Verifying Asynchronous Event-Driven Programs Using Partial Abstract Transformers

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 [15]. 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) [14,9], or by reasoning about a small number of representative execution schedules, to avoid interleaving explosion [6].

In this paper we consider the P language for AED programming [15]. 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 [10].

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 [17,3,15,8] 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 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? The sequence $(R_k)_{k=0}^{\infty}$ of reachable global states in general grows without bound. Instead, 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 abstract reachability sets $\overline{R}_k$ could be determined approximately, as the fixpoint under an appropriate abstract image operation $\overline{Im}$ [13]. In this paper, however, we compute each $\overline{R}_k$ *exactly*, by first determining the exact concrete set R_k (which is finite and computable), and then obtaining $\overline{R}_k$ as $\alpha(R_k)$.

The second major step in our approach is the detection of the point of convergence of $(\overline{R}_k)_{k=0}^{\infty}$[1]: We show that, for the *best abstract transformer* $\overline{Im}$ [13,30, see Sec. 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 of a single machine, irrespective of the behavior

[1] Note that simply stopping as soon as $\overline{R}_k = \overline{R}_{k+1}$ is unsound; see Sec. 4.2.

of other machines. By their nature, machine invariants 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 provable invariants; (ii) almost all examples converges at a small value of $k_{\max}$. This allows our method to succeed in practice.

Proofs and other supporting material can be found in the Appendix.

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 [15]. Fig. 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 head of the Receiver's queue is always Ping. 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 Prime or 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

$$\mathcal{Q} = \text{Prime.Prime.Prime.Done.Ping.Ping.Ping.Ping} \tag{1}$$

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

```
event PRIME, DONE, PING;

machine Sender {
  var receiver: machine;
  start state Init {
    entry {
      receiver = new Receiver();
    }
    goto Prime_it;
  }
  state Prime_it {
    entry {
      var i:int;
      while (i < 3) { // 3x PRIME
        send receiver, PRIME; i = i + 1;
      }
      send receiver, DONE; goto Ping_it;
    }
  }
  state Ping_it {
    entry {
      send receiver, PING; goto Ping_it;
    }
  }
}

machine Receiver {
  start state Init {
    defer PRIME;
    on DONE goto Ignore_it;
  }

  state Ignore_it {
    ignore PRIME, PING;
  }
}
```

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.

the head of the queue. This and the machine state is 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 Sec. 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 refinement maintains finiteness of the abstraction and increases precision, by revealing that the queue starts with three PRIME events; see Sec. 4.1 for a general motivation of this feature. 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. Inspecting the abstract state $\bar{s} \in \overline{Im}(\overline{R}_5) \setminus \overline{R}_5$ that foils the test (reverting to prefix $p = 0$), 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

$$\mathsf{G}\,(\textsc{Done} \Rightarrow \mathsf{G}\,\neg\textsc{Prime}) \tag{2}$$

in a temporal-logic like notation called QuTL (Sec. 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.[2] 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):

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

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

[2] The P language in principle permits unbounded machine creation, a feature that we do not consider here as it 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 path from s^I to s (without queue bound). The QUBA problem is undecidable [10]. We write R for the set of reachable states.

Obs. 1 *Given a CQS P^n and $k \in \mathbb{N}$, $R_k \subseteq R$.*

That is, queue-bounded underapproximates queue-unbounded reachability.

4 Convergence Detection using 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 $\mathcal{Q}$ ($0 \leq i < |\mathcal{Q}|$). For example, $\mathcal{Q}[0]$ denotes the event at the head of a non-empty queue.

Def. 2 *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 $\mathcal{Q} = P \cdot e$,*

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

Intuitively, α_p abstracts a queue by leaving its first p events unchanged. Starting from position p it keeps only the first occurrence of each event e in the queue, if any; repeat occurrences are dropped.[3] The preservation of existence and order of the first occurrences of all present events motivates the term *list abstraction*. 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 $\overline{\mathcal{Q}} = \mathit{pref} \,|\, \mathit{suff}$ such that $p = |\mathit{pref}|$, and refer to *pref* as $\overline{\mathcal{Q}}$'s *prefix* (which it shares with $\mathcal{Q}$), and *suff* as $\overline{\mathcal{Q}}$'s *suffix*.

