Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers

Peizun Liu[1(✉)], Thomas Wahl[1], and Akash Lal[2]

[1] Northeastern University, Boston, USA
lpzun@ccs.neu.edu
[2] Microsoft Research, Bangalore, India

Abstract. We address the problem of analyzing asynchronous event-driven programs, in which concurrent agents communicate via unbounded message queues. The safety verification problem for such programs is undecidable. We present in this paper a technique that combines *queue-bounded exploration* with a *convergence test*: if the sequence of certain abstractions of the reachable states, for increasing queue bounds k, converges, we can prove any property of the program that is preserved by the abstraction. If the abstract state space is finite, convergence is *guaranteed*; the challenge is to catch the point $k_{\max}$ where it happens. We further demonstrate how simple invariants formulated over the *concrete* domain can be used to eliminate spurious *abstract* states, which otherwise prevent the sequence from converging. We have implemented our technique for the P programming language for event-driven programs. We show experimentally that the sequence of abstractions often converges fully automatically, in hard cases with minimal designer support in the form of sequentially provable invariants, and that this happens for a value of $k_{\max}$ small enough to allow the method to succeed in practice.

1 Introduction

Asynchronous event-driven (AED) programming refers to a style of programming multi-agent applications. The agents communicate shared work via messages. Each agent waits for a message to arrive, and then processes it, possibly sending messages to other agents, in order to collectively achieve a goal. This programming style is common for distributed systems as well as low-level designs such as device drivers [11]. Getting such applications right is an arduous task, due to the inherent concurrency: the programmer must defend against all possible interleavings of messages between agents. In response to this challenge, recent years have seen multiple approaches to verifying AED-like programs, e.g. by delaying send actions, or temporarily bounding their number (to keep queue sizes small) [7,10],

Work supported by the US National Science Foundation under Grant No. 1253331, and by Microsoft Research India while hosting the second author for a sabbatical.

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 386–404, 2019.
https://doi.org/10.1007/978-3-030-25543-5_22

or by reasoning about a small number of representative execution schedules, to avoid interleaving explosion [5].

In this paper we consider the P language for AED programming [11]. A P program consists of multiple state machines running in parallel. Each machine has a local store, and a message queue through which it receives events from other machines. P allows the programmer to formulate safety specifications via a statement that asserts some predicate over the local state of a single machine. Verifying such reachability properties of course requires reasoning over global system behavior and is, for unbounded-queue P programs, undecidable [8].

The unboundedness of the reachable state space does not prevent the use of testing tools that try to explore as much of the state space as possible [3,6,11,13] in the quest for bugs. Somewhat inspired by this kind of approach, the goal of this paper is a verification technique that can (sometimes) *prove* a safety property, despite exploring only a finite fraction of that space. Our approach is as follows. Assuming that the machines' queues are the only source of unboundedness, we consider a bound k on the queue size, and exhaustively compute the reachable states R_k of the resulting finite-state problem, checking the local assertion Φ along the way. We then increase the queue bound until (an error is found, or) we reach some point $k_{\max}$ of *convergence*: a point that allows us to conclude that increasing k further is not required to prove Φ.

What kind of "convergence" are we targeting? We design a sequence $(\overline{R}_k)_{k=0}^{\infty}$ of abstractions of each reachability set over a *finite* abstract state space. Due to the monotonicity of sequence $(\overline{R}_k)_{k=0}^{\infty}$, this ensures convergence, i.e. the existence of $k_{\max}$ such that $\overline{R}_K = \overline{R}_{k_{\max}}$ for all $K \geq k_{\max}$. Provided that an abstract state satisfies Φ exactly if all its concretizations do, we have: if all abstract states in $\overline{R}_{k_{\max}}$ comply with Φ, then so do all reachable concrete states of P—we have proved the property.

We implement this strategy using an abstraction function α with a finite co-domain that leaves the local state of a machine unchanged and maintains the *first occurrence* of each event in the queue; repeat occurrences are dropped. This abstraction preserves properties over the local state and the head of the queue, i.e. the visible (to the machine) part of the state space, which is typically sufficient to express reachability properties.

The second major step in our approach is the detection of the point of convergence of $(\overline{R}_k)_{k=0}^{\infty}$: We show that, for the *best abstract transformer* $\overline{Im}$ [9,27, see Sect. 4.2], if $\overline{Im}(\overline{R}_k) \subseteq \overline{R}_k$, then $\overline{R}_K = \overline{R}_k$ for all $K \geq k$. In fact, we have a stronger result: under an easy-to-enforce condition, it suffices to consider abstract *dequeue operations*: all others, namely enqueue and local actions, never lead to abstract states in $\overline{R}_{k+1} \setminus \overline{R}_k$. The best abstract transformer for dequeue actions is efficiently implementable for a given P program.

It is of course possible that the convergence condition $\overline{Im}(\overline{R}_k) \subseteq \overline{R}_k$ never holds (the problem is undecidable). This manifests in the presence of a *spurious* abstract state in the image produced by $\overline{Im}$, i.e. one whose concretization does not contain any reachable state. Our third contribution is a technique to assist users in eliminating such states, enhancing the chances for convergence. We

have observed that spurious abstract states are often due to violations of simple *machine invariants*: invariants that do not depend on the behavior of other machines. By their nature, they can be proved using a cheap sequential analysis.

We can eliminate an abstract state (e.g. produced by $\overline{Im}$) if *all* its concretizations violate a machine invariant. In this paper, we propose a domain-specific temporal logic to express invariants over machines with event queues and, more importantly, an algorithm that decides the above *abstract queue invariant checking* problem, by reducing it efficiently to a plain model checking problem. We have used this technique to ensure the convergence in "hard" cases that otherwise defy convergence of the abstract reachable states sequence.

We have implemented our technique for the P language and empirically evaluated it on an extensive set of benchmark programs. The experimental results support the following conclusions: (i) for our benchmark programs, the sequence of abstractions often converges fully automatically, in hard cases with minimal designer support in the form of separately dischargeable invariants; (ii) almost all examples converge at a small value of $k_{\max}$; and (iii) the overhead our technique adds to the bounding technique is small: the bulk is spent on the exhaustive bounded exploration itself.

Proofs and other supporting material can be found in the Appendix of [23].

2 Overview

We illustrate the main ideas of this paper using an example in the P language. A machine in a P program consists of multiple states. Each state defines an entry code block that is executed when the machine enters the state. The state also defines handlers for each event type e that it is prepared to receive. A handler can either be on e do foo (executing foo on receiving e), or ignore e (dequeuing and dropping e). A state can also have a defer e declaration; the semantics is that a machine dequeues the first non-deferred event in its queue. As a result, a queue in a P program is not strictly FIFO. This relaxation is an important feature of P that helps programmers express their logic compactly [11]. Figure 1 shows a P program named *PiFl*, in which a Sender (eventually) floods a Receiver's queue with PING events. This queue is the only source of unboundedness in *PiFl*.

