Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers*

Samik Basu
Department of Computer Science
Iowa State University
Ames, IA 50011
sbasu@iastate.edu

Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
bultan@cs.ucsb.edu

ABSTRACT

Asynchronous communication requires message queues to store the messages that are yet to be consumed. Verification of interactions in asynchronously communicating systems is challenging since the sizes of these queues can grow arbitrarily large during execution. In fact, behavioral models for asynchronously communicating systems typically have infinite state spaces, which makes many analysis and verification problems undecidable. In this paper, we show that, focusing only on the interaction behavior (modeled as the global sequence of messages that are sent, recorded in the order they are sent) results in decidable verification for a class of asynchronously communicating systems. In particular, we present the necessary and sufficient condition under which asynchronously communicating systems with unbounded queues exhibit interaction behavior that is equivalent to their interactions over finitely bounded queues. We show that this condition can be automatically checked, ensuring existence of a finite bound on the queue sizes, and, we show that, the finite bound on the queue sizes can be automatically computed.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal Methods, Model Checking

General Terms

Verification

Keywords

Asynchronous communication, unbounded buffers, verification

1. INTRODUCTION

Asynchronous communication is commonly used in many domains that rely on concurrent and distributed processes, such as operating systems [9], web services [22] and telecommunication systems [1]. In the asynchronous communication paradigm, when a sender peer (i.e., process) sends a message, the message gets stored in a message queue to be consumed by a specific receiver. This message queue is often referred to as the receive queue of the receiver. The receiver consumes messages in the order they are sent (assuming a FIFO communication model). The sender and the receiver do not synchronize to exchange messages.

In an asynchronously communicating system the receive queues can grow arbitrarily large. As a result, asynchronously communicating systems exhibit behaviors over infinite state-spaces. In general, computing reachable states of asynchronously communicating systems with unbounded receive queues is undecidable [6]. This implies that automatic verification of (temporal) properties of asynchronous systems is also undecidable. As a result, the verification problem is addressed either by identifying subclasses for which verification is decidable or by developing sound and incomplete verification techniques [7, 20, 3, 4].

In this paper, we focus on the interactions among the peers, which are sequences of send actions. Each send action appends a message sent by a peer to another peer's receive queue. Note that, receive actions are local to a peer, where a peer consumes a message from the head of its receive queue. We present the necessary and sufficient condition under which interactions among peers in a system (say A) with unbounded receive queues are identical to another system's (say, B) interactions involving the same peers with bounded receive queues. If this condition holds, then A becomes automatically verifiable, since (i) B can be modeled with a finite state-space, hence (ii) B can be automatically verified, and (iii) the verification results of B remain valid for A.

Asynchronous systems exhibit infinite state behavior, when the sender can send unbounded number of messages to some peer which does not have the capability of consuming them or which does not consume at the same rate as the sender is producing messages, thus forcing the receiver's queue to grow in an unbounded fashion. We are not interested in identifying whether the receive queue remains finitely bounded. Instead, our objective is to identify the condition which, when satisfied, ensures that behaviors exhibited in the presence of unbounded receive queues can be represented using behaviors with bounded receive queues. Intuitively, the condition states that if a peer can send un-

*This work is supported in parts by NSF grant CCF-1116836.

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2643016.

bounded number of messages, then the corresponding receiver must include some behavior where it can consume all the messages sent to it without disabling any of its own send actions. We formally describe this condition and its properties in this paper. We show that the condition can be automatically verified by exploring and analyzing finite number of states in the system.

Once the condition is successfully verified, which ensures the existence of some finite bound (say k) on the receive queue size, one can automatically compute such a bound. The process is based on iteratively checking whether the peer interactions with size i receive queues are identical to peer interactions with size $i+1$ receive queues, starting from $i = 1$. This iteration is guaranteed to terminate when $i = k$. This is because the interactions between peers using size i receive queues include the same peers' interactions using receive queues with size less than i [3].

2. BACKGROUND

Peers & Systems. We first present the formal models for asynchronously communicating peers and their interactions [3, 4].

Definition 1 (PEER BEHAVIOR). *A peer behavior (or simply a peer), denoted by $\mathcal{P}$, is a Finite State Machine (M, T, s_0, δ) where M is the union of input (M^{in}) and output (M^{out}) message sets, T is the finite set of states, $s_0 \in T$ is the initial state, and $\delta \subseteq T \times (M \cup \{\epsilon\}) \times T$ is the transition relation.*

A transition $\tau \in \delta$ can be one of the following three types: (1) a send-transition of the form $(t_1, !m_1, t_2)$ which sends out a message $m_1 \in M^{out}$, (2) a receive-transition of the form $(t_1, ?m_2, t_2)$ which consumes a message $m_2 \in M^{in}$ from its input queue, and (3) an ϵ-transition of the form (t_1, ϵ, t_2). We write $t \xrightarrow{a} t'$ to denote that $(t, a, t') \in \delta$.

We will focus on deterministic peer behaviors, where $\forall t_1, t_2 : t \xrightarrow{a} t_1 \wedge t \xrightarrow{a} t_2 \Rightarrow (t_1 = t_2)$. Peer behaviors can be determinized following standard methods for translation of non-deterministic state machines to deterministic ones. Figure 1(a) illustrates some example peers. The initial states are subscripted with 0. Each transition is represented by send or receive actions.