Ex. 3 *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 Ex. 3), 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,

[3] Note that the head of the queue is always preserved by α_p, for any $p \geq 0$.

the property may look beyond the prefix and refer to the existence of events in the queue, but not their frequency or their order.

The rich information preserved by the abstraction (despite being finite-state) 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 [15]. 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).

Def. 4 *Given an abstract queue* $\overline{\mathcal{Q}} = e_0 \dots e_{p-1} \,|\, e_p \dots e_{z-1}$*, 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. $\gamma_p(\overline{\mathcal{Q}}) := L(RE_p(\overline{\mathcal{Q}}))$.

In particular, we have $RE_p(\varepsilon) = \varepsilon$ and hence $\gamma_p(\varepsilon) = \{\varepsilon\}$ 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 Alg. 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 $\to$, $p \in \mathbb{N}$, 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 Thm. 5 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* [13,30] $\overline{Im} := \alpha_p \circ Im_{\to} \circ \gamma_p$ 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:

Thm. 5 *If $\overline{R}_k = \overline{R}_{k-1}$ and $\overline{T} \subseteq \overline{R}_k$, then for any $K \geq k$, $\overline{R}_K = \overline{R}_k$.*

We note that the theorem requires the condition $\overline{R}_k = \overline{R}_{k-1}$ (tested efficiently in Line 6): without it, any action may lead to new reachable abstract states, not just dequeues (see proof in App. A.1).

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:

Cor. 6 *Line 9 of Alg. 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 Alg. 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:

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

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 Sec. 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 Sec. 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 *Ptolemaically* [12], i.e. from the perspective of a single machine, while collapsing all other machines into a nondeterministically behaving environment. Consider our example from Sec. 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 very 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, and then show how checking an abstract queue against a QuTL invariant can be efficiently solved as a model checking problem.

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. We introduce some light notation. 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; if $\mathcal{Q}$ has fewer than i events, $\mathcal{Q}[i \rightarrow]$ is undefined.

Queue Relational Expressions. These are expressions 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 queue relational expression 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.

Def. 7 (Syntax of QuTL) *The following are QuTL formulas:*

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

The set QuTL is the closure under ***disjunction*** *of the above set of formulas.*

See the discussion below Def. 9 on the use of conjunction and negation.

Def. 8 (Concrete semantics of QuTL) *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 $\mathcal{Q}[0] = e$; $\mathcal{Q} \models \neg e$ iff $\mathcal{Q} \not\models e$.*
- *for a relational expression E, $\mathcal{Q} \models E$ iff $V(E) = true$; $\mathcal{Q} \models \neg E$ iff $\mathcal{Q} \not\models E$.*

- $\mathcal{Q} \models \mathsf{X}\,\phi$ *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* $0 \leq i < |\mathcal{Q}|$, $\mathcal{Q}[i \rightarrow] \models \phi$.
- $\mathcal{Q} \models \phi_1 \vee \phi_2$ *iff* $\mathcal{Q} \models \phi_1$ *or* $\mathcal{Q} \models \phi_2$.

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 $\mathsf{G}(\#e \geq 1)$ is true of $\mathcal{Q}$ iff $\mathcal{Q}$ is empty or its final event is e. See App. B 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 do: under this condition, if ϕ is an invariant, we know $\overline{\mathcal{Q}}$ is not reachable. Equivalently:

Def. 9 (Abstract semantics of QuTL) *Abstract queue* $\overline{\mathcal{Q}}$ ***satisfies*** *QuTL formula* ϕ*, written* $\overline{\mathcal{Q}} \models_\alpha \phi$*, if there exists a concretization of* $\overline{\mathcal{Q}}$ *that satisfies* ϕ*:*

$$\overline{\mathcal{Q}} \models_\alpha \phi \quad := \quad \exists \mathcal{Q} \in \gamma(\overline{\mathcal{Q}}) : \mathcal{Q} \models \phi \; . \tag{7}$$

For example, given $p = 2$, we have $bb \mid ba \models_\alpha \mathsf{G}(a \Rightarrow \mathsf{G}\,\neg b)$ since for instance $bbba \in \gamma(bb \mid ba)$ satisfies the formula. See App. B for more examples.