A critical property for P programs is *(bounded) responsiveness*: the receiving machine must have a handler (e.g. on, defer, ignore) for every event arriving at the queue head; otherwise the event will come as a "surprise" and crash the machine. To prove responsiveness for *PiFl*, we have to demonstrate (among others) that in state Ignore_it, the DONE event is never at the head of the Receiver's queue. We cannot perform exhaustive model checking, since the set of reachable states is infinite. Instead, we will compute a conservative abstraction of this set that is precise enough to rule out DONE events at the queue head in this state.

We first define a suitable abstraction function α that collapses repeated occurrences of events to each event's first occurrence. For instance, the queue

$$\mathcal{Q} = \text{PRIME.PRIME.PRIME.DONE.PING.PING.PING.PING} \tag{1}$$

```
1  event PRIME, DONE, PING;
2
3  machine Sender {
4    var receiver: machine;
5    start state Init {
6      entry {
7        receiver = new Receiver();
8      }
9      goto Prime_it;
10   }
11   state Prime_it {
12     entry {
13       var i:int;
14       while (i < 3) { // 3x PRIME
15         send receiver, PRIME; i = i + 1;
16       }
17       send receiver, DONE; goto Ping_it;
18     }
19   }
20   state Ping_it {
21     entry {
22       send receiver, PING; goto Ping_it;
23     }
24   }
25 }
26
27 machine Receiver {
28   start state Init {
29     defer PRIME;
30     on DONE goto Ignore_it;
31   }
32
33   state Ignore_it {
34     ignore PRIME, PING;
35   }
36 }
```

Fig. 1. *PiFl*: a Ping-Flood scenario. The Sender and the Receiver communicate via events of types PRIME, DONE, and PING. After sending some PRIME events and one DONE, the Sender floods the Receiver with PINGs. The Receiver initially defers PRIMEs. Upon receiving DONE it enters a state in which it ignores PING.

will be abstracted to $\overline{\mathcal{Q}} = \alpha(\mathcal{Q}) = \text{PRIME.DONE.PING}$. The *finite* number of possible abstract queues is $1+3+3\cdot 2+3\cdot 2\cdot 1 = 16$. The abstraction preserves the head of the queue. This and the machine state has enough information to check responsiveness.

We now generate the sequence $\overline{R}_k$ of abstractions of the reachable states sets R_k for queue size bounds $k = 0, 1, 2, \ldots$, by computing each finite set R_k, and then $\overline{R}_k$ as $\alpha(R_k)$. The obtained monotone sequence $(\overline{R}_k)_{k=0}^{\infty}$ over a finite domain will eventually converge, but we must prove that it has. This is done by applying the *best abstract transformer* $\overline{Im}$, restricted to dequeue operations (defined in Sect. 4.2), to the current set $\overline{R}_k$, and confirming that the result is contained in $\overline{R}_k$.

As it turns out, the confirmation fails for the *PiFl* program: $k = 5$ marks the first time set $\overline{R}_k$ repeats, i.e. $\overline{R}_4 = \overline{R}_5$, so we are motivated to run the convergence test. Unfortunately we find a state $\bar{s} \in \overline{Im}(\overline{R}_5) \setminus \overline{R}_5$, preventing convergence. Our approach now offers two remedies to this dilemma. One is to refine the queue abstraction. In our implementation, function α is really α_p, for a parameter p that denotes the size of the *prefix* of the queue that is kept unchanged by the abstraction. For example, for the queue from Eq. (1) we have $\alpha_4(\mathcal{Q}) = \text{PRIME.PRIME.PRIME.DONE} \mid \text{PING}$, where $\mid$ separates the prefix from the "infinite tail" of the abstract queue. This (straightforward) refinement maintains finiteness of the abstraction and increases precision, by revealing that the queue starts with three PRIME events. Re-running the analysis for the *PiFl* program with $p = 4$, at $k = 5$ we find $\overline{Im}(\overline{R}_5) \subseteq \overline{R}_5$, and the proof is complete.

The second remedy to the failed convergence test dilemma is more powerful but also less automatic. Let's revert to prefix $p = 0$ and inspect the abstract state $\bar{s} \in \overline{Im}(\overline{R}_5) \setminus \overline{R}_5$ that foils the test. We find that it features a DONE event followed by a PRIME event in the Receiver's queue. A simple static analysis of the Sender's machine in isolation shows that it permits no path from the send DONE

to the send PRIME statement. The behavior of other machines is irrelevant for this invariant; we call it a *machine invariant*. We pass the invariant to our tool via the command line using the expression

$$\mathsf{G}\,(\text{DONE} \Rightarrow \mathsf{G}\,\neg\text{PRIME}) \tag{2}$$

in a temporal-logic like notation called QuTL (Sect. 5.1), where G universally quantifies over all queue entries. Our tool includes a QuTL checker that determines that **every concretization** of $\bar{s}$ violates property (2), concluding that $\bar{s}$ is spurious and can be discarded. This turns out to be sufficient for convergence.

3 Queue-(Un)Bounded Reachability Analysis

Communicating Queue Systems. We consider P programs consisting of a fixed and known number n of machines communicating via event passing through unbounded FIFO queues.[1] For simplicity, we assume the machines are created at the start of the program; dynamic creation at a later time can be simulated by having the machine ignore all events until it receives a special creation event.

We model such a program as a *communicating queue system* (CQS). Formally, given $n \in \mathbb{N}$, a CQS P^n is a collection of n *queue automata* (QA) $P_i = (\Sigma, \mathcal{L}_i, Act_i, \Delta_i, \ell_i^I)$, $1 \leq i \leq n$. A QA consists of a finite queue alphabet Σ shared by all QA, a finite set $\mathcal{L}_i$ of local states, a finite set Act_i of action labels, a finite set $\Delta_i \subseteq \mathcal{L}_i \times (\Sigma \cup \{\varepsilon\}) \times Act_i \times \mathcal{L}_i \times (\Sigma \cup \{\varepsilon\})$ of transitions, and an initial local state $\ell_i^I \in \mathcal{L}_i$. An action label $act \in Act_i$ is of the form

- $act \in \{deq, loc\}$, denoting an action *internal* to P_i (no other QA involved) that either *deq*ueues an event (*deq*), or updates its *loc*al state (*loc*); **or**
- $act = !(e, j)$, for $e \in \Sigma$, $j \in \{1, \ldots, n\}$, denoting a *transmission*, where P_i (the *sender*) adds event e to the end of the queue of P_j (the *receiver*).

The individual QA of a CQS model machines of a P program; hence we refer to QA states as *machine states*. A transmit action is the only communication mechanism among the QA.

