(s, d) = \{d[i] \mid i \in \text{Pos}_c(s)\}$ to represent those states that occur in $d$ at the positions $\text{Pos}_c(s)$. Intuitively, if there is a sequence of transitions from $c$ to $d$, these are the target states of those positions of $c$ that store $s$.

Let a computation $\pi = c \rightarrow^+ c$ with $\text{Set}(c) = \{s_1, \ldots, s_m\}$ be given. We show that there is a cycle $\{(s_1), \ldots, (s_m)\} \rightarrow^+_G \{(s_1), \ldots, (s_m)\}$ in $G$. To this end, assume $\sigma$ is of the form $\pi = c \rightarrow c_1 \rightarrow \cdots \rightarrow c_k \rightarrow c$. Since $c \rightarrow c_1$ is a proper transition in the broadcast network, there is an edge

\[
\{(s_1), \ldots, (s_m)\} \rightarrow_G (\text{Target}_c(s_1, c_1), \ldots, \text{Target}_c(s_m, c_1))
\]

in $G$ where each state $s_i$ gets replaced by the set of target states in $c_1$. Applying this argument inductively, we get a path in the graph:

\[
\{(s_1), \ldots, (s_m)\} \rightarrow_G (\text{Target}_c(s_1, c_1), \ldots, \text{Target}_c(s_m, c_1))
\rightarrow_G (\text{Target}_c(s_1, c_2), \ldots, \text{Target}_c(s_m, c_2))
\rightarrow_G \cdots
\rightarrow_G (\text{Target}_c(s, c), \ldots, \text{Target}_c(s_m, c)).
\]

Since $\text{Target}_c(s, c) = \{s_i\}$, we found the desired cycle.
For the other direction, let a cycle \( \sigma = (\{s_1\}, \ldots, \{s_m\}) \rightarrow^G_\ell (\{s_1\}, \ldots, \{s_m\}) \) be given. As required, we construct from \( \sigma \) a computation \( \pi = c \rightarrow^+ c \) in the broadcast network such that \( \text{Set}(c) = \{s_1, \ldots, s_m\} \). The difficulty in constructing \( \pi \) is to ensure that at any point in time there are enough clients in appropriate states. For instance, if a transition \( s \overset{i_a}{\rightarrow} s' \) occurs, we need to decide on how many clients move to \( s' \), meaning change their state to \( s' \). Having too few clients in \( s' \) may stall the computation at a later point: There might be a number of required sends that can only be obtained by transitions outgoing \( s' \). If there are too few clients in \( s' \), we cannot guarantee the sends. The solution is to start with enough clients in any state. With invariants we guarantee that at any point, the number of clients in the needed states suffices.

Let cycle \( \sigma \) be \( V_0 \rightarrow_G V_1 \rightarrow_G \cdots \rightarrow_G V_\ell \) with \( V_0 = V_\ell = (\{s_1\}, \ldots, \{s_m\}) \). Further, let \( V_j = (S^1_j, \ldots, S^m_j) \). We will construct the computation \( \pi \) over configurations in \( Q^n \), where \( n = n \cdot |Q|^\ell \). The idea is to have \( |Q|^\ell \) clients for each of the \( m \) components of the vertices \( V_t \) occurring in \( \sigma \). To access the clients belonging to a particular component, we split configurations in \( Q^n \) up into blocks, intervals \( I(i) = [(i-1) \cdot |Q|^\ell + 1, i \cdot |Q|^\ell] \) for each \( i \in [1,m] \). Let \( d \in Q^n \) be arbitrary. For \( i \in [1,m] \), let \( B_d(i) = \{d[t] \mid t \in I(i)\} \) be the set of states occurring in the \( i \)-th block of \( d \). Moreover, we blockwise collect clients that are currently in a particular state \( s \in Q \). Let the set \( \text{Pos}_d(i,s) = \{t \in I(i) \mid d[t] = s\} \) be those positions of \( d \) in the \( i \)-th block that store state \( s \).