Note that the existential flavor of $\models_\alpha$ implies that $\models_\alpha$ does not distribute over conjunction; see Ex. 13 in App. B.1 (analogously, LTL satisfaction does not distribute over disjunction). To keep the abstract model checking algorithm simple and compositional, Def. 7 excludes conjunction as a Boolean connective in QuTL, and allows negation only in front of atomic formulas.

5.2 Abstract QuTL Model Checking

Model checking an abstract queue $\overline{\mathcal{Q}}$ against a QuTL formula 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 $L \colon S \to 2^\Sigma$. The LTS *characterizes* the concretization $\gamma(\overline{\mathcal{Q}})$ of $\overline{\mathcal{Q}}$, as illustrated in Fig. 2 using an example: the concretizations of $\overline{\mathcal{Q}}$ are exactly the "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. 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 by pointwise application to their states. This gives rise to the following characterization of $\gamma(\overline{\mathcal{Q}})$ (Def. 4).

Lem. 10 *Given abstract queue* $\overline{\mathcal{Q}}$ *over alphabet* Σ*, let* $M = (S, T, L)$ *be its LTS.*

$$\gamma(\overline{\mathcal{Q}}) \quad = \quad \{L(q) \in \Sigma^* \mid \; q \text{ is a complete path from } s_0 \text{ in } M.\} \tag{8}$$

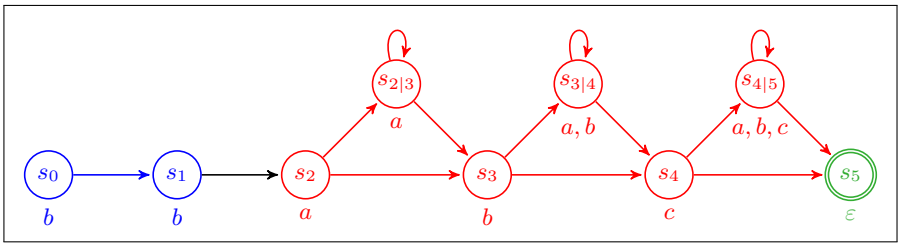


Fig. 2. LTS for $\overline{\mathcal{Q}} = bb \mid abc$ ($p = 2$), with label sets written underneath each state. The blue and red parts encode the concretizations of the prefix and suffix of $\overline{\mathcal{Q}}$, resp.

Cor. 11 *Let $\overline{\mathcal{Q}}$ and M as in Lem. 10, and ϕ a QuTL formula. Then the following are equivalent.*

1. $\overline{\mathcal{Q}} \models_\alpha \phi$.
2. *There exists a complete path q from s_0 in M such that $L(q) \models \phi$.*

Proof. immediate from Def. 9 and Lem. 10. □

Given an abstract queue $\overline{\mathcal{Q}}$, its LTS M, and a QuTL formula ϕ, our abstract queue model checking algorithm is based on Cor. 11: we need to find a complete path from s_0 in M whose labeling 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 formulas work like in standard temporal logics. For example: there exists a complete path π from s_0 in M such that $L(\pi) \models \mathsf{X}\,\psi$ iff for some successor s_1 of s_0, there exists a complete path π' from s_1 such that $L(\pi') \models \psi$. Similar reductions apply to eventualities F and invariants G.

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** $L(\pi) \models e$ **(for** $e \in \Sigma$**):**
this is true iff $e \in L(s)$, as is immediate from the $\mathcal{Q} \models e$ case in Def. 8.

$\exists \pi :$ π **from** s **complete and** $L(\pi) \models \neg e$ **(for** $e \in \Sigma$**):**
this is true iff $L(s) \neq \{e\}$: this condition states that either $L(s) = \emptyset$ (e.g. for the empty queue), or there exists some label other than e in $L(s)$, so the *existential* property $\neg e$ holds.