Semantics. A *machine state* m of a QA is of the form $(\ell, \mathcal{Q}) \in \mathcal{L} \times \Sigma^*$; state $m^I = (\ell^I, \varepsilon)$ is *initial*. We define machine transitions corresponding to internal actions as follows (transmit actions are defined later at the global level):

$$\frac{(\ell, \varepsilon) \xrightarrow{loc} (\ell', \varepsilon) \in \Delta}{(\ell, \mathcal{Q}) \rightarrow (\ell', \mathcal{Q})} \text{ for } \ell, \ell' \in \mathcal{L},\ \mathcal{Q} \in \Sigma^* \qquad \textbf{(local)}$$

$$\frac{(\ell, e) \xrightarrow{deq} (\ell', \varepsilon) \in \Delta}{(\ell, e\mathcal{Q}) \rightarrow (\ell', \mathcal{Q})} \text{ for } \ell, \ell' \in \mathcal{L},\ e \in \Sigma,\ \mathcal{Q} \in \Sigma^* \qquad \textbf{(dequeue)}$$

[1] The P language permits unbounded machine creation, a feature that we do not allow here and that is not used in any of the benchmarks we are aware of.

A *(global) state* s of a CQS is a tuple $\langle(\ell_1, \mathcal{Q}_1), \ldots, (\ell_n, \mathcal{Q}_n)\rangle$ where $(\ell_i, \mathcal{Q}_i) \in \mathcal{L}_i \times \Sigma^*$ for $i \in \{1, \ldots, n\}$. State $s^I = \langle(\ell_1^I, \varepsilon), \ldots, (\ell_n^I, \varepsilon)\rangle$ is initial. We extend the machine transition relation $\rightarrow$ to states as follows:

$$\langle(\ell_1, \mathcal{Q}_1), \ldots, (\ell_n, \mathcal{Q}_n)\rangle \rightarrow \langle(\ell_1', \mathcal{Q}_1'), \ldots, (\ell_n', \mathcal{Q}_n')\rangle$$

if there exists $i \in \{1, \ldots, n\}$ such that one of the following holds:

(internal) $(\ell_i, \mathcal{Q}_i) \rightarrow (\ell_i', \mathcal{Q}_i')$, and for all $k \in \{1, \ldots, n\} \setminus \{i\}$, $\ell_k = \ell_k'$, $\mathcal{Q}_k = \mathcal{Q}_k'$;
(transmission) there exists $j \in \{1, \ldots, n\}$ and $e \in \Sigma$ such that:

1. $(\ell_i, \varepsilon) \xrightarrow{!(e,j)} (\ell_i', \varepsilon) \in \Delta_i$;
2. $\mathcal{Q}_j' = \mathcal{Q}_j e$;
3. $\ell_k' = \ell_k$ for all $k \in \{1, \ldots, n\} \setminus \{i\}$; and
4. $\mathcal{Q}_k' = \mathcal{Q}_k$ for all $k \in \{1, \ldots, n\} \setminus \{j\}$.

The execution model of a CQS is strictly interleaving. That is, in each step, one of the two above transitions **(internal)** or **(transmission)** is performed for a nondeterministically chosen machine i.

Queue-Bounded and Queue-Unbounded Reachability. Given a CQS P^n, a state $s = \langle(\ell_1, \mathcal{Q}_1), \ldots, (\ell_n, \mathcal{Q}_n)\rangle$, and a number k, the *queue-bounded reachability problem* (for s and k) determines whether s is *reachable under queue bound* k, i.e. whether there exists a path $s_0 \rightarrow s_1 \ldots \rightarrow s_z$ such that $s_0 = s^I$, $s_z = s$, and for $i \in \{0, \ldots, z\}$, all queues in state s_i have at most k events. Queue-bounded reachability for k is trivially decidable, by making enqueue actions for queues of size k *blocking* (the sender cannot continue), which results in a finite state space. We write $R_k = \{s : s \text{ is reachable under queue bound } k\}$.

Queue-bounded reachability will be used in this paper as a tool for solving our actual problem of interest: Given a CQS P^n and a state s, the *Queue-UnBounded reachability Analysis (QUBA) problem* determines whether s is reachable, i.e. whether there exists a (queue-unbounded) path from s^I to s. The QUBA problem is undecidable [8]. We write R $(= \bigcup_{k \in \mathbb{N}} R_k)$ for the set of reachable states.

4 Convergence via Partial Abstract Transformers

In this section, we formalize our approach to detecting the convergence of a suitable sequence of *observations* about the states R_k reachable under k-bounded semantics. We define the observations as abstractions of those states, resulting in sets $\overline{R}_k$. We then investigate the convergence of the sequence $(\overline{R}_k)_{k=0}^{\infty}$.

4.1 List Abstractions of Queues

Our abstraction function applies to queues, as defined below. Its action on machine and system states then follows from the hierarchical design of a CQS. Let $|\mathcal{Q}|$ denote the number of events in $\mathcal{Q}$, and $\mathcal{Q}[i]$ the ith event in $\mathcal{Q}$ $(0 \leq i < |\mathcal{Q}|)$.

Definition 1. *For a parameter $p \in \mathbb{N}$, the* ***list abstraction*** *function $\alpha_p : \Sigma^* \mapsto \Sigma^*$ is defined as follows:*

1. $\alpha_p(\varepsilon) = \varepsilon$.
2. *For a non-empty queue $\mathcal{Q} = P \cdot e$,*

$$\alpha_p(\mathcal{Q}) = \begin{cases} \alpha_p(P) & \text{if there exists } j \text{ s.t. } p \leq j < |P| \text{ and } \mathcal{Q}[j] = e \\ \alpha_p(P) \cdot e & \text{otherwise} \end{cases} . \quad (3)$$

Intuitively, α_p abstracts a queue by leaving its first p events unchanged (an idea also used in [16]). Starting from position p it keeps only the first occurrence of each event e in the queue, if any; repeat occurrences are dropped.[2] The preservation of existence and order of the first occurrences of all present events motivates the term *list abstraction*. An alternative is an abstraction that keeps only the *set* (not: list) of queue elements from position p, i.e. it ignores multiplicity *and order*. This is by definition less precise than the list abstraction and provided no efficiency advantages in our experiments. An abstraction that keeps only the queue head proved cheap but too imprecise.

The motivation for parameter p is that many protocols proceed in *rounds* of repeating communication patterns, involving a bounded number of message exchanges. If p exceeds that number, the list abstraction's loss of information may be immaterial.

We write an abstract queue $\overline{\mathcal{Q}} = \alpha_p(\mathcal{Q})$ in the form $\mathit{pref} \,|\, \mathit{suff}$ s.t. $p = |\mathit{pref}|$, and refer to *pref* as $\overline{\mathcal{Q}}$'s *prefix* (shared with $\mathcal{Q}$), and *suff* as $\overline{\mathcal{Q}}$'s *suffix*.