Definition 2 (SYSTEM BEHAVIOR). *A system behavior (or simply a system) over a set of peers $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$, where $\mathcal{P}_i = (M_i, T_i, s_{0i}, \delta_i)$ and $M_i = M_i^{in} \cup M_i^{out}$, is denoted by a state machine (possibly infinite state) $\mathcal{I} = (M, C, c_0, \Delta)$, where M is the set of messages, C is the set of states, c_0 is the initial state, and Δ is the transition relation defined as:*

1. $M = \cup_i M_i$
2. $C \subseteq \mathcal{Q}_1 \times T_1 \times \mathcal{Q}_2 \times T_2 \ldots \mathcal{Q}_n \times T_n$ *such that* $\forall i \in [1..n] : \mathcal{Q}_i \subseteq (M_i^{in})^*$
3. $c_0 \in C$ *such that* $c_0 = (\epsilon, s_{01}, \epsilon, s_{02} \ldots, \epsilon, s_{0n})$; *and*
4. $\Delta \subseteq C \times M \times C$, *for* $c = (Q_1, t_1, Q_2, t_2, \ldots Q_n, t_n)$ *and* $c' = (Q_1', t_1', Q_2', t_2', \ldots Q_n', t_n')$

   (a) $c \xrightarrow{!m} c' \in \Delta$ *if* $\exists i, j \in [1..n] : m \in M_i^{out} \cap M_j^{in}$,
   (i) $t_i \xrightarrow{!m} t_i' \in \delta_i$, *(ii)* $Q_j' = Q_j m$,
   (iii) $\forall k \in [1..n] : k \neq j \Rightarrow Q_k = Q_k'$ *and*
   (iv) $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$

   (b) $c \xrightarrow{?m} c' \in \Delta$ *if* $\exists i \in [1..n] : m \in M_i^{in}$
   (i) $t_i \xrightarrow{?m} t_i' \in \delta_i$, *(ii)* $Q_i = mQ_i'$,
   (iii) $\forall k \in [1..n] : k \neq i \Rightarrow Q_k = Q_k'$ *and*
   (iv) $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$

   (c) $c \xrightarrow{\epsilon} c' \in \Delta$ *if* $\exists i \in [1..n]$
   (i) $t_i \xrightarrow{\epsilon} t_i' \in \delta_i$, *(ii)* $\forall k \in [1..n] Q_k = Q_k'$ *and*
   (iii) $\forall k \in [1..n] : k \neq i \Rightarrow t_k' = t_k$

In the above, a system state is described in terms of local states of the participating peers and their respective receive queues. The transitions describe the evolution of the system from one state to another via send, receive or internal (ϵ) actions. The send action (item 4a) is non-blocking and as a result of the send action, the message sent is added to the tail of the receive queue of the receiver (4a-ii). The receive (item 4b) is blocking because a receiver can only make a move on a receive action if message to be consumed is present at the head of the receive queue (4b-ii). The epsilon-labeled transition (item 4c) is presented to allow for internal actions in the peers; internal actions simply change the local state of the peer executing it and do not directly affect any other peers.

Figure 1(b) presents a partial view of a system $\mathcal{I}$. Each configuration in $\mathcal{I}$ is denoted by a tuple capturing the local state of the peers and the state of the corresponding receive queues. For instance, initially, all the receive queues are empty, denoted by [ ]. After $\mathcal{P}_1$ sends a and loops back to s_0, the receive queue of $\mathcal{P}_3$ gets updated to contain a, denoted by $[a]$. Whenever a receive action is performed by a peer, it consumes the "matching" message present at the head of its receive queue (leftmost element inside $[\ldots]$). Once a message is consumed, it is removed from the queue.

Definition 3 (K-BOUNDED SYSTEM). *A k-bounded system (denoted by $\mathcal{I}_k$) is a system where the receive queue length for any peer is at most k. The k-bounded system behavior is, therefore, defined by augmenting condition 4(a) in Definition 2 to include the condition $|Q_j| < k$, where $|Q_j|$ denotes the length of the queue for peer j.*

In k-bounded system $\mathcal{I}_k$ the send actions are blocked when the corresponding receive queue, where the sent message is supposed to be buffered, is full (i.e., it already contains k messages pending to be consumed by the receiver). Therefore, $\mathcal{I}_k$ has a finite state-space.

Notations. For a system $\mathcal{I} = (M, C, c_0, \Delta)$ with n peers $\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n \rangle$ and a configuration $c = (Q_1, s_1, Q_2, s_2, \ldots, Q_n, s_n)$ of the system, we use

$c\downarrow^{st} = (s_1, s_2, \ldots, s_n)$
projection of configuration to local states

$c\downarrow^{st}_{\mathcal{P}_i} = s_i$ projection of configuration to $\mathcal{P}_i$'s local state

$c\downarrow^{st}_{\overline{\mathcal{P}_i}} = (s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n)$
projection of configuration to local states of all peers except $\mathcal{P}_i$

$c\downarrow^{qu}_{\mathcal{P}_i} = Q_i$ projection of configuration to receive queue of $\mathcal{P}_i$

Send Sequences & Languages. The following concepts form the basis for describing the interaction behavior of the