$\exists \pi :$ π **from** s **complete and** $L(\pi) \models \#e > c$ **(for** $e \in \Sigma, c \in \mathbb{N}$**):**
this is true iff
- the number of states reachable from (= to the right of) s labeled with 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 $\#e \triangleright c$ and their negations can be checked using similar state counting techniques. □

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	83.99	12.38
6/OSR	5	378	✓	0	5	77.92	44.86
7/openWSN	6	294	✓	2	5	2574.25	376.29
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	3	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 [15], an analysis tool for P programs. PTester employs a similar bounded exploration strategy as Zing [5]. We denote by PAT the implementation of Alg. 1, and by PAT+I the version with queue invariants ("PAT+ Invariants"). A detailed introduction to the tool design and implementation is available online [26].

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 an extensive set of P programs; most of these have been used in previous publications. We describe them briefly as follows:

1–5: a set of protocols implemented in P: the German Cache Coherence protocol with different number of clients (**1**–**2**) [15], a buggy version of a token ring protocol [15], and a fixed version (**3**–**4**), and a failure detector protocol from [1] (**5**).
6–7: two device drivers where OSR is used for testing USB devices [14].
8–14: miscellaneous: **8**–**10** [1], **11** [18], **12** is the example from Sec. 2, **13**–**14** are the buggy and fixed versions of an Elevator controller [15].

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 3600sec (1h); the memory limit to 4 GB. All benchmarks and results are available online [26].

Results. In Table 1, column PAT details the results of our basic approach. We first observe 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 the smallest value for which the respective example converges. For the converging examples, 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.

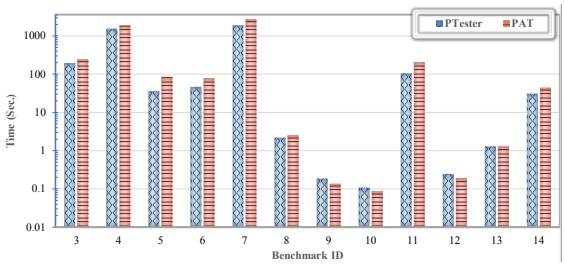
The GERMAN protocol does not converge in reasonable time. In this case, we request minimal manual assistance from the designer. Our tool inspects spurious abstract states, compares them to actually reached abstract states, and suggests candidate invariants to exclude them. We found that these candidates can often be proved by a sequential or static analysis, as in Sec. 2. We describe the process of invariant discovery, and why and how they are easy to prove, in [26].

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 machine's queue[4].

Performance Evaluation. We finally consider the following question: *To perform full verification, how much overhead does* PAT *incur compared to PTester?* We pass $k_{\max}$ (from Table 1) as an upper bound to PTester and perform bounded model checking. The figure on the right compares the running times of PAT and PTester. We observe that the difference is small, in all cases. This suggests that turning the systematic PTester into a full verifier comes with little extra cost, if any.



7 Related Work

Automatic verification for asynchronous event-driven programs communicating via unbounded FIFO queues is undecidable [10], even when the agents are finite-state machines. To sidestep the undecidability, various remedies are proposed which mainly focus on two directions. One is to consider decidable subclasses, by restricting the type of communication, like when the communication is *lossy* [4], *half-dumplex* [11], *rendezvous* [7], when the communication is via a bag of messages instead of a FIFO queue [31,20], when only a single kind of message is

[4] The two invariants express the same property, but for different machines.

allowed in the queue [28], or when the communication adheres to a forest architecture [22]. We do not have such restrictions in this paper.

The other direction is to underapproximate program behaviors using various bounding techniques; examples include depth- [19] and context-bounded analysis [29,23,24], delay-bounding [17], bounded asynchrony [18], preemption-bounding [27], and phase-bounded analysis [8,3]. All above bounding techniques systematically explore an artificially bounded space of reachable states. It has been shown that most of these bounding techniques admit a decidable model checking probleme [29,23,24]. These techniques have been successfully used in practice for finding bugs. They may however miss bugs that manifest after reaching the respective bounds. The goal of our approach is to obtain conclusive results at least in some cases.

Our approach can be categorized as a *cutoff* detection technique [16,2]. 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 [21,25], which targeted forms of multi-threaded shared-memory programs, not concurrent agents asynchronously communicating via unbounded queues. The *generator* technique proposed in [25] is unlikely to work for P programs, due to the large local state space of machines.