Example 2. *The queues $\mathcal{Q} \in \{bbbba, bbba, bbbaa\}$ are α_2-****equivalent****: $\alpha_2(\mathcal{Q}) = bb \,|\, ba$.*

We extend α_p to act on a machine state via $\alpha_p(\ell_i, \mathcal{Q}_i) = (\ell_i, \alpha_p(\mathcal{Q}_i))$, on a state via $\alpha_p(s) = \langle(\ell_1, \alpha_p(\mathcal{Q}_1)), \ldots, (\ell_n, \alpha_p(\mathcal{Q}_n))\rangle$, and on a set of states pointwise via $\alpha_p(S) = \{\alpha_p(s) : s \in S\}$.

Discussion. The abstract state space is finite since the queue prefix is of fixed size, and each event in the suffix is recorded at most once (the event alphabet is finite). The sets of reachable abstract states grow monotonously with increasing queue size bound k, since the sets of reachable concrete states do:

$$k_1 \leq k_2 \quad \Rightarrow \quad R_{k_1} \subseteq R_{k_2} \quad \Rightarrow \quad \alpha_p(R_{k_1}) \subseteq \alpha_p(R_{k_2}) \; .$$

Finiteness and monotonicity guarantee convergence of the sequence of reachable abstract states.

We say the abstraction function α_p *respects* a property of a state if, for any two α_p-equivalent states (see Example 2), the property holds for both or for neither. Function α_p respects properties that refer to the local-state part of a machine, and to the first $p + 1$ events of its queue (which are preserved by α_p). In addition, the property may look beyond the prefix and refer to the existence of events in the queue, but not their frequency or their order after the first occurrence.

[2] Note that the head of the queue is always preserved by α_p, even for $p = 0$.

The rich information preserved by the abstraction (despite being finite-state) especially pays off in connection with the defer feature in the P language, which allows machines to delay handling certain events at the head of a queue [11]. The machine identifies the first non-deferred event in the queue, a piece of information that is precisely preserved by the list abstraction (no matter what p).

Definition 3. *Given an abstract queue* $\overline{\mathcal{Q}} = e_0 \dots e_{p-1} \mid e_p \dots e_{z-1}$*, the* ***concretization function*** $\gamma_p \colon \Sigma^* \to 2^{\Sigma^*}$ *maps* $\overline{\mathcal{Q}}$ *to the* language *of the regular expression*

$$RE_p(\overline{\mathcal{Q}}) := e_0 \dots e_{p-1} e_p \{e_p\}^* e_{p+1} \{e_p, e_{p+1}\}^* \dots e_{z-1} \{e_p, \dots, e_{z-1}\}^* \,, \qquad (4)$$

i.e. $\gamma_p(\overline{\mathcal{Q}}) := \mathcal{L}(RE_p(\overline{\mathcal{Q}}))$.

As a special case, $RE_p(\varepsilon) = \varepsilon$ and so $\gamma_p(\varepsilon) = \mathcal{L}(\varepsilon) = \{\varepsilon\}$ for the empty queue. We extend γ_p to act on abstract (machine or global) states in a way analogous to the extension of α_p, by moving it inside to the queues occurring in those states.

4.2 Abstract Convergence Detection

Recall that finiteness and monotonicity of the sequence $(\overline{R}_k)_{k=0}^{\infty}$ guarantee its convergence, so nothing seems more suggestive than to compute the limit. We summarize our overall procedure to do so in Algorithm 1. The procedure iteratively increases the queue bound k and computes the concrete and (per α_p-projection) the abstract reachability sets R_k and $\overline{R}_k$. If, for some k, an error is detected, the procedure terminates (Lines 4–5; in practice implemented as an on-the-fly check).

Algorithm 1. Queue-unbounded reachability analysis

Input: CQS with transition relation $\rightarrow$, $p \in \mathbb{N}$, property Φ respected by α_p.

```
1: compute R_0; R̄_0 := α_p(R_0)
2: for k := 1 to ∞ do
3:    compute R_k; R̄_k := α_p(R_k)
4:    if ∃r ∈ R_k : r ⊭ Φ then
5:       return "error reachable with queue bound k"
6:    if |R̄_k| = |R̄_{k−1}| then
7:       T̄ := (α_p ∘ Im_deq ∘ γ_p)(R̄_k)        ▷ partial best abstract transformer
8:       if T̄ ⊆ R̄_k then
9:          return "safe for any queue bound"
```

The key of the algorithm is reflected in Lines 6–9 and is based on the following idea (all claims are proved as part of Theorem 4 below). If the computation of $\overline{R}_k$ reveals no new abstract states in round k (Line 6; by monotonicity,

"same size" implies "same sets"), we apply the *best abstract transformer* [9,27] $\overline{Im} := \alpha_p \circ Im_{\rightarrow} \circ \gamma_p$ to $\overline{R}_k$: if the result is contained in $\overline{R}_k$, the abstract reachability sequence has converged. However, we can do better: we can restrict the successor function $Im_{\rightarrow}$ of the CQS to *dequeue* actions, denoted Im_{deq} in Line 7. The ultimate reason is that firing a local or transmit action on two α_p-equivalent states r and s results again in α_p-equivalent states r' and s'. This fact does *not* hold for dequeue actions: the successors r' and s' of dequeues depend on the abstracted parts of r and s, resp., which may differ and become "visible" during the dequeue (e.g. the event behind the queue head moves into the head position). Our main result therefore is: if $\overline{R}_k = \overline{R}_{k-1}$ and dequeue actions do not create new abstract states (Lines 7 and 8), sequence $(\overline{R}_k)_{k=0}^{\infty}$ has converged:

Theorem 4. *If $\overline{R}_k = \overline{R}_{k-1}$ and $\overline{T} \subseteq \overline{R}_k$, then for any $K \geq k$, $\overline{R}_K = \overline{R}_k$.*

If the sequence of reachable abstract states has converged, then **all** reachable concrete states (any k) belong to $\gamma_p(\overline{R}_k)$ (for the current k). Since the abstraction function α_p respects property Φ, we know that if any reachable concrete state violated Φ, so would any other concrete state that maps to the same abstraction. However, for each abstract state in $\overline{R}_k$, Line 4 has examined at least one state r in its concretization; a violation was not found. We conclude:

Corollary 5. *Line 9 of Algorithm 1 correctly asserts that no reachable concrete state of the given CQS violates Φ.*

The corollary (along with the earlier statement about Lines 4–5) confirms the partial correctness of Algorithm 1. The procedure is, however, necessarily incomplete: if no error is detected and the convergence condition in Line 8 never holds, the **for** loop will run forever.

We conclude this part with two comments. First, note that we do not compute the sets $\overline{R}_k$ as reachability fixpoints in the abstract domain (i.e. the domain of α_p). Instead, we compute the *concrete* reachability sets first, and then obtain the $\overline{R}_k$ via projection (Line 1). The reason is that the projection gives us the *exact* set of abstractions of reachable concrete states, while an abstract fixpoint likely overapproximates (for instance, the best abstract transformer from Line 7 does) and loses precision. Note that a primary motivation for computing abstract fixpoints, namely that the concrete fixpoint may not be computable, does not apply here: the concrete domains are finite, for each k.