We fix the configuration \( c \in Q^n \). For each component \( i \in [1,m] \), it contains \( |Q|^\ell \) copies of the state \( s_i \) in the \( i \)-th block. Formally, \( B_c(i) = \{s_i\} \). Our goal is to construct the computation \( \pi = c_0 \rightarrow^+ c_1 \rightarrow^+ \cdots \rightarrow^+ c_\ell \) with \( c_0 = c_\ell = c \) such that the following two invariants are satisfied: (1) For each \( j \in [0,\ell] \) and \( i \in [1,m] \) we have \( B_{c_j}(i) \subseteq S^j_i \). (2) For any state \( s \) in a set \( S^j_i \) we have \( |\text{Pos}_{c_j}(i,s)| \geq |Q|^{\ell-i} \). Intuitively, (1) means that during the computation \( \pi \) we visit at most those states that occur in the cycle \( \sigma \). Invariant (2) guarantees that at each configuration \( c_j \) there are enough clients available in these states.

We construct \( \pi \) inductively. The base case is given by configuration \( c_0 = c \) which satisfies invariants (1) and (2) by definition. For the induction step, assume \( c_j \) is already constructed such that (1) and (2) hold for the configuration. Our first goal is to construct a configuration \( d \) such that \( c_j \rightarrow^+ d \) and \( d \) satisfies invariant (2). In a second step we show to construct a computation \( d \rightarrow^* c_{j+1} \).

In the cycle \( \sigma \) there is an edge \( V_j \rightarrow_G V_{j+1} \). From the definition of \( \rightarrow_G \) we get a component \( t \in [1,m] \), states \( s \in S^t_j \) and \( s' \in S^t_{j+1} \), and an \( a \in D \) such that there is a send transition \( s \overset{a}{\rightarrow} s' \). Moreover, there are sets \( \text{Gen}_t \subseteq \text{post}_{\tau_a}(S^t_j) \) and \( \text{Kill}_t \subseteq \text{enabled}_{\tau_a}(S^t_j) \) such that the following equality holds:

\[
S^t_{j+1} = (U_t \setminus \text{Kill}_t) \cup \text{Gen}_t \cup \{s'\}.
\]

Here, \( U_t \) is either \( S^t_j \) or \( S^t_j \setminus \{s\} \). We focus on \( t \) and take care of other components later. We apply a case distinction for the states in \( S^t_{j+1} \).

Let \( q \) be a state in \( S^t_{j+1} \setminus \{s'\} \). If \( q \in \text{Gen}_t \), there exists a \( p \in S^t_j \) such that \( p \overset{a}{\rightarrow} q \). We apply this transition to \( |Q|^\ell-(j+1) \) many clients in the \( t \)-th block of
configuration $c_j$. If $q \in U_t \setminus \text{Kill}_t$ and $q$ not in $\text{Gen}_t$, then certainly $q \in U_t \subseteq S^{t+1}_{j+1}$. In this case, we let $|Q|^{t-(j+1)}$ many clients of block $t$ stay idle in state $q$. For state $s'$, we apply a sequence of sends. More precise, we apply the transition $s \xrightarrow{a} s'$ to $|Q|^{t-(j+1)}$ many clients in block $t$ of $c_j$. The first of these sends synchronizes with the previously described receive transitions. The other sends do not have any receivers. For components different from $t$, we apply the same procedure. Since there are only receive transitions, we also let them synchronize with the first send of $a$. This leads to a computation $\tau$

$$c_j \xrightarrow{a} d^1 \xrightarrow{a} d^2 \xrightarrow{a} \ldots \xrightarrow{a} d^{Q|^{t-(j+1)}} = d.$$ 