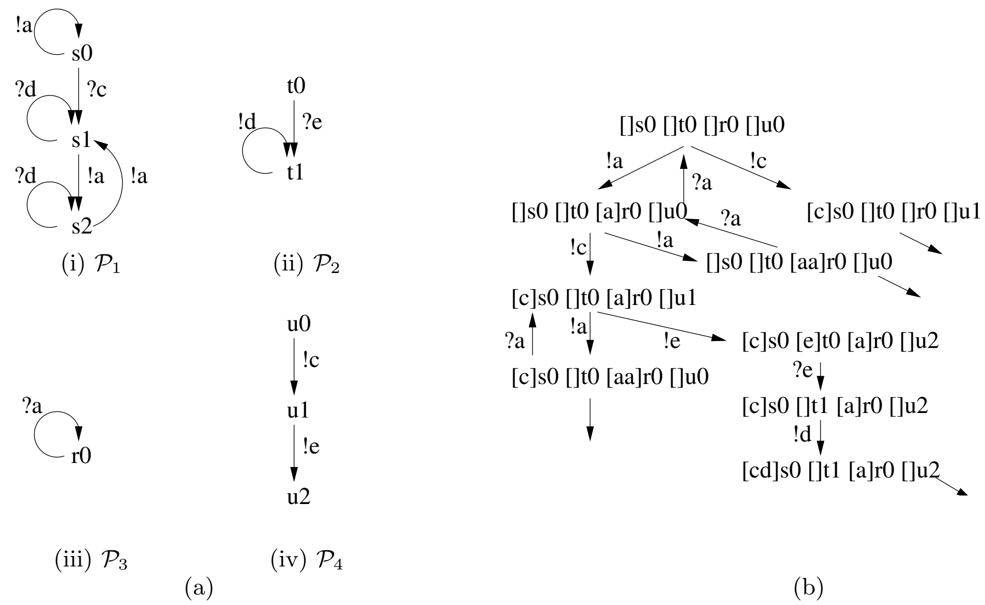


Figure 1: (a) Peers; (b) Partial View of $\mathcal{I}$ composed of $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ and $\mathcal{P}_4$.

system, where the interaction is viewed as messages sent from one peer to another [3].

Definition 4 (Language Equivalence). *The language of a system $\mathcal{I} = (M, C, c_0, F, \Delta)$, denoted by $\mathcal{L}(\mathcal{I})$, is the set of sequences of send actions on any (finite or infinite) path from c_0. Systems $\mathcal{I}$ and $\mathcal{I}'$ are language equivalent if and only if $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}')$.*

Going back to the example in Figure 1(b), some of the example sequences of send actions that belong to $\mathcal{L}(\mathcal{I})$ are as follows: (i) sequence of a's: a aa, …, $aaa\ldots$; and (ii) $a^*ce(a|d)^\omega$, where $*$ represents zero or more occurrences, $|$ represents OR, and ω represents infinite repetition.

Proposition 1. $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}) \Leftrightarrow \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$

Proof. The proof follows the same outline as the proof for synchronizability in Theorem 1 in [3], where the proof was done for a specific value $k = 0$ (synchronous systems, when the peers move in lock-step).

Given the systems X and Y, we say that $\mathcal{L}(X) \subseteq \mathcal{L}(Y)$, if $\forall \omega \in \mathcal{L}(X)$, either ω is subsequence of some $\omega' \in \mathcal{L}(Y)$ or $\omega \in \mathcal{L}(Y)$.

To Prove: $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}) \Rightarrow \mathcal{L}(\mathcal{I}_{k+1}) = \mathcal{L}(\mathcal{I})$, note that, $\forall i \geq 1 : \mathcal{L}(\mathcal{I}_i) \subseteq \mathcal{L}(\mathcal{I}_{i+1})$ and $\mathcal{L}(\mathcal{I}_i) \subseteq \mathcal{L}(\mathcal{I})$. This is because, $\mathcal{I}_{i+1}$ (as well as $\mathcal{I}$) can mimic any send sequence present in $\mathcal{I}_i$. The receive queue size of $\mathcal{I}_{i+1}$ (and $\mathcal{I}$) is larger than that for $\mathcal{I}_i$, which allows the former to avoid blocking of some send actions that are blocked in the latter. Therefore, based on $\subseteq$-relation, $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$ implies that $\mathcal{L}(\mathcal{I}_{k+1}) = \mathcal{L}(\mathcal{I})$, because $\mathcal{L}(\mathcal{I}_k) \subseteq \mathcal{L}(\mathcal{I}_{k+1}) \subseteq \mathcal{L}(\mathcal{I})$.

To Prove: $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \Rightarrow \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$, we first show that $\forall k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \Rightarrow \forall i \geq k : \mathcal{L}(\mathcal{I}_i) = \mathcal{L}(\mathcal{I}_{i+1})$. This means that increasing receive queue size beyond k does not have any impact on the behavior in terms of send sequence. Therefore, $\forall i \geq k : \mathcal{L}(\mathcal{I}_i) = \mathcal{L}(\mathcal{I})$.

We will proceed with proof-by-contradiction. Given that for some k, $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$, **assume** that there exists $n > k+1$ such that $\mathcal{L}(\mathcal{I}_{k+1}) \neq \mathcal{L}(\mathcal{I}_n)$, i.e., $\mathcal{L}(\mathcal{I}_{k+1}) \subset \mathcal{L}(\mathcal{I}_n)$. Therefore, there exists a finite length *witness* where both $\mathcal{I}_n$ and $\mathcal{I}_{k+1}$ have a path over the same sequence of send actions such that the path eventually leads to a state from where $\mathcal{I}_n$ can perform a send action which is not possible in $\mathcal{I}_{k+1}$. In the following, we will consider paths with $\Longrightarrow$-transitions such that $\stackrel{!m}{\Longrightarrow}$ represents a sequence of transitions containing zero of more transitions over actions in $\{\epsilon\} \cup M^{\text{in}}$ and a single transition over $!m$.

Consider that such a path with l send actions is

$$t^n_0 \stackrel{!m_1}{\Longrightarrow} t^n_1 \stackrel{!m_2}{\Longrightarrow} \ldots \stackrel{!m_l}{\Longrightarrow} t^n_l \text{ in } \mathcal{I}_n \tag{1}$$

and the corresponding path in $\mathcal{I}_{k+1}$ that cannot mimic the above sequence after l send actions is

$$t^{k+1}_0 \stackrel{!m_1}{\Longrightarrow} t^{k+1}_1 \stackrel{!m_2}{\Longrightarrow} \ldots \stackrel{!m_l}{\Longrightarrow} t^{k+1}_l \text{ in } \mathcal{I}_{k+1} \tag{2}$$

such that $\forall j \in [0..l] :\ t^n_j = t^{k+1}_j$.