Second, we observe that this projection technique comes with a cost: sequence $(\overline{R}_k)_{k=0}^{\infty}$ may *stutter* at intermediate moments: $\overline{R}_k \subsetneq \overline{R}_{k+1} = \overline{R}_{k+2} \subsetneq \overline{R}_{k+3}$. The reason is that $\overline{R}_{k+3}$ is not obtained as a functional image of $\overline{R}_{k+2}$, but by projection from R_{k+3}. As a consequence, we cannot short-cut the convergence detection by just "waiting" for $(\overline{R}_k)_{k=0}^{\infty}$ to stabilize, despite the finite domain.

4.3 Computing Partial Best Abstract Transformers

Recall that in Line 7 we compute

$$\overline{T} = \overline{Im}_{deq}(\overline{R}_k) = (\alpha_p \circ Im_{deq} \circ \gamma_p)(\overline{R}_k) \ . \tag{5}$$

The line applies the best abstract transformer, restricted to dequeue actions, to $\overline{R}_k$. This result cannot be computed as defined in (5), since $\gamma_p(\overline{R}_k)$ is typically infinite. However, $\overline{R}_k$ is finite, so we can iterative over $\bar{r} \in \overline{R}_k$, and little information is actually needed to determine the abstract successors of $\bar{r}$. The "infinite fragment" of $\bar{r}$ remains unchanged, which makes the action implementable.

Formally, let $\bar{r} = (\ell, \overline{\mathcal{Q}})$ with $\overline{\mathcal{Q}} = e_0 e_1 \dots e_{p-1} \,|\, e_p e_{p+1} \dots e_{z-1}$. To apply a dequeue action to $\bar{r}$, we first perform local-state updates on ℓ as required by the action, resulting in ℓ'. Now consider $\overline{\mathcal{Q}}$. The first suffix event, e_p, moves into the prefix due to the dequeue. We do not know whether there are later occurrences of e_p before or after the first suffix occurrences of $e_{p+1} \dots e_{z-1}$. This information determines the possible abstract queues resulting from the dequeue. To compute the exact best abstract transformer, we enumerate these possibilities:

$$\overline{Im}_{deq}(\{(\ell, \overline{\mathcal{Q}})\}) \quad = \quad \{\ (\ell', \overline{\mathcal{Q}}')\ :\ \overline{\mathcal{Q}}' \in \left\{ \begin{array}{l} e_1 \dots e_p \,|\, e_{p+1} e_{p+2} \dots e_{z-1} \\ e_1 \dots e_p \,|\, \boxed{e_p}\, e_{p+1} e_{p+2} \dots e_{z-1} \\ e_1 \dots e_p \,|\, e_{p+1} \boxed{e_p}\, e_{p+2} \dots e_{z-1} \\ \qquad\qquad \vdots \\ e_1 \dots e_p \,|\, e_{p+1} e_{p+2} \dots e_{z-1} \boxed{e_p} \end{array} \right\} \}$$

The first case for $\overline{\mathcal{Q}}'$ applies if there are no occurrences of e_p in the suffix after the dequeue. The remaining cases enumerate possible positions of the *first* occurrence of e_p (boxed, for readability) in the suffix after the dequeue. The cost of this enumeration is linear in the length of the suffix of the abstract queue.

Since our list abstraction maintains the first occurrence of each event, the semantics of defer (see the *Discussion* in Sect. 4.1) can be implemented abstractly without loss of information (not shown above, for simplicity).

5 Abstract Queue Invariant Checking

The abstract transformer function in Sect. 4 is used to decide whether sequence $(\overline{R}_k)_{k=0}^{\infty}$ has converged. Being an overapproximation, the function may generate *spurious* states: they are not reachable, i.e. no concretization of them is. Unfortunate for us, spurious abstract states always prevent convergence.

A key empirical observation is that concretizations of spurious abstract states often violate simple machine invariants, which can be proved from the perspective of a single machine, while collapsing all other machines into a nondeterministically behaving environment. Consider our example from Sect. 2 for $p = 0$. It fails to converge since Line 7 generates an abstract state $\bar{s}$ that features a Done event followed by a Prime event in the Receiver's queue. A light-weight static analysis proves that the Sender's machine permits no path from the send Done to the send Prime statement. Since **every** concretization of $\bar{s}$ features a Done followed by a Prime event, the abstract state $\bar{s}$ is spurious and can be eliminated.

Our tool assists users in *discovering* candidate machine invariants, by facilitating the inspection of states in $\overline{T} \setminus \overline{R}_k$ (which foil the test in Line 8). We *discharge* such invariants separately, via a simple sequential model-check or static analysis. In the section we focus on the more interesting question of how to *use* them. Formally, suppose the P program comes with a *queue invariant* I, i.e. an invariant property of *concrete* queues. The *abstract invariant checking problem* is to decide, for a given abstract queue $\overline{\mathcal{Q}}$, whether *every* concretization of $\overline{\mathcal{Q}}$ violates I; in this case, and this case only, an abstract state containing $\overline{\mathcal{Q}}$ can be eliminated. In the following we define a language QuTL for specifying concrete queue invariants (5.1), and then show how checking an abstract queue against a QuTL invariant can be efficiently solved as a model checking problem (5.2).

5.1 Queue Temporal Logic (QuTL)

Our logic to express invariant properties of queues is a form of first-order linear-time temporal logic. This choice is motivated by the logic's ability to constrain the order (via temporal operators) and multiplicity of queue events, the latter via relational operators that express conditions on the number of event occurrences.

Queue Relational Expressions (QuRelE). These are of the form $\#e \triangleright c$, where $e \in \Sigma$ (queue alphabet), $\triangleright \in \{<, \leq, =, \geq, >\}$, and $c \in \mathbb{N}$ is a literal natural number. The *value* of a QuRelE is defined as the Boolean

$$V(\#e \triangleright c) \quad = \quad |\{i \in \mathbb{N} : 0 \leq i < |\mathcal{Q}| \wedge \mathcal{Q}[i] = e\}| \;\triangleright\; c \tag{6}$$

where $|\cdot|$ denotes set cardinality and $\triangleright$ is interpreted as the standard integer arithmetic relational operator. In the following we write $\mathcal{Q}[i \rightarrow]$ (read: "$\mathcal{Q}$ from i") for the queue obtained from queue $\mathcal{Q}$ by dropping the first i events.

Definition 6 (Syntax of QuTL). *The following are QuTL formulas:*

- *false and true.*
- *e, for $e \in \Sigma$.*
- *E, for a queue relational expression E.*
- *$\mathsf{X}\,\phi$, $\mathsf{F}\,\phi$, $\mathsf{G}\,\phi$, for a QuTL formula ϕ.*

The set QuTL is the Boolean closure of the above set of formulas.