We argue that the computation $\tau$ is valid. There are enough clients in $c_j$ such that $\tau$ can be carried out. We again focus on component $t$, other components are similar. Let $p \in \text{Set}(c_j) = S^t_j$. Note that the equality is due to invariants (1) and (2). We count the clients of $c_j$ in state $p$ (in block $t$) that are needed to perform $\tau$. We need

$$|Q|^{t-(j+1)} \cdot |\text{post}_a(p) \cup \{p, s'\}| \leq |Q|^{t-(j+1)} \cdot |Q| = |Q|^{t-j}$$

of these clients. The set $\text{post}_a(p) \cup \{p, s'\}$ appears as a consequence of the case distinction above: There might be transitions mapping $p$ to a state in $\text{post}_a(p)$, it may happen that clients stay idle in $p$, and in the case $p = s$, we need to add $s'$ for the send transition. Since $|\text{Pos}_{c_j}(t, p)| \geq |Q|^{t-j}$ by invariant (2), we get that $\tau$ is a valid computation. Moreover, note that configuration $d$ satisfies invariant (2) for $j + 1$: For each state $q \in S^t_{j+1}$, the computation $\tau$ was constructed such that $|\text{Pos}_d(t, q)| \geq |Q|^{t-(j+1)}$.

To satisfy invariant (1), we need to erase states that are present in $d$ but not in $S^t_{j+1}$. To this end, we reconsider the set $\text{Kill}_t \subseteq \text{enabled}_a(S^t_j)$. For each state $p \in \text{Kill}_t$, we know by the definition of $\to_G$ that $\text{post}_a(p) \cap \text{Gen}_t \neq \emptyset$. Hence, there is a $q \in S^t_{j+1}$ such that $p \xrightarrow{a} q$. We apply this transition to all clients in $d$ currently in state $p$, that were not active (not mentioned) in the computation $\tau$. In case $U_t = S^t_j \setminus \{s\}$, we apply the send $s \xrightarrow{a} s'$ to all clients that are still in $s$ and were not active in $\tau$. Altogether, this leads to a computation $\eta = d \to^* c_{j+1}$.

There is a subtlety in the definition of $\eta$. There might be no send transition for the receivers to synchronize with since $s$ may not need to be erased. In this case, we synchronize the receive transitions of $\eta$ with the last send of $\tau$. This does not change the result in the end.

Computation $\eta$ substitutes the states in $\text{Kill}_t$ and state $s$, depending on $U_t$, by states in $S^t_{j+1}$. But this means that in the $t$-th block of $c_{j+1}$, there are only states of $S^t_{j+1}$ left. Hence, $B_{c_{j+1}}(t) \subseteq S^t_{j+1}$, and invariant (1) holds.

After the construction of $\pi = c \to^* c_t$, it is left to argue that $c_t = c$. But this is due to the fact that invariant (1) holds for $c_t$ and $S^t_t = \{s_1, \ldots, s_m\}$. 

The graph $G$ is of exponential size. To obtain a polynomial-time procedure, we cannot just search it for a cycle as required by Lemma 3. Instead, we now show that if such a cycle exists, then there is a cycle in a certain normal form.
Hence, it suffices to look for a normal-form cycle. As we will show, this can be done in polynomial time. We define the normal form more generally for paths.

A path is in normal form, if it takes the shape \( V_1 \rightarrow^*_G V_m \rightarrow^*_G V_n \) such that the following conditions hold. In the prefix \( V_1 \rightarrow^*_G V_m \) the sets of states increase monotonically, \( V_i \subseteq V_{i+1} \) for all \( i \in [1..m] \). Here, \( \subseteq \) denotes the componentwise inclusion. In the suffix \( V_m \rightarrow^*_G V_n \), the sets of states decrease monotonically, \( V_i \supseteq V_{i+1} \) for all \( i \in [m..n] \). The following lemma states that if there is a path in the graph, then there is also a path in normal form. The intuition is that the variants of the transitions that decrease the sets of states can be postponed towards the end of the computation.