In the above paths, assume that t^n_l is capable of realizing $\stackrel{!m'}{\Longrightarrow}$ which is not possible from t^{k+1}_l, i.e., at t^{k+1}_l the peer (say $\mathcal{P}$) which is responsible for consuming m' is not ready to move on any receive action and its message queue is full (contains $k+1$ pending receive action).

As $\mathcal{L}(\mathcal{I}_{k+1}) = \mathcal{L}(\mathcal{I}_k)$, there exists a path,

$$t^k_0 \stackrel{!m_1}{\Longrightarrow} t^k_1 \stackrel{!m_2}{\Longrightarrow} \ldots \stackrel{!m_l}{\Longrightarrow} t^k_l \text{ in } \mathcal{I}_k \tag{3}$$

Next, we will show that one can construct a path over the same sequence of l sends ($m_1 m_2 \ldots m_l$) in $\mathcal{I}_{k+1}$ such that the receive queue of $\mathcal{P}$ does not exceed k. Let us consider a path

$$t'^{k+1}_0 \stackrel{!m'_1}{\Longrightarrow} t'^{k+1}_1 \stackrel{!m'_2}{\Longrightarrow} \ldots \stackrel{!m'_l}{\Longrightarrow} t'^{k+1}_l \tag{4}$$

where $\forall i \in [0, l] : t'^{k+1}_i \downarrow^{st}_{\mathcal{P}} = t^k_i \downarrow^{st}_{\mathcal{P}}$ and $t'^{k+1}_i \downarrow^{st}_{\overline{\mathcal{P}}} = t^k_i \downarrow^{st}_{\overline{\mathcal{P}}}$. In other words, in the above path, the peer $\mathcal{P}$ moves as it has moved along the path in (3), while all other peers except $\mathcal{P}$ move as they have moved along the path in (2). Our objective is to show that such a path exists and $\forall i \in [1, l]$: m'_i can be made equal to m_i.

Note that, the states $t_0^{k+1}, t_0^k, t_0'^{k+1}$ are identical as these are initial configuration of the system, i.e., the local states of all peers at these states are identical.

Base case: i=1. If $!m_1$ is an action where the sender or the receiver of the message is not $\mathcal{P}$, then there exists an identical action $!m'_1 = !m_1$ as the states of peers that are not $\mathcal{P}$ are identical in t_0^{k+1} and $t_0'^{k+1}$. The resulting next states of t_0^{k+1} and $t_0'^{k+1}$ are also identical.

If $!m_1$ is an action where the receiver is the peer $\mathcal{P}$, then there exists an identical action $!m'_1 = !m_1$ as the states of peers other than $\mathcal{P}$ are identical in t_0^{k+1} and $t_0'^{k+1}$. Furthermore, as $!m_1$ is allowed at the configuration t_0^k, the peer $\mathcal{P}$ must be capable of consuming a message from its receive queue if the addition of $!m_1$ in its receive queue makes the length of the queue $> k$.

If $!m_1$ is an action from peer $\mathcal{P}$ to some other peer, then there exists an identical action $!m'_1 = !m_1$ as the local state of $\mathcal{P}$ are identical in $t_0'^{k+1}$ and t_0^k.

We can, therefore, construct a matching path of length 1 from $t_0'^{k+1}$ to $t_1'^{k+1}$ where

$$m_1 = m'_1 \text{ and } t_1'^{k+1} \downarrow^{st}_{\mathcal{P}} = t_1^k \downarrow_{\mathcal{P}} \text{ and } t_1'^{k+1} \downarrow^{st}_{\overline{\mathcal{P}}} = t_1^{k+1} \downarrow^{st}_{\overline{\mathcal{P}}}$$

Induction Step. Let $\forall i \leq j : !m'_i = !m_i$ and $t_i'^{k+1} \downarrow^{st}_{\mathcal{P}} = t_i^k \downarrow^{st}_{\mathcal{P}}$ and $t_i'^{k+1} \downarrow^{st}_{\overline{\mathcal{P}}} = t_i^{k+1} \downarrow^{st}_{\overline{\mathcal{P}}}$.

Using the arguments as above, we can prove that $!m'_{j+1} = !m_{j+1}$, and $t_{j+1}'^{k+1} \downarrow^{st}_{\mathcal{P}} = t_{j+1}^k \downarrow^{st}_{\mathcal{P}}$ and $t_{j+1}'^{k+1} \downarrow^{st}_{\overline{\mathcal{P}}} = t_{j+1}^k \downarrow^{st}_{\overline{\mathcal{P}}}$.

Therefore, paths 2 and 4 are over exactly the same sequence of send actions. Note that at configuration $t_l'^{k+1}$ peer $\mathcal{P}$ has a message queue with $\leq k$ pending messages as the local state of $\mathcal{P}$ at this configuration is based on configurations in path 3 of $\mathcal{I}_k$. As a result the configuration $t_l'^{k+1}$ is capable of realizing the action $!m'$. This violates our assumption that the path in Equation 1 is a witness distinguishing $\mathcal{I}_n$ and $\mathcal{I}_{k+1}$.

In other words, no witness distinguishing $\mathcal{L}(\mathcal{I}_{k+1})$ and $\mathcal{I}_n$ can be identified when $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$. Therefore, $\forall k$: $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1}) \Rightarrow \forall i \geq k : \mathcal{L}(\mathcal{I}_i) = \mathcal{L}(\mathcal{I}_{i+1})$.