Definition 7 (Concrete semantics of QuTL). *Concrete queue $\mathcal{Q}$ **satisfies** QuTL formula ϕ, written $\mathcal{Q} \models \phi$, depending on the form of ϕ as follows.*

- *$\mathcal{Q} \models true$.*
- *for $e \in \Sigma$, $\mathcal{Q} \models e$ iff $|\mathcal{Q}| > 0$ and $\mathcal{Q}[0] = e$.*
- *for a queue relational expression E, $\mathcal{Q} \models E$ iff $V(E) = true$.*
- *$\mathcal{Q} \models \mathsf{X}\,\phi$ iff $|\mathcal{Q}| > 0$ and $\mathcal{Q}[1 \rightarrow] \models \phi$.*
- *$\mathcal{Q} \models \mathsf{F}\,\phi$ iff there exists $i \in \mathbb{N}$ such that $0 \leq i < |\mathcal{Q}|$ and $\mathcal{Q}[i \rightarrow] \models \phi$.*
- *$\mathcal{Q} \models \mathsf{G}\,\phi$ iff for all $i \in \mathbb{N}$ such that $0 \leq i < |\mathcal{Q}|$, $\mathcal{Q}[i \rightarrow] \models \phi$.*

Satisfaction of Boolean combinations is defined as usual, e.g. $\mathcal{Q} \models \neg\phi$ *iff* $\mathcal{Q} \not\models \phi$. *No other pair* $(\mathcal{Q}, \phi)$ *satisfies* $\mathcal{Q} \models \phi$.

For instance, formula $\#e \leq 3$ is true exactly for queues containing at most 3 e's, and formula $\mathsf{G}(\#e \geq 1)$ is true of $\mathcal{Q}$ iff $\mathcal{Q}$ is empty or its final event (!) is e. See App. B of [23] for more examples.

Algorithmically checking whether a concrete queue $\mathcal{Q}$ satisfies a QuTL formula ϕ is straightforward, since $\mathcal{Q}$ is of fixed size and straight-line. The situation is different with abstract queues. Our motivation here is to declare that an abstract queue $\overline{\mathcal{Q}}$ *violates* a formula ϕ if *all its concretizations* (Definition 3) do: under this condition, if ϕ is an invariant, we know $\overline{\mathcal{Q}}$ is not reachable. Equivalently:

Definition 8 (Abstract semantics of QuTL). *Abstract queue* $\overline{\mathcal{Q}}$ ***satisfies*** *QuTL formula* ϕ, *written* $\overline{\mathcal{Q}} \models_p \phi$, *if some concretization of* $\overline{\mathcal{Q}}$ *satisfies* ϕ:

$$\overline{\mathcal{Q}} \models_p \phi \quad := \quad \exists \mathcal{Q} \in \gamma_p(\overline{\mathcal{Q}}) : \mathcal{Q} \models \phi. \tag{7}$$

For example, we have $bb\,|\,ba \models_2 \mathsf{G}(a \Rightarrow \mathsf{G}\,\neg b)$ since for instance $bbba \in \gamma_2(bb\,|\,ba)$ satisfies the formula. See App. B of [23] for more examples.

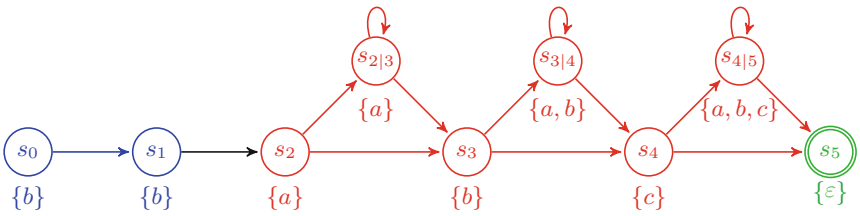


Fig. 2. LTS for $\overline{\mathcal{Q}} = bb\,|\,abc$ $(p = 2)$, with label sets written below each state. The blue and red parts encode the concretizations of the prefix and suffix of $\overline{\mathcal{Q}}$, resp. (Color figure online)

5.2 Abstract QuTL Model Checking

A QuTL *constraint* is a QuTL formula without Boolean connectives. We first describe how to model check against QuTL constraints, and come back to Boolean connectives at the end of Sect. 5.2.

Model checking an abstract queue $\overline{\mathcal{Q}}$ against a QuTL constraint ϕ, i.e. checking whether some concretization of $\overline{\mathcal{Q}}$ satisfies ϕ, can be reduced to a standard model checking problem over a labeled transition system (LTS) $M = (S, T, L)$ with states S, transitions T, and a labeling function $L\colon S \to 2^{\Sigma} \cup \{\varepsilon\}$. The LTS *characterizes* the concretization $\gamma_p(\overline{\mathcal{Q}})$ of $\overline{\mathcal{Q}}$, as illustrated in Fig. 2 using an example: the concretizations of $\overline{\mathcal{Q}}$ are formed from the regular-expression traces generated by paths of $\overline{\mathcal{Q}}$'s LTS that end in the double-circled green state.

The straightforward construction of the LTS M is formalized in App. A.2 of [23]. Its size is linear in $|\overline{\mathcal{Q}}|$: $|S| = p+2\times(|\overline{\mathcal{Q}}|-p)+1$ and $|T| = p+4\times(|\overline{\mathcal{Q}}|-p)$.

We call a path through M *complete* if it ends in the right-most state s_z of M (green in Fig. 2). The labeling function extends to paths via $L(s_i \to \ldots \to s_j) = L(s_i) \cdot \ldots \cdot L(s_j)$. This gives rise to the following characterization of $\gamma_p(\overline{\mathcal{Q}})$:

Lemma 9. *Given abstract queue $\overline{\mathcal{Q}}$ over alphabet Σ, let $M = (S, T, L)$ be its LTS.*

$$\gamma_p(\overline{\mathcal{Q}}) \quad = \quad \bigcup \{\mathcal{L}(L(\pi)) \in 2^{\Sigma^*} \mid \pi \textit{ is a complete path from } s_0 \textit{ in } M\}. \qquad (8)$$

We say path π *satisfies* ϕ, written $\pi \models_p \phi$, if there exists $\mathcal{Q} \in \mathcal{L}(L(\pi))$ s.t. $\mathcal{Q} \models \phi$.

Corollary 10. *Let $\overline{\mathcal{Q}}$ and M as in Lemma 9, and ϕ a QuTL constraint. Then the following are equivalent.*

1. $\overline{\mathcal{Q}} \models_p \phi$.
2. *There exists a complete path π from s_0 in M such that $\pi \models_p \phi$.*

Proof. immediate from Definition 8 and Lemma 9. □

Given an abstract queue $\overline{\mathcal{Q}}$, its LTS M, and a QuTL constraint ϕ, our abstract queue model checking algorithm is based on Corollary 10: we need to find a complete path from s_0 in M that satisfies ϕ. This is similar to standard model checking against existential temporal logics like ECTL, with two particularities:

First, paths must be complete. This poses no difficulty, as completeness is suffix-closed: a path ends in s_z iff any suffix does. This implies that temporal reductions on QuTL constraints work like in standard temporal logics. For example: there exists a complete path π from s_0 in M such that $\pi \models_p \mathsf{X}\phi$ iff there exists a complete path π' from some successor s_1 of s_0 such that $\pi' \models_p \phi$.