**Lemma 4.** There is a path from \( V_1 \) to \( V_2 \) in \( G \) if and only if there is a path in normal form from \( V_1 \) to \( V_2 \).

**Proof.** If \( V_1 \rightarrow^*_G V_2 \) is a path in normal form, there is nothing to prove. For the other direction, let \( \sigma = V_1 \rightarrow^*_G V_2 \) be an arbitrary path. To get a path in normal form, we first simulate the edges of \( \sigma \) in such a way that no states are deleted. Hence, we have to respect a particular deletion order ensuring that we construct a valid path.

Let \( \sigma = U_1 \rightarrow_G U_2 \rightarrow_G \cdots \rightarrow_G U_{\ell} \) with \( U_1 = V_1 \) and \( U_{\ell} = V_2 \). Further, we fix an \( m \leq |Q| \) and we let \( U_j \in \mathcal{P}(Q)^m \). We inductively construct an increasing path \( \sigma_{\text{inc}} = U'_1 \rightarrow_G \cdots \rightarrow_G U'_{\ell} \) with \( U'_j \supseteq U_j \) by mimicking the edges of \( \sigma \).

For the base case, we set \( U'_1 = U_1 \). Now assume \( \sigma_{\text{inc}} \) has already been constructed up to vertex \( U'_j \). There is an edge \( e = U_j \rightarrow_G U_{j+1} \in \sigma \). Since \( U'_{j+1} \supseteq U_j \), we can simulate \( e \) on \( U'_j \). All states needed to execute the edge are present in \( U'_j \). Moreover, we can mimic \( e \) such that no state gets deleted. This is achieved by setting the corresponding kill sets to the empty set. Hence, we get an edge \( U'_j \rightarrow U'_{j+1} \rightarrow U'_{j+1} \supseteq U'_j \) (no deletion) and \( U'_{j+1} \supseteq U_{j+1} \) (simulation of \( e \)).

The states in \( V^*_2 = U'_2 \) that are not in \( V_2 \) are those states that were deleted along \( \sigma \). We construct a decreasing path \( \sigma_{\text{dec}} = V'_2 \rightarrow_G V_2 \), deleting all these states. To this end, let \( V'_2 = (T_1, \ldots, T_m) \) and \( V_2 = (S_1, \ldots, S_m) \). An edge in \( \sigma \) deletes sets of states in each component \( i \in [1..m] \). Hence, to mimic the deletion, we need to consider subsets of \( \text{Del} = \bigcup_{i \in [1..m]} (T_i \setminus S_i) \times \{i\} \). Note that index \( i \) in a tuple \( (s, i) \) displays the component the state \( s \) is in.

Consider the equivalence relation \( \sim \) over \( \text{Del} \) defined by \((x, i) \sim (y, t)\) if and only if the last occurrence of \( x \) in component \( i \) and \( y \) in component \( t \) in the path \( \sigma \) coincide. Intuitively, two elements are equivalent if they get deleted at the same time and do not appear again in \( \sigma \). We introduce an order on the equivalence classes: \([x, i] \prec [y, t] \) if and only if the last occurrence of \((x, i)\) was before the last occurrence of \((y, t)\). Since the order is total, we get a partition of \( \text{Del} \) into equivalence classes \( P_1, \ldots, P_n \) such that \( P_j \prec P_{j+1} \) for each \( j \in [1..n-1] \).

Now we can construct \( \sigma_{\text{dec}} = K_0 \rightarrow_G \cdots \rightarrow_G K_n \) with \( K_0 = V'_2 \) and \( K_n = V_2 \) as follows. During each edge \( K_{j+1-1} \rightarrow_G K_j \), we precisely delete the elements in \( P_j \) and do not add further states. Note that deleting \( P_j \) is due to an edge \( e = U_{j_1} \rightarrow_G U_{j_1+1} \in \sigma \). We mimic \( e \) in such a way that no state gets added, we set the corresponding gen sets to the empty set. Since we respect the order
with the deletions, the simulation of $e$ is possible: Suppose, we need a state $s$ in component $t$ to simulate $e$ but the state is not available in component $t$ of $K_{j-1}$. Then it was deleted before, $(s, t) \in P_1 \cup \cdots \cup P_{j-1}$. But this contradicts that $s$ is present in $U_j$. Hence, all the needed states are available.