Temporal Properties. We consider temporal ordering of sends expressed in the logic of Linear Temporal Logic (LTL). For instance, GFa represents the LTL property which is satisfied if along all paths of the system message a is sent infinitely often. On the other hand, $FG\neg a$ represents the LTL property which is satisfied if along all paths of the system message a is sent finitely many times. For details of LTL, refer to [8].

Proposition 2. *Given two systems $\mathcal{I}$ and $\mathcal{I}'$, if $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}')$ then for any LTL property ψ over the send actions, $\mathcal{I}$ satisfies ψ if and only if $\mathcal{I}'$ satisfies ψ.*

The Propositions 1 and 2 form the basis for the verification of systems with unbounded receive queues. In the subsequent sections, we identify the condition under which one can guarantee the existence of k such that the language of a system $\mathcal{I}$ with unbounded receive queues is identical to the language of $\mathcal{I}_k$. Once the existence of k is guaranteed, the value of k can be computed by checking iteratively for equality between $\mathcal{L}(\mathcal{I}_i)$ and $\mathcal{L}(\mathcal{I}_{i+1})$ starting from $i = 1$ (Proposition 1). Finally, the computed $\mathcal{I}_k$ can be used to verify any LTL property over send actions using traditional model checking tools and the verification results will hold for $\mathcal{I}$ as well (Proposition 2).

3. CONDITION FOR EQUIVALENT BOUNDED BUFFER BEHAVIOR

In this section, we present a condition (φ) which, when satisfied in any configuration in the system $\mathcal{I}$, guarantees that the interactions between peers participating in the system *cannot* be represented by interactions in any $\mathcal{I}_k$. We proceed by introducing the concept of unbounded send sequence.

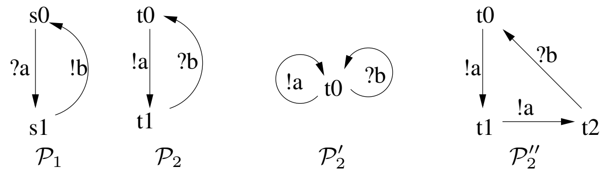
Definition 5 (Unbounded Send Sequence). *Given a system $\mathcal{I} = (M, C, c_0, \Delta)$ over n peers $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$, consider a configuration $c \in C$. A sequence of sends starting from c is unbounded if the following holds.*

- *the composition of peers from their respective local states (t_is) in c produces a cycle with respect to the local states, i.e., same set of t_is are revisited;*
- *in the cycle, none of the peers consume messages from their respective receive buffer corresponding to the configuration c; and*
- *when the cycle is detected, there is some peer whose buffer holds more messages compared to the same peer's buffer at configuration c.*

The first condition ensures that the peer-compositions can send messages in a cycle. The second condition ensures that there is no influence of the history of events, i.e., messages pending in the buffer do not influence the existence of a cycle. This is because, the messages in a buffer, if consumed, may allow for some finite unfoldings of a cycle, but will not ensure its unbounded unfolding. Finally, the third condition ensures that at least one of the peers will have an increase in the number of buffer elements after one unfolding of the cycle.

Example. Figure 2 presents $\mathcal{P}_1$ along with three variants of a second peer ($\mathcal{P}_2$, $\mathcal{P}'_2$ and $\mathcal{P}''_2$). Assume that at a particular configuration c of the system, the peers are at their local states s_0 and t_0. The composition and subsequent detection of cycle (with respect to local states) is presented in the figure. Note that, when the pairs $\mathcal{P}_1$ and $\mathcal{P}''_2$ are considered, there is no unbounded send sequence if at the configuration c, the buffer of $\mathcal{P}_1$ is non-empty or the buffer of $\mathcal{P}''_2$ is non-empty. This may appear to be too restrictive because in the presence of some buffer contents, the peers may still be able to produce send sequences that are unbounded (for instance, when initially $\mathcal{P}_1$ contains a in its receive queue buffer). While configuration ($[a]s0$ $[\ ]t0$) is not classified to produce unbounded send sequence as per Definition 5, the system can proceed (after $\mathcal{P}_1$ consumes a) to configuration ($[\ ]s1$ $[\ ]t0$), which will be classified to produce unbounded send sequences as per Definition 5. In short, removing all history influence does not discard any unbounded send sequences but ensures that no finite send sequence is incorrectly classified as unbounded.

Going back to the Figure 1(b), the start configuration of the system has the local states s_0, t_0, r_0, u_0; the peer $\mathcal{P}_1$ is capable of sending unbounded number of a's to peer $\mathcal{P}_3$.



Compositions for finding unbounded send sequences

$\mathcal{P}_1, \mathcal{P}_2$	[ ]s0[ ]t0$\xrightarrow{!a}$[a]s0[ ]t1$\xrightarrow{?a}$[ ]s1[ ]t1$\xrightarrow{!b}$[ ]s0[b]t1$\xrightarrow{?b}$[ ]s0[ ]t0	No unbounded send sequence
$\mathcal{P}_1, \mathcal{P}_2'$	[ ]s0[ ]t0$\xrightarrow{!a}$[a]s0[ ]t0	Unbounded send sequence
$\mathcal{P}_1, \mathcal{P}_2''$	[ ]s0[ ]t0$\xrightarrow{!a}$[a]s0[ ]t1$\xrightarrow{!a}$[aa]s0[ ]t2$\xrightarrow{?a}$[a]s1[ ]t2$\xrightarrow{!b}$[a]s0[b]t2$\xrightarrow{?b}$[a]s0[ ]t0	Unbounded send sequence

No unbounded send sequence when the buffer of $\mathcal{P}_1$ or $\mathcal{P}_2''$ is non-empty as per Definition 5

Figure 2: Illustration of Unbounded Send Sequences

When the configuration of the system is such that the peers are at their respective local states s_1, t_1, r_0 and u_2, then peers can send unbounded number of a and d messages.