Several partial verification approaches for asynchronous message-passing programs have been recently presented [14,6,9]. In [6], Bakst et al. propose *canonical sequentialization*, which avoids exploring all interleavings by sequentializing concurrent programs. The approach is based on an observation that correct programs tend to be well-structured and hence requires programs to validate several structural properties. Desai et al [14] propose an alternative way to avoid reasoning about *all* interleavings, 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. In the worst case, however, it needs to explore all interleavings. Similarly, Bouajjani et al. [9] propose an iterative bounded analysis by bounding send actions in each interaction phase. It approaches the completeness by checking a program's synchronizability under the bounds. The synchronizability, in turn, allows concluding the program's correctness if no violation occurs. Similar to our work, all above three work are sound and incomplete. An experimental comparison against the techniques reported in [14,9] suffers from the unavailability of a tool that implements them.

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*). We plan to address this by a form of predicate abstraction.

References

1. https://github.com/p-org/p
2. Abdulla, A.P., Haziza, F., Holík, L.: All for the price of few (parameterized verification through view abstraction). In: VMCAI. pp. 476–495 (2013)
3. Abdulla, P.A., Atig, M.F., Cederberg, J.: Analysis of message passing programs using smt-solvers. In: ATVA. pp. 272–286 (2013)
4. Alur, R., Yannakakis, M.: Model checking of message sequence charts. In: CONCUR. pp. 114–129 (1999)
5. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A model checker for concurrent software. In: CAV. pp. 484–487 (2004)
6. Bakst, A., Gleissenthall, K.v., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL **1**(OOPSLA), 110:1–110:27 (Oct 2017)
7. Basu, S., Bultan, T., Ouederni, M.: Synchronizability for verification of asynchronously communicating systems. In: Verification, Model Checking, and Abstract Interpretation. pp. 56–71. Berlin, Heidelberg (2012)
8. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. Int. J. Softw. Tools Technol. Transf. **16**(2), 127–146 (Apr 2014)
9. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. In: CAV. pp. 372–391 (2018)
10. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (Apr 1983)
11. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Information and Computation **202**(2), 166 – 190 (2005)
12. Clarke, E., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: TACAS. pp. 33–47 (2008)
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
14. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. In: OOPSLA. pp. 709–725 (2014)
15. Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., Zufferey, D.: P: Safe asynchronous event-driven programming. In: PLDI. pp. 321–332 (2013)
16. Emerson, E., Kahlon, V.: Reducing model checking of the many to the few. In: CADE. vol. 1831, pp. 236–254 (2000)
17. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. pp. 411–422. POPL (2011)
18. Fisher, J., Henzinger, T.A., Mateescu, M., Piterman, N.: Bounded asynchrony: Concurrency for modeling cell-cell interactions. In: Formal Methods in Systems Biology. pp. 17–32 (2008)
19. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL. pp. 174–186 (1997)
20. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL. pp. 339–350 (2007)

21. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: CAV. pp. 645–659 (2010)
22. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: TACAS. pp. 299–314 (2008)
23. La Torre, S., Parthasarathy, M., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI. pp. 211–222 (2009)
24. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. **35**(1), 73–97 (Aug 2009)
25. Liu, P., Wahl, T.: CUBA: Interprocedural context-unbounded analysis of concurrent programs. In: PLDI. pp. 105–119 (2018)
26. Liu, P., Wahl, T., Lal, A.: https://www.khoury.northeastern.edu/home/lpzun/quba
27. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. pp. 446–455. PLDI (2007)
28. Peng, W., Purushothaman, S.: Analysis of a class of communicating finite state machines. Acta Informatica **29**(6), 499–522 (Jun 1992)
29. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. pp. 93–107 (2005)
30. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: VMCAI. pp. 252–266 (2004)
31. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: CAV. pp. 300–314 (2006)

Appendix

A Proofs and Related Material

A.1 Proof of Theorem 5

Recall the definition of $\overline{T}$:

$$\overline{T} = \{\alpha_p(s') \mid \exists s \in \gamma_p(\overline{R}_k) : s \overset{deq}{\to} s'\} \ . \tag{9}$$

Thm. 5 *If $\overline{R}_k = \overline{R}_{k-1}$ and $\overline{T} \subseteq \overline{R}_k$, then for any $K \geq k$, $\overline{R}_K = \overline{R}_k$.*

Proof: we need two lemmas.