Since after the last edge of $\sigma_{\text{dec}}$ we have deleted all elements from Del, we get that $K_n = V_2$. This finishes the proof. \qed

Using the normal-form result in Lemma 4 we now give a polynomial-time algorithm to check whether $\left(\{s_1\}, \ldots, \{s_m\}\right) \rightarrow_{C}^+ \left(\{s_1\}, \ldots, \{s_m\}\right)$. The idea is to mimic the monotonically increasing prefix of the computation by a suitable post-operator, the monotonically decreasing suffix by a suitable pre-operator, and intersect the two. The difficulty in computing an appropriate post operator is to ensure that the receive operations are enabled by sends leading to a state in the intersection, and similar for the pre. The solution is to use a greatest fixed-point computation. In a first Kleene iteration step, we determine the intersection, and similar for the pre. The solution is to use a greatest fixed-point computation. For the definition of the operators, consider $C = (C_1, \ldots, C_m) \in \mathcal{P}(Q)^m$ for an $m \leq \left|Q\right|$. Given a sequence of sets of states $X_1, \ldots, X_m$ where each $X_i \subseteq C_i$, we define $\operatorname{post}_C(X_1, \ldots, X_m) = (X_1', \ldots, X_m')$ with

$$X_i' = \{s' \in Q \mid \exists s \in X_i : s \xrightarrow{a}_{P[C_i]} s' \} \cup \{s' \in Q \mid \exists s_1, s_2 \in X_i : \exists s \in X_i : s \xrightarrow{a}_{P[C_i]} s_2 \land s \xrightarrow{a}_{P[C_i]} s' \}.$$  

Here, $P[C_i]$ denotes the automaton obtained from $P$ by restricting it to the states $C_i$. Similarly, we define $\operatorname{pre}_C(X_1, \ldots, X_m) = (X_1', \ldots, X_m')$ with

$$X_i' = \{s \in Q \mid \exists s' \in X_i : s \xrightarrow{a}_{P[C_i]} s' \} \cup \{s \in Q \mid \exists s_1, s_2 \in X_i : \exists s' \in X_i : s_1 \xrightarrow{a}_{P[C_i]} s_2 \land s \xrightarrow{a}_{P[C_i]} s' \}.$$  

The following lemma shows that the (reflexive) transitive closures of these relations can be computed in polynomial time.

**Lemma 5.** The closures $\operatorname{post}_C^+(X_1, \ldots, X_m)$ and $\operatorname{pre}_C^+(X_1, \ldots, X_m)$ can be computed in polynomial time.

**Proof.** Computing both closures can be done by a saturation. For $\operatorname{post}_C^+(X_1, \ldots, X_m)$, we keep $m$ sets $R_1, \ldots, R_m$, each being the post of a component. Initially, we set $R_i = X_i$. The defining equation of $X_i'$ in $\operatorname{post}_C^+(X_1, \ldots, X_m)$ gives the saturation. One just needs to substitute $X_i$ by $R_i$ and $X_i$ by $R_i$ on the right hand side. The resulting set of states is added to $R_i$. This process is
applied consecutively to each component and then repeated until the sets \( R_i \) do not change anymore, the fixed point is reached.

The saturation terminates in polynomial time. After updating \( R_i \) in each component, we either already terminated or added at least one new state to a set \( R_i \). Since there are \( m \leq |Q| \) of these sets and each one is a subset of \( Q \), we need to update the sets \( R_i \) at most \( |Q|^2 \) many times. For a single of these updates, the dominant time factor comes from finding appropriate send and receive transitions. This can be achieved in \( O(|\delta|^2) \) time.