The unbounded sends can potentially make the size of the receivers' queues to grow in an unbounded fashion resulting in an infinite state-space. Our objective is to find the condition under which the size of the receivers' queues can be finitely bounded without changing the behavior of the system as described by the sequences of send actions. In other words, we want to identify the condition which when satisfied guarantees the existence of k such the $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$.

The intuition for checking when/how a finite queue size system ($\mathcal{I}_k$) can replicate all behaviors of unbounded queue size system ($\mathcal{I}$) is as follows. Every unbounded send sequence will result in repetition of some sequence of messages being sent. The receiver peers must be capable of consuming these messages infinitely often ensuring that the receive queues do not have to hold unbounded number of messages. Furthermore, the receive actions of a peer are local and are not visible to the other peers (as the receiver consume messages from its own receive queue). Therefore, it is also necessary that after consuming any subsequence of unbounded sequences of messages, receiver peers should be able to provide the same set of send sequences as they were able to before consuming the messages–ensuring that any ordering of sends between peers that are possible in $\mathcal{I}$ is also possible in $\mathcal{I}_k$. Theorem 1 below presents the necessary and sufficient condition for guaranteeing the existence of $k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$. We proceed by first describing the simulation relation with respect to the send actions. This will be used to ensure that peers while consuming unbounded sequences of messages will not disable any sequence of send actions.

Definition 6 (Send-only Simulation). *Given a finite state machine (M, T, s_0, δ), $t_1 \in T$ is send-simulated by $t_2 \in T$, denoted by $t_1 \prec_! t_2$, if and only if*

$$\forall t_1' : t_1 \xrightarrow{!m} t_1' \;\Rightarrow\; \exists t_2' : t_2 \overset{!m}{\Rightarrow} t_2' \;\wedge\; t_1' \prec_! t_2'$$

where $\overset{!m}{\Rightarrow}$ denotes zero or more ϵ-transitions followed by a $!m$.

The states s_0, s_1 and s_2 in $\mathcal{P}_1$ (Figure 1(a-i)) are related to each other by the $\prec_!$-relation; $s_0 \prec_! s_1 \prec_! s_2 \prec_! s_0$; each can perform unbounded number of $!a$.

Theorem 1. *Given $\mathcal{I} = (M, C, c_0, \Delta)$ over a set of n peers, $\neg[\exists k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})]$ if and only if there exists a configuration $c = (Q_1, s_1, Q_2, s_2, \ldots Q_n, s_n)$ reachable from c_0 such that the condition $\varphi = \varphi_1 \wedge \varphi_2$ holds at c, where*

- φ_1 : *there exists an unbounded send sequence, i.e., there exists a set of peers $\mathcal{PS}$ which can send unbounded number of messages to some peer $\mathcal{P}_i$;*
- φ_2 : *if $\mathcal{P}_i$ can move from s_i to s_i' by only consuming all the pending messages in its receive queue, then $s_i \not\prec_! s_i'$.*

Proof. **To prove**: $\varphi \Rightarrow \neg[\exists k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})]$. Let ω be the sequence of sends leading to configuration c from c_0. As there is a finite number of states in each peer, an unbounded send sequence starting from c results from an unbounded repetition of a sequence (say, σ).

First consider the case where $\mathcal{P}_i$ does not consume all messages in its receive queue Q_i (negation of the antecedent of the implication in φ_2). Therefore, it will require Q_i to be of infinite size to allow storing of messages resulting from unbounded repetition of send sequence σ. That is, in the given path, the send sequence $\omega\sigma\sigma\sigma\ldots$ will require that Q_i size is not finite.

Next consider the case where $\mathcal{P}_i$ can consume all messages in its receive queue Q_i to reach s_i' from s_i and $s_i \not\prec_! s_i'$ (condition φ_2). Let σ' be the sequence of sends possible from s_i that is not possible from s_i'. Consider as before that, ω is the sequence that led to c from c_0 and the unbounded send sequence results from the unbounded repetition of σ. Therefore, $\mathcal{I}$ can have a sequence $\omega\sigma\sigma\ldots\sigma'\ldots$ where the number of times σ can be repeated depends on the size of $\mathcal{P}_i$'s queue at the state s_i.

In summary, in the above paths, it is necessary for the peer $\mathcal{P}_i$ to have infinite receive queue size. Therefore, if φ holds, then there does not exist any k such that $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$.

To prove: $\neg\varphi \Rightarrow \exists k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$. Let there be no reachable configuration from where peers can send unbounded number of messages to some peer $\mathcal{P}_i$ ($\neg\varphi_1$). In this case, the queue size is finite in all configurations of the system. Therefore, there exists a $k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$ where k is the maximum size of the queue in any reachable configuration.