Lem. 6 *Given $\overline{R}_k = \overline{R}_{k-1}$ and $\overline{T} \subseteq \overline{R}_k$, we in fact have $(\overline{R}_k)' \subseteq \overline{R}_k$, for the **full** best abstract transformer image of $\overline{R}_k$,*

$$(\overline{R}_k)' = \{\alpha_p(s') \mid \exists s \in \gamma_p(\overline{R}_k) : s \to s'\} \ . \tag{10}$$

Eq. (10) is identical to Eq. (9), except that it permits all CQS transitions in $\to$, not just those due to dequeue actions.

Proof: we show $(\overline{R}_k)' \setminus \overline{R}_k \subseteq \overline{T}$. From this fact and the given $\overline{T} \subseteq \overline{R}_k$, we have $(\overline{R}_k)' \setminus \overline{R}_k \subseteq \overline{R}_k$, which is equivalent to $(\overline{R}_k)' \subseteq \overline{R}_k$.

The above claim is equivalent to $(\overline{R}_k)' \setminus \overline{T} \subseteq \overline{R}_k$, which we now prove. Let $\bar{s}' \in (\overline{R}_k)' \setminus \overline{T}$, i.e. $\bar{s}' = \alpha_p(s')$ for some s, s', r such that $s = \gamma_p(\alpha_p(r))$ (hence $\alpha_p(s) = \alpha_p(r)$), $r \in R_k$, and $s \overset{loc}{\to} s'$ or $s \overset{!}{\to} s'$ (local or transmit action). Since $\alpha_p(s) = \alpha_p(r)$, concrete states s and r agree in all machines' local states, and in the prefixes of length $p+1$ of all machines' queues. We now attempt to execute on r the action that takes s to s'. We distinguish what that action is:

Case $s \overset{loc}{\to} s'$: the internal action is executable on r (it does not depend on any queue) and leads to a successor state r' such that $\alpha_p(r') = \alpha_p(s')$. From $r \in R_k$ we have $r' \in R_k$ (R_k = concrete reachability fixpoint), and thus $\bar{s}' = \alpha_p(s') = \alpha_p(r') \in \alpha_p(R_k) = \overline{R}_k$.

Case $s \overset{!}{\to} s'$: the transmit action is also executable on r, since it depends only on the local state (on which s and r agree), **and, under bounded semantics, on the current size of the queue**: we must make sure the queue the action applies to is not full (of size k). This is the only place where we need the condition $\overline{R}_k = \overline{R}_{k-1}$: it ensures that in fact $r \in R_{k-1}$, hence all queues in r have at most $k-1$ events. The enqueue action can proceed; the rest of the proof is as in the previous case.

Observe that the above argument does not apply to dequeue actions: it is not guaranteed that the states s' and r' obtained after applying the dequeue are equivalent, i.e. that $\alpha_p(s') = \alpha_p(r')$, because the exact events in the abstracted parts of the queues are unknown and may differ.

Lem. 7 *Given $\overline{R}_k = \overline{R}_{k-1}$ and $(\overline{R}_k)' \subseteq \overline{R}_k$, we have for any $K \geq k$, $\overline{R}_K = \overline{R}_k$.*

Proof: by induction on K. The claim holds for $K = k$. Now suppose $\overline{R}_K = \overline{R}_k$; we prove $\overline{R}_{K+1} = \overline{R}_k$, which is equivalent to $\overline{R}_{K+1} = \overline{R}_K$. Since $\overline{R}_{K+1} \supseteq \overline{R}_K$, it suffices to show that $\overline{R}_{K+1} \subseteq \overline{R}_K$.

To this end, let $a \in \overline{R}_{K+1}$, i.e. $a = \alpha_p(s_a)$ for some $s_a \in R_{K+1}$. We show: for all states s along the path π that reaches s_a, $\alpha_p(s) \in \overline{R}_K$; call this claim `(*)`. In particular, then, $a = \alpha_p(s_a) \in \overline{R}_K$.