Second, we have domain-specific atomic (non-temporal) propositions. These are accommodated as follows, for an arbitrary start state $s \in S$:

$\exists \pi :$ π **from** s **complete and** $\pi \models_p e$ **(for** $e \in \Sigma$**):**
this is true iff $e \in L(s)$, as is immediate from the $\mathcal{Q} \models e$ case in Definition 7.
$\exists \pi :$ π **from** s **complete and** $\pi \models_p \#e > c$ **(for** $e \in \Sigma, c \in \mathbb{N}$**):** this is true iff

- the number of states reachable from s labeled e is greater than c, **or**
- there exists a state reachable from s labeled with e that has a self-loop.

The other relational expressions $\#e \triangleright c$ are checked similarly. □

Boolean Connectives. Let now ϕ be a full-fledged QuTL formula. We first bring it into negation normal form, by pushing negations inside, exploiting the usual dualities $\neg\mathsf{X} = \mathsf{X}\neg$, $\neg\mathsf{F} = \mathsf{G}\neg$, and $\neg\mathsf{G} = \mathsf{F}\neg$. The subset $\triangleright \in \{<, \leq, \geq, >\}$ of the queue relational expressions is semantically closed under negation; "$\neg=$" is replaced by "$> \vee <$". A path π from s satisfies $\neg e$ (for $e \in \Sigma$) iff $L(s) \neq \{e\}$: this condition states that either $L(s) = \varepsilon$, or there exists some label other than e in $L(s)$, so the *existential* property $\neg e$ holds.

Disjunctions are handled by distributing $\models_p$ over them: $\overline{\mathcal{Q}} \models_p \phi_1 \vee \phi_2$ iff $\overline{\mathcal{Q}} \models_p \phi_1 \vee \overline{\mathcal{Q}} \models_p \phi_2$. What remains are conjunctions. The existential flavor of $\models_p$ implies that $\models_p$ does *not* distribute over them; see Ex. 13 in App. B.1 of [23]. Suppose we ignore this and replace a check of the form $\overline{\mathcal{Q}} \models_p \phi_1 \wedge \phi_2$ by the **weaker** check $\overline{\mathcal{Q}} \models_p \phi_1 \wedge \overline{\mathcal{Q}} \models_p \phi_2$, which may produce false positives. Now consider how we use these results: if $\overline{\mathcal{Q}} \models_p \phi$ holds, we decide to *keep* the state containing the abstract queue. False positives during abstract model checks therefore may create extra work, but do not introduce unsoundness. In summary, our abstract model checking algorithm soundly approximates conjunctions, but remains exact for the purely disjunctive fragment of QuTL.

Table 1. Results: $\#M$: #P machines; *Loc*: #lines of code; *Safe*? = ✓: property holds; p: *minimum* unabstracted prefix for required convergence; $k_{\max}$: point of convergence or exposed bugs (– means divergence); *Time*: runtime (sec); *Mem.*: memory usage (Mb.).

ID/Program	Program Features			Pat			
	$\#M$	*Loc*	*Safe?*	p	$k_{\max}$	*Time*	*Mem.*
1/German-1	3	242	✓	4	–	TO	–
2/German-2	4	244	✓	4	–	TO	–
3/TokenRing-buggy	6	164	✗	0	2	241.44	35.96
4/TokenRing-fixed	6	164	✓	0	4	1849.25	130.87
5/FailureDetector	6	229	✓	0	4	183.99	12.38
6/OSR	5	378	✓	0	5	77.92	44.86
7/openWSN	6	294	✓	2	5	2574.25	376.29

ID/Program	Program Features			Pat			
	$\#M$	*Loc*	*Safe?*	p	$k_{\max}$	*Time*	*Mem.*
8/Failover	4	132	✓	0	2	2.91	8.56
9/MaxInstances	4	79	✓	0	3	0.14	0.56
10/PingPong	2	76	✓	0	2	0.06	0.43
11/BoundedAsync	4	96	✓	0	5	203.39	29.32
12/PingFlood	2	52	✓	4	5	0.11	0.43
13/Elevator-buggy	4	270	✗	0	1	1.29	5.23
14/Elevator-fixed	4	271	✓	0	4	49.23	45.36

6 Empirical Evaluation

We implemented the proposed approaches in C# atop the bounded model checker PTester [11], an analysis tool for P programs. PTester employs a bounded exploration strategy similar to Zing [4]. We denote by Pat the implementation of Algorithm 1, and by Pat+I the version with queue invariants ("Pat+ Invariants"). A detailed introduction to tool design and implementation is available online [22].

Experimental Goals. We evaluate the approaches against the following questions:

Q1. Is Pat effective: does it converge for many programs? for what values of k?
Q2. What is the impact of the QuTL invariant checking?

Experimental Setup. We collected a set of P programs (available online [22]); most have been used in previous publications:

1–5: protocols implemented in P: the German Cache Coherence protocol with different number of clients (**1**–**2**) [11], a buggy version of a token ring protocol [11], and a fixed version (**3**–**4**), and a failure detector protocol from [25] (**5**).

6–7: two device drivers where OSR is used for testing USB devices [10].
8–14: miscellaneous: **8–10** [25], **11** [15], **12** is the example from Sect. 2, **13–14** are the buggy and fixed versions of an Elevator controller [11].

We conduct two types of experiments: (i) we run Pat on each benchmark to empirically answer **Q1**; (ii) we run Pat+I on the examples which fail to verify in (i) to answer **Q2**. All experiments are performed on a 2.80 GHz Intel(R) Core(TM) i7-7600 machine with 8 GB memory, running 64-bit Windows 10. The timeout is set to 3600 s (1h); the memory limit to 4 GB.

Results. Table 1 shows that Pat converges on *almost all* safe examples (and successfully exposes the bugs for unsafe ones). Second, in most cases, the $k_{\max}$ where convergence was detected is small, 5 or less. This is what enables the use of this technique in practice: the exploration space grows fast with k, so early convergence is critical. Note that $k_{\max}$ is guaranteed to be the smallest value for which the respective example converges. If convergent, the verification succeeded fully automatically: the queue abstraction prefix parameter p is incremented in a loop whenever the current value of p caused a spurious abstract state.

The German protocol does not converge in reasonable time. In this case, we request minimal manual assistance from the designer. Our tool inspects spurious abstract states, compares them to actually reached abstract states, and suggests candidate invariants to exclude them. We describe the process of invariant discovery, and why and how they are easy to prove, in [22].

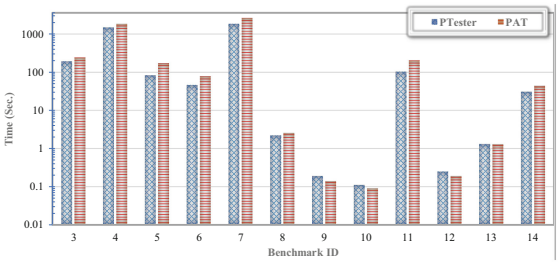
The following table shows the invariants that make the German protocol converge, and the resulting times and memory consumption.