Computing the closure \( \text{pre}^*_C(X_1, \ldots, X_m) \) is similar. One can apply the above saturation and only needs to revert the transitions in the client \( P \).

As argued above, the existence of a cycle reduces to finding a fixed point. The following lemma shows that it can be computed efficiently.

**Lemma 6.** There is a cycle \( (\{s_1\}, \ldots, \{s_m\}) \rightarrow G^+ (\{s_1\}, \ldots, \{s_m\}) \) if and only if there is a non-trivial solution to the equation

\[
C = \text{post}^+_C(\{s_1\}, \ldots, \{s_m\}) \cap \text{pre}^*_C(\{s_1\}, \ldots, \{s_m\}).
\]

Such a solution can be found in polynomial time.

**Proof.** For finding the solution in polynomial time, we use a Kleene iteration to compute the greatest fixed point. It invokes Lemma 5 as a subroutine. Every step of the Kleene iteration reduces the number of states in \( C \) by at least one, and initially there are at most \( |Q| \) entries with at most \( |Q| \) states each. Hence, we terminate after quadratically many iteration steps.

It is left to prove correctness. Let \( (\{s_1\}, \ldots, \{s_m\}) \rightarrow G^+ (\{s_1\}, \ldots, \{s_m\}) \) be a cycle in \( G \). By Lemma 4 we can assume it to be in normal form. Let \( (\{s_1\}, \ldots, \{s_m\}) \rightarrow G^+ C \) be the increasing part and \( C \rightarrow G^+ (\{s_1\}, \ldots, \{s_m\}) \) the decreasing part. Then, \( C \) is a solution to the equation.

For the other direction, let a solution \( C \) be given. Since \( C \) is contained in \( \text{post}^+_C(\{s_1\}, \ldots, \{s_m\}) \) we can construct a monotonically increasing path \( (\{s_1\}, \ldots, \{s_m\}) \rightarrow G^+ C \). Similarly, since \( C \subseteq \text{pre}^*_C(\{s_1\}, \ldots, \{s_m\}) \), we get a decreasing path \( C \rightarrow G^+ (\{s_1\}, \ldots, \{s_m\}) \). Hence, we get the desired cycle. ⊓ ⊔

What is yet open is the question on which states \( s_1 \) to \( s_m \) to perform the search for a cycle. After all, we need that the corresponding configuration is reachable from an initial configuration. The idea is to use the set of all states reachable from an initial state in the client. Note that there is a live computation if and only if there is a live computation involving all those states. Intuitively, the states that do not contribute to the cycle will not be chosen by any client. Since the states reachable from an initial state can be computed in polynomial time [8], the proof of Theorem 1 is completed.

The liveness verification problem does not take fairness into account. A client may contribute to the live computation (and help the distinguished client reach a final state) without ever making progress towards its own final state.
4 Fair Liveness

We study the *fair liveness verification problem* that strengthens the requirement on the computation sought. Given a broadcast network $\mathcal{N} = (D, P)$ with clients $P = (Q, I, \delta)$ and a set of final states $F \subseteq Q$, the problem asks whether there is an infinite initialized computation $\pi$ in which clients that send or receive messages infinitely often also visit their final states infinitely often, $\text{Inf}(\pi) \subseteq \text{Fin}(\pi)$. This requirement is also known as compassion or strong fairness [22].

**Fair Liveness Verification**

**Input:** A broadcast network $\mathcal{N} = (D, P)$ and final states $F \subseteq Q$.

**Question:** Is there an initialized computation $\pi$ with $\text{Inf}(\pi) \subseteq \text{Fin}(\pi)$?

Our algorithm is an instrumentation that turns the instance $(\mathcal{N}, F)$ of the fair liveness verification problem into an instance $(\mathcal{N}_F, Q)$ of the liveness verification problem studied in the previous section. Recall that $Q$ is the set of client states in $\mathcal{N}$. The main result is the following.