Next consider that, φ_1 holds resulting in unbounded number of sends. Further consider that, $Q_i = m_{i1}m_{i2}\ldots m_{il}$ and there exists some path over the sequence $?m_{i1}?m_{i2}\ldots?m_{il}$ from s_i to s'_i in peer $\mathcal{P}_i$, i.e, along this path all pending messages in Q_i are consumed. Furthermore, s_i and s'_i allows the same sequence of send actions ($s_i \prec_! s'_i$ according to $\neg\varphi_2$). In other words, for all reachable configurations c, from where some peers can send unbounded number of messages, all receiver peers (e.g., $\mathcal{P}_i$) are capable of consuming all messages in their receive queues and are capable of sending the same set of messages. This implies that along all paths of the system, the queues of the peers that may receive unbounded number of messages become empty regularly (within some finite bound). As the peer behaviors are represented by finite state machines, there is a way to restrict the queue size of any peer from growing unboundedly without disabling any send sequence behavior. Therefore, $\neg\varphi$ implies that the send sequences in $\mathcal{I}$ can be replicated by those in $\mathcal{I}_k$ for some finite k. □

Example. Consider the partial view of $\mathcal{I}$ in Figure 1(b). The configuration ($[cd]s_0$ $[]t_1$ $[a]r_0$ $[]u_2$) is one from where peers $\mathcal{P}_1$ and $\mathcal{P}_2$ can produce unbounded number of sends for peers $\mathcal{P}_3$ and $\mathcal{P}_1$, respectively. The peer $\mathcal{P}_3$ can consume a in its receive queue, and remain at r_0; as $r_0 \prec_! r_0$, it satisfies $\neg\varphi_2$. The peer $\mathcal{P}_1$ can consume c followed by d from its receive queue and move to state s_1. Note that $s_0 \prec_! s_1$, i.e., the condition $\neg\varphi_2$ is satisfied. If however, the transition $s_2 \xrightarrow{!a} s_1$ was not present in $\mathcal{P}_1$ (see Figure 1(a-i)), then $s_0 \not\prec_! s_1$ and φ_2 would be satisfied. In that scenario, $\neg[\exists k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})]$ will hold. The sequence of sends where ce followed by any number of d's followed by any number of a's is possible in $\mathcal{I}$ and will not be possible in $\mathcal{I}_k$ for any specific value of k if the transition $s_2 \xrightarrow{!a} s_1$ is absent in $\mathcal{P}_1$.

Example. Consider the peer-pairs illustrated in Figure 2. The pair $\mathcal{P}_1$ and $\mathcal{P}_2$ does not have any unbounded send sequence, i.e., condition φ_1 is not satisfied, which, in turn, implies that the composition of this pair of peers can be represented using finitely bounded receive queues. The pair $\mathcal{P}_1$ and $\mathcal{P}'_2$, on the other hand, has unbounded send sequences. Consider the configuration $[aa]s_0[\,]t0$, where the $\mathcal{P}_1$ has two messages in its receive queue. As presented in the Figure 2, there is an unbounded send sequence from the local states s_0 and t_0, i.e., condition φ_1 is satisfied. Furthermore, the peer $\mathcal{P}_1$ at s_0 cannot consume all the messages $[aa]$ in its receive queue, i.e, condition φ_2 is satisfied. Therefore, the behavior of the peer-pairs $\mathcal{P}_1$ and $\mathcal{P}'_2$ cannot be represented using finitely bounded buffer. The same is true for the peer-pairs $\mathcal{P}_1$ and $\mathcal{P}''_2$.

4. UNBOUNDED TO EQUIVALENT BOUNDED BEHAVIOR

Our objective is to present an algorithm that can automatically verify the condition φ in Theorem 1 for all possible configurations in $\mathcal{I}$. Two problems need to be addressed to realize such an algorithm: (a) identifying whether a set of peers in a reachable configuration can generate unbounded number of sends (see Section 4.1) and (b) exploring *sufficient* (finite) number of configurations in the system to check φ (see Section 4.2).

4.1 Configurations with Unbounded Send Sequence

An unbounded send sequence requires that at least one of the peers must have a cycle or loop in its behavior. In order to check whether a set of peers $\mathcal{PS}$ at a configuration c can potentially send unbounded number of sends to some peer, we deploy the following method. Consider some subset of peers $\mathcal{PS} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$ and the local states of peers in $\mathcal{PS}$ at c are $s_1, s_2, \ldots, s_n$.

1. For each $\mathcal{P}_i \in \mathcal{PS}$, find the strongly connected components (SCC$_i$) involving the corresponding local state s_i.
2. Compose the SCCs starting from s_is as per Definition 2 with additional constraints that
   (a) any send action meant for $\mathcal{P} \notin \mathcal{PS}$ are buffered but never consumed,
   (b) all transitions depending on the inputs from $\mathcal{P} \notin \mathcal{PS}$ are permanently blocked,
   (c) all transitions depending on consuming messages from receive queue at configuration c are blocked.
3. If there exists a path in the composition such that
   (a) it ends with a configuration where all the peers $\mathcal{P}_i$ in $\mathcal{PS}$ loop back to state s_i and
   (b) at least one of the buffers has more pending messages when compared to the same at the configuration c,

   then it is said to exhibit unbounded send sequence as per Definition 5.

Conditions 1 requires SCCs for possibly producing unbounded send sequences. Condition 2 removes any external influence (including history influence). Note that, the state-space resulting from composition can be unbounded; however, for the purpose of identifying unbounded send sequence, it is sufficient to check composition paths where each peer is allowed to move at most L steps, where L is the total number of transitions in all SCCs. If none of the composition paths satisfy the conditions 3(a) and 3(b), then no extensions of any of the paths will satisfy those conditions. The proof of the above statement is straightforward. First, the longest cycle in any peer $\mathcal{P}_i \in \mathcal{PS}$ (at state s_i) is of the order of the size of the SCC. Second, L will allow each peer to go through its respective SCC (if possible), which is sufficient to identify any unbounded sends from peers in $\mathcal{PS}$.

4.2 Algorithm for Exploring $\mathcal{I}$ & Verifying φ

In this section, we focus on exploring sufficient (finite) number of configurations in the $\mathcal{I}$ (finite number of times) and verifying the condition φ (see Theorem 1). Algorithm EXPLORE describes such exploration.