To show `(*)`, we induct on the length of path π. The initial state belongs to R_l for every l, so its abstraction belongs to $\overline{R}_K$. Let now s along π be such that $\alpha_p(s) \in \overline{R}_K$. Since $\overline{R}_K = \overline{R}_k$, we have $s \in \gamma_p(\overline{R}_k)$. By Eq. (10), the successor s' of s along π satisfies $\alpha_p(s') \in (\overline{R}_k)' \subseteq \overline{R}_k = \overline{R}_K$. □

Proof of Thm. 5:

From $\overline{R}_k = \overline{R}_{k-1}$, $\overline{T} \subseteq \overline{R}_k$ and Lem. 6 we conclude $(\overline{R}_k)' \subseteq \overline{R}_k$.

From $\overline{R}_k = \overline{R}_{k-1}$, $(\overline{R}_k)' \subseteq \overline{R}_k$ and Lem. 7 we conclude the claim in Thm. 5. □

A.2 Construction of the LTS for Abstract Queue $\overline{\mathcal{Q}}$

This construction is required for Lem. 10 (App. A.3). We formalize it as follows. If $\overline{\mathcal{Q}} = \varepsilon$, we let $S = \{s_0\}$, $T = \emptyset$, $L(s_0) = \{\varepsilon\}$. Otherwise, let $\overline{\mathcal{Q}} = e_0 \ \ldots \ e_{p-1} \mid e_p \ \ldots \ e_{z-1}$. We first define two separate LTS, M_p and M_s, for prefix and suffix, resp., and then conjoin them to get M:

$$
\begin{aligned}
M_p = (S_p, T_p, L_p): \quad & S_p = \{s_i : 0 \leq i < p\} \ , \\
& T_p = \{(s_i, s_{i+1}) : 0 \leq i < p-1\} \ , \\
& L_p(s_i) = \{e_i\} \ \text{ for } i\text{: } 0 \leq i < p
\end{aligned}
$$

$$
\begin{aligned}
M_s = (S_s, T_s, L_s): \quad & S_s = \{s_i, s_{i|i+1} : p \leq i < z\} \ \cup \ \{s_z\} \ , \\
& T_s = \{(s_i, s_{i|i+1}), (s_i, s_{i+1}), (s_{i|i+1}, s_{i|i+1}), \\
& \qquad (s_{i|i+1}, s_{i+1}) : p \leq i < z\} \\
& L[s](s_i) = \{e_i\} \ \text{ for } i\text{: } p \leq i < z \ , \\
& T_s(s_{i|i+1}) = \{e_j : p \leq j < i+1\} \ \text{ for } i\text{: } p \leq i < z \ , \\
& L_s(s_z) = \{\varepsilon\} \ .^5
\end{aligned}
\tag{11}
$$

Now we define $M = (S, T, L)$ with

$$
S = S_p \cup S_s \ , \quad T = T_p \cup \{(s_{p-1}, s_p)\} \cup T_s \ , \quad L = L_p \cup L_s \ . \tag{12}
$$

A.3 Proof of Lem. 10

Lem. 10 *Given abstract queue $\overline{\mathcal{Q}}$ over alphabet Σ, let $M = (S, R, L)$ be its LTS.*

$$
\gamma(\overline{\mathcal{Q}}) \quad = \quad \{L(q) \in \Sigma^* \mid \ q \text{ is a complete path from } s_0 \text{ in } M.\} \tag{13}
$$

[5] Note: if $p = z$ (empty suffix), M_s consists only of node s_z (labeled ε) and no edges.

Proof: by induction on $|\overline{\mathcal{Q}}|$. If $\overline{\mathcal{Q}} = \varepsilon$, then $\gamma(\overline{\mathcal{Q}}) = L(\varepsilon) = \{\varepsilon\}$. The only complete path in M is $q = s_0$ with $L(s_0) = \varepsilon$, so $L(q) = \varepsilon$.

Now suppose $\overline{\mathcal{Q}} = \overline{T}.a$, i.e. a is the final symbol of $\overline{\mathcal{Q}}$, and let $M_{\overline{T}}$ be $\overline{T}$'s LTS. $M_{\overline{T}}$ is a sub-LTS (a "prefix" really) of M.