**Theorem 7.** $(\mathcal{N}, F)$ is a yes-instance of the fair liveness verification problem if and only if $(\mathcal{N}_F, Q)$ is a yes-instance of the liveness verification problem. The instrumentation is computable in linear time.

With the instrumentation, we immediately get:

**Corollary 8.** Fair liveness verification is in $\text{P}$.

To explain the instrumentation, we need the notion of a good computation, where good means the computation respects fairness. Computation $c_1 \rightarrow^+ c_n$ is good for $F$, denoted $c_1 \Rightarrow_F c_n$, if every client $i$ that makes a move during the computation, $i \in \text{idx}(c_j \rightarrow c_{j+1})$ for a $j$, also sees a final state in the computation, $c_k[i] \in F$ for some $k$. The following strengthens Lemma 2.

**Lemma 9.** There is a fair computation from $c_0$ if and only if $c_0 \rightarrow^* c \Rightarrow_F c$.

The instrumentation $\mathcal{N}_F$ is designed to detect good cycles $c \Rightarrow_F c$. The idea is to let the clients compute in phases. The original state space $Q$ is the first phase. As soon as a client participates in the computation, it moves to a second phase given by a copy $\tilde{Q}$ of $Q$. From this copy it enters a third phase $\tilde{\tilde{Q}}$ upon seeing a final state. From $\tilde{\tilde{Q}}$ it may return to $Q$.

Let the given broadcast network be $\mathcal{N} = (D, P)$ with $P = (Q, I, \delta)$. We define $\mathcal{N}_F = (D \cup \{n\}, P_F)$ with fresh symbol $n \notin D$ and instrumented client

$$P_F = (\tilde{Q}, \tilde{I}, \tilde{\delta}) \quad \text{where} \quad \tilde{Q} = Q \cup \tilde{Q} \cup \tilde{\tilde{Q}}.$$ 

For every transition $(q, a, q') \in \delta$, we have $(\tilde{q}, a, \tilde{q}')$, $(\tilde{q}, a, \tilde{q}')$, $(\tilde{q}, a, \tilde{q}') \in \tilde{\delta}$. For every final state $q \in F$ we have $(\tilde{q}, !n, \tilde{q}) \in \tilde{\delta}$. For every state $q \in Q$ we have $(\tilde{q}, !n, \tilde{q}) \in \tilde{\delta}$. Configuration $c$ admits a good cycle if and only if there is a cycle at $c$ in the instrumented broadcast network. Even more, also an initial prefix can be mimicked by computations in the third phase.
Lemma 10. $c_0 \rightarrow^\ast c \Rightarrow F c$ in $\mathcal{N}$ if and only if $\tilde{c}_0 \rightarrow^\ast c \rightarrow^+ c$ in $\mathcal{N}_F$.

We argue that the cycle can be mimicked, the reasoning for the prefix is simpler. A good cycle entails a cycle in the instrumented broadcast network. For the reverse direction, note that in $c$ all clients are in states from $Q$. As soon as a client participates in the computation, it will move to $\tilde{Q}$. To return to $Q$ the client will have to see a final state. This makes the computation good.

The proof of Theorem 7 follows from Lemma 2, Lemma 9, and Lemma 10.