Program	p	$k_{\max}$	*Time*	*Mem.*	Invariant
German-1	0	4	15.65	45.65	Server: $\#req\_excl \leq 1 \wedge \#req\_share \leq 1$
German-2	0	4	629.43	284.75	Client: $\#ask\_excl \leq 1 \wedge \#ask\_share \leq 1$

The invariant states that there is always at most one exclusive request and at most one shared request in the Server or Client machine's queue.

Performance Evaluation. We finally consider the following question: *To perform full verification, how much overhead does* Pat *incur compared to PTester?* We iteratively run PTester with a queue bound from 1 up to $k_{\max}$ (from Table 1).

The figure on the right compares the running times of Pat and PTester. We observe that the difference is small, in all cases, suggesting that turning PTester into a full verifier comes with little



extra cost. Therefore, as for improving PAT's scalability, the focus should be on the efficiency of the R_k computation (Line 3 in Algorithm 1). Techniques that lend themselves here are *partial order reduction* [2,28] or *symmetry reduction* [29]. Note that our proposed approach is orthogonal to how these sets are computed.

7 Related Work

Automatic verification for asynchronous event-driven programs communicating via unbounded FIFO queues is undecidable [8], even when the agents are finite-state machines. To sidestep the undecidability, various remedies are proposed. One is to underapproximate program behaviors using various bounding techniques; examples include depth- [17] and context-bounded analysis [19,20,26], delay-bounding [13], bounded asynchrony [15], preemption-bounding [24], and phase-bounded analysis [3,6]. It has been shown that most of these bounding techniques admit a decidable model checking problem [19,20,26] and thus have been successfully used in practice for finding bugs.

Gall et al. proposed an abstract interpretation of FIFO queues in terms of regular languages [16]. While our works share some basic insights about taming queues, the differences are fundamental: our abstract domain is *finite*, guaranteeing convergence of our sequence. In [16] the abstract domain is infinite; they propose a widening operator for fixpoint computation. More critically, we use the abstract domain *only* for convergence detection; the set of reachable states returned is in the end exact. As a result, we can prove and refute properties but may not terminate; [16] is inexact and cannot refute but always returns.

Several partial verification approaches for asynchronous message-passing programs have been presented recently [5,7,10]. In [5], Bakst et al. propose *canonical sequentialization*, which avoids exploring all interleavings by sequentializing concurrent programs. Desai et al. [10] propose an alternative way, namely by prioritizing receive actions over send actions. The approach is complete in the sense that it is able to construct *almost-synchronous invariants* that cover all reachable local states and hence suffice to prove local assertions. Similarly, Bouajjani et al. [7] propose an iterative analysis that bounds send actions in each interaction phase. It approaches the completeness by checking a program's synchronizability under the bounds. Similar to our work, the above three works are sound but incomplete. An experimental comparison against the techniques reported in [7,10] fails due to the unavailability of a tool that implements them. While tools implementing these techniques are not available [7,10], a comparison based on what is reported in the papers suggests that our approach is competitive in both performance and precision.

Our approach can be categorized as a *cutoff* detection technique [1,12,14,28]. Cutoffs are, however, typically determined statically, often leaving them too large for practical verification. Aiming at minimal cutoffs, our work is closer in nature to earlier *dynamic* strategies [18,21], which targeted different forms of concurrent programs. The *generator* technique proposed in [21] is unlikely to work for P programs, due to the large local state space of machines.

8 Conclusion

We have presented a method to verify safety properties of asynchronous event-driven programs of agents communicating via unbounded queues. Our approach is sound but incomplete: it can both prove (or, by encountering bugs, disprove) such properties but may not terminate. We empirically evaluate our method on a collection of P programs. Our experimental results showcase our method can successfully prove the correctness of programs; such proof is achieved with little extra resource costs compared to plain state exploration. Future work includes an extension to P programs with other sources of unboundedness than the queue length (e.g. messages with integer *payloads*).

Acknowledgments. We thank Dr. Vijay D'Silva (Google, Inc.), for enlightening discussions about partial abstract transformers.

References

1. Abdulla, A.P., Haziza, F., Holík, L.: All for the price of few (parameterized verification through view abstraction). In: VMCAI, pp. 476–495 (2013)
2. Abdulla, P., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL, pp. 373–384 (2014)
3. Abdulla, P.A., Atig, M.F., Cederberg, J.: Analysis of message passing programs using SMT-solvers. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 272–286. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-02444-8_20
4. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: a model checker for concurrent software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_42
5. Bakst, A., Gleissenthall, K.v., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL **1**(OOPSLA), 110:1–110:27 (2017)
6. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. Int. J. Softw. Tools Technol. Transf. **16**(2), 127–146 (2014)
7. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 372–391. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_23
8. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
9. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL, pp. 269–282 (1979)
10. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. In: OOPSLA, pp. 709–725 (2014)
11. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: safe asynchronous event-driven programming. In: PLDI, pp. 321–332 (2013)
12. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 236–254. Springer, Heidelberg (2000). https://doi.org/10.1007/10721959_19

13. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: POPL, pp. 411–422 (2011)
14. Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: POPL, pp. 407–420 (2015)
15. Fisher, J., Henzinger, T.A., Mateescu, M., Piterman, N.: Bounded asynchrony: concurrency for modeling cell-cell interactions. In: Fisher, J. (ed.) FMSB 2008. LNCS, vol. 5054, pp. 17–32. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68413-8_2
16. Le Gall, T., Jeannet, B., Jéron, T.: Verification of communication protocols using abstract interpretation of FIFO queues. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 204–219. Springer, Heidelberg (2006). https://doi.org/10.1007/11784180_17
17. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186 (1997)
18. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_55
19. La Torre, S., Parthasarathy, M., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222 (2009)
20. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. **35**(1), 73–97 (2009)
21. Liu, P., Wahl, T.: CUBA: interprocedural context-unbounded analysis of concurrent programs. In: PLDI, pp. 105–119 (2018)
22. Liu, P., Wahl, T., Lal, A.: (2019). www.khoury.northeastern.edu/home/lpzun/quba
23. Liu, P., Wahl, T., Lal, A.: Verifying asynchronous event-driven programs using partial abstract transformers (extended manuscript). CoRR abs/1905.09996 (2019)
24. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
25. P-GitHub: The P programming langugage (2019). https://github.com/p-org/P
26. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_7
27. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_21
28. Sousa, M., Rodríguez, C., D'Silva, V., Kroening, D.: Abstract interpretation with unfoldings. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 197–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_11
29. Wahl, T., Donaldson, A.: Replication and abstraction: symmetry in automated formal verification. Symmetry **2**(2), 799–847 (2010)