- If a is in the prefix of $\overline{\mathcal{Q}}$, i.e. the suffix of $\overline{\mathcal{Q}}$ is empty, we have

$$\begin{aligned}\gamma(\overline{\mathcal{Q}}) &\overset{\text{(D4)}}{=} \gamma(\overline{T}) \cdot \{a\} \\ &\overset{\text{(IH)}}{=} \{L(t) \in \Sigma^* \mid t \text{ is a complete path from } s_0 \text{ in } M_{\overline{T}}\} \cdot \{a\}\end{aligned}$$

  where "D4" refers to Def. 4, and "IH" denotes the induction hypothesis. Since M equals $M_{\overline{T}}$ extended by an edge to an a-labeled state, with a unique complete path, Eq. (8) follows.
- If a is in the suffix of $\overline{\mathcal{Q}}$, then let Σ_s be the set of suffix symbols of $\overline{\mathcal{Q}}$ (in particular, $a \in \Sigma_s$).

$$\begin{aligned}\gamma(\overline{\mathcal{Q}}) &\overset{\text{(D4)}}{=} \gamma(\overline{T}) \cdot \{a\} \cdot {\Sigma_s}^* \\ &\overset{\text{(IH)}}{=} \{L(t) \in \Sigma^* \mid t \text{ is a complete path from } s_0 \text{ in } M_{\overline{T}}\} \cdot \{a\} \cdot {\Sigma_s}^*\end{aligned}$$

  By the LTS construction in Sec. 5.2, M equals $M_{\overline{T}}$ with an a-labeled state and a Σ_s-labeled state inserted before the right-most state, from which Eq. (8) follows. □

B Additional Material

B.1 Examples for Sec. 5

Ex. 11 *Here are some examples of QuTL formulas and their intuitive meanings. Let $\mathcal{Q}$ be a queue; $a \Rightarrow b$ abbreviates $\neg a \vee b$.*

Satisfaction relation	*Meaning*
$\mathcal{Q} \models \#e \leq 3$	*$\mathcal{Q}$ contains at most 3 e's.*
$\mathcal{Q} \models \mathsf{G}(e_1 \Rightarrow \mathsf{G}\,\neg e_2)$	*In $\mathcal{Q}$, e_1 is never (eventually) followed by e_2.*
$\mathcal{Q} \models \mathsf{F}(\#e < 2)$	*$\mathcal{Q}$ is non-empty.*
$\mathcal{Q} \models \mathsf{G}(\#e \geq 2)$	*$\mathcal{Q}$ is empty.*
$\mathcal{Q} \models \mathsf{G}(\#e \geq 1)$	*$\mathcal{Q}$ is empty or its tail event is an e.*

Ex. 12 *Here are some examples of abstract (non-)satisfaction. As indicated, these assume $p = 2$, i.e. the first two queue events (if any) are unabstracted. Again, $a \Rightarrow b$ abbreviates $\neg a \vee b$.*

$$bb \,|\, ba \quad \models_\alpha \quad \mathsf{G}(a \Rightarrow \mathsf{G}\,\neg b)$$

since there is a concretization, for instance bbba, that satisfies the formula.

$$ac \,|\, b \quad \not\models_\alpha \quad \mathsf{G}(a \Rightarrow \mathsf{X}\,b)$$

since the violation is caused by the prefix of queue (ac), so all *concretizations violate this formula.*

Ex. 13 *Relation* $\models_\alpha$ *does not distribute over conjunction, i.e.* $\overline{\mathcal{Q}} \models_\alpha \phi_1 \wedge \phi_2$ *is not equivalent to* $\overline{\mathcal{Q}} \models_\alpha \phi_1 \wedge \overline{\mathcal{Q}} \models_\alpha \phi_2$*: Let* $\overline{\mathcal{Q}} = a \mid ab$ *and* $\phi = \psi \wedge \neg\psi$ *where* $\psi = (\#a \geq 3)$*. Clearly,* $\overline{\mathcal{Q}} \not\models_\alpha \phi$ *since* $\phi \equiv$ *false. However,* $\overline{\mathcal{Q}} \models_\alpha \psi$ *and* $\overline{\mathcal{Q}} \models_\alpha \neg\psi$ *both hold, witnessed by two distinct concrete queues* $\mathcal{Q}_\psi = aaab$ *and* $\mathcal{Q}_{\neg\psi} = aab$*.*