References

1. P. A. Abdulla, M. F. Atig, and O. Rezine. Verification of directed acyclic ad hoc networks. In FORTE, volume 7892 of LNCS, pages 193–208. Springer, 2013.
2. A. R. Balasubramanian, N. Bertrand, and N. Markey. Parameterized verification of synchronization in constrained reconfigurable broadcast networks. In TACAS, volume 10806 of Lecture Notes in Computer Science, pages 38–54. Springer, 2018.
3. N. Bertrand, P. Fournier, and A. Sangnier. Playing with probabilities in reconfigurable broadcast networks. In FOSSACS, volume 8412 of LNCS, pages 134–148. Springer, 2014.
4. N. Bertrand, P. Fournier, and A. Sangnier. Distributed local strategies in broadcast networks. In CONCUR, volume 42 of LIPIcs, pages 44–57. Schloss Dagstuhl, 2015.
5. 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.
6. P. Bouyer, N. Markey, M. Randour, A. Sangnier, and D. Stan. Reachability in networks of register protocols under stochastic schedulers. In ICALP, volume 55 of LIPIcs, pages 106:1–106:14. Schloss Dagstuhl, 2016.
7. P. Chini, R. Meyer, and P. Saivasan. Fine-grained complexity of safety verification. In TACAS, volume 10806 of LNCS, pages 20–37. Springer, 2018.
8. G. Delzanno, A. Sangnier, R. Traverso, and G. Zavattaro. On the complexity of parameterized reachability in reconfigurable broadcast networks. In FSTTCS, volume 18 of LIPIcs, pages 289–300. Schloss Dagstuhl, 2012.
9. G. Delzanno, A. Sangnier, and G. Zavattaro. Parameterized verification of ad hoc networks. In CONCUR, volume 6269 of LNCS, pages 313–327. Springer, 2010.
10. G. Delzanno, A. Sangnier, and G. Zavattaro. On the power of cliques in the parameterized verification of ad hoc networks. In FOSSACS, volume 6604 of LNCS, pages 441–455. Springer, 2011.
11. G. Delzanno, A. Sangnier, and G. Zavattaro. Verification of ad hoc networks with node and communication failures. In FORTE, volume 7273 of LNCS, pages 235–250. Springer, 2012.
12. A. Durand-Gasselin, J. Esparza, P. Ganty, and R. Majumdar. Model checking parameterized asynchronous shared-memory systems. In CAV, volume 9206 of LNCS, pages 67–84. Springer, 2015.
13. J. Esparza. Some applications of petri nets to the analysis of parameterised systems (talk). In WISP, 2003.
14. J. Esparza. Keeping a crowd safe: On the complexity of parameterized verification (invited talk). In STACS, volume 25 of LIPIcs, pages 1–10. Schloss Dagstuhl, 2014.
15. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In LICS, pages 352–359. IEEE, 1999.
16. J. Esparza, P. Ganty, and R. Majumdar. Parameterized verification of asynchronous shared-memory systems. In CAV, volume 8044 of LNCS, pages 124–140. Springer, 2013.
17. J. Esparza and M. Nielsen. Decidability issues for petri nets - a survey. Bulletin of the EATCS, 52:244–262, 1994.
18. P. Fournier. Parameterized verification of networks of many identical processes. PhD thesis, University of Rennes 1, 2015.
19. S. M. German and A. P. Sistla. Reasoning about systems with many processes. J. ACM, 39(3):675–735, 1992.
20. M. Hague. Parameterised pushdown systems with non-atomic writes. In FSTTCS, volume 13 of LIPIcs, pages 457–468. Schloss Dagstuhl, 2011.
21. A. Pnueli. The temporal logic of programs. In FOCS, pages 46–57. IEEE, 1977.
22. A. Pnueli and Y. Sa’ar. All you need is compassion. In VMCAI, volume 4905 of Lecture Notes in Computer Science, pages 233–247. Springer, 2008.
23. M. O. Rabin and D. Scott. Finite automata and their decision problems. IBM journal of research and development, 3(2):114–125, 1959.
24. A. Singh, C. R. Ramakrishnan, and S. A. Smolka. Query-based model checking of ad hoc network protocols. In CONCUR, volume 5710 of LNCS, pages 603–619. Springer, 2009.
25. M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In LICS, pages 322–331. IEEE, 1986.