EXPLORE essentially visits configurations in $\mathcal{I}$ in a depth-first fashion starting from the initial configuration c_0. It carries two important information sets per depth-first exploration path: (a) the set *Visited* of visited configurations projected onto local states of the peers along with the action

Algorithm 1 Depth-first exploration for checking φ

1: **procedure** $\textsc{Explore}(c, m, \mathit{Visited}, \mathit{VObl})$
2: **if** `Receivers`:=$\textsc{Cycle}(c) \neq \emptyset$ **then** ▷ *Receivers of unbounded sends*
3: $\mathit{obl} := \emptyset$
4: **for** all $\mathcal{P} \in$ `Receivers` **do**
5: $\mathit{obl}' := \textsc{Verify}(c, \mathcal{P})$
6: **if** $\mathit{obl}' = -1$ **then**
7: **return** false ▷ *Cannot have equivalent finite-buffer behavior*
8: **end if**
9: $\mathit{obl} := \mathit{obl} \cup \mathit{obl}'$ ▷ *Set of all obligations*
10: **end for**
11: **if** $(\mathit{obl} \subseteq \mathit{VObl})$ **then**
12: $\mathit{Visited} := \mathit{Visited} \cup \{(c{\downarrow}^{st}, m)\}$
13: **forall** c' in $c \xrightarrow{m'} c'$ such that $(c'{\downarrow}^{st}, m') \notin \mathit{Visited}$ **do**
14: **if** $\neg\textsc{Explore}(c', m', \mathit{Visited}, \mathit{VObl})$ **then**
15: **return** false ▷ *Terminate exploration*
16: **end if**
17: **end for**
18: **return** true
19: **end if**
20: $\mathit{VObl} := \mathit{VObl} \cup \mathit{obl}$ ▷ *Obligations for depth-first exploration*
21: **end if**
22: **for** all c' in $c \xrightarrow{m'} c'$ **do** ▷ *To all next configurations*
23: **if** $\neg\textsc{Explore}(c', m', \mathit{Visited}, \mathit{VObl})$ **then**
24: **return** false ▷ *Terminate exploration*
25: **end if**
26: **end for**
27: **return** true ▷ *Terminate exploration after the for-loop*
28: **end procedure**

that led to the configuration; and (b) the set VObl of tuples of the form $\langle c{\downarrow}^{st}\!: c'{\downarrow}^{st}\rangle$, where c and c' are configurations in $\mathcal{I}$. The semantics of this tuple is explained below.

The algorithm first checks whether local states of the peers in the configuration c can produce unbounded sequence of sends (Line 2). It uses the Algorithm Cycle (as described in Section 4.1) to identify the unbounded sends, which returns the receivers of the send actions. For each receiver peer, Algorithm Verify is invoked to check whether (a) the peer can move from its current state to some state after consuming all the pending messages in its receive queue and (b) the destination state can send-simulate the current state (Lines 4–5 in Algorithm Verify). If the check is successful, a tuple is returned, which contains local states of the peers in the current configuration and the local states of the peers after the receiver consumes all the pending messages. We use the notation $c{\downarrow}^{st}_{\overline{\mathcal{P}}}$ to denote local states of the peers in c that are not equal to $\mathcal{P}$. We will refer to the returned tuple as (verification) **obligation** based on the fact that for each tuple of the form $\langle c{\downarrow}^{st}\!:\ c'{\downarrow}^{st}\rangle$, at c' some peer has the obligation to consume unbounded send sequences that can be possibly sent to it by some other peers from c'. If the check is unsuccessful, -1 is returned, in which case, Algorithm Explore returns false (Lines 6, 7). In Lines 11–19, Explore checks whether the obligation for the visited configuration has already been computed. If the check is successful, then exploration continues to configurations that are yet to contribute to $\mathit{Visited}$ set (Line 13). Note that, the $\mathit{Visited}$-set only captures the local states (projected from the configuration) and the action that led to the configuration from which the local states are obtained. Therefore, $\mathit{Visited}$ is finite as the peers have finite state-space. Once the for-loop in Lines 13–17 completes, intuitively this implies that the configuration (wrt local states) is revisited, which in turn implies the presence of a cycle in some peers' behavior. Furthermore, as the receiver of the unbounded send sequences do not produce any new obligation tuple, the receiver peer also exhibits a cycle where it can successfully consume the unbounded send sequences.

Algorithm 2 Checking $\neg\varphi_2$

1: **procedure** $\textsc{Verify}(c, \mathcal{P})$
2: $s := c{\downarrow}^{st}_{\mathcal{P}}$
3: $Q := c{\downarrow}^{qu}_{\mathcal{P}}$
4: **if** s moves to s' by only consuming messages in Q and $s \prec_! s'$ **then**
5: **return** $\langle c{\downarrow}^{st}, (c{\downarrow}^{st}_{\overline{\mathcal{P}}}, s')\rangle$
6: **end if**
7: **return** -1
8: **end procedure**

If the check at Line 11 fails, the VObl set for this path of exploration are updated (Line 20) and depth-first exploration continues (Lines 22-26).

Example. Figure 3 illustrates the partial exploration tree of Algorithm Explore for the peers in Figure 1. Consider the path highlighted using dotted line. The order in which they are visited and the corresponding obligations generated when SCCs are considered in the exploration are also presented. The exploration along this path terminates with a true result, as two configurations are visited with the same local states $s_2 t_1 r_0 u_2$ and with identical obligations (last configuration in the highlighted path shown in the partial view). All extensions of this path as per for-loop at Line 13 of Algorithm Explore will return true (e.g., $\mathcal{P}_1$ at s_2 sending $!a$ or consuming $?d$; $\mathcal{P}_2$ at state t_1 sending $!d$; $\mathcal{P}_3$ at state r_0 consuming $?a$). In this example, Algorithm Explore returns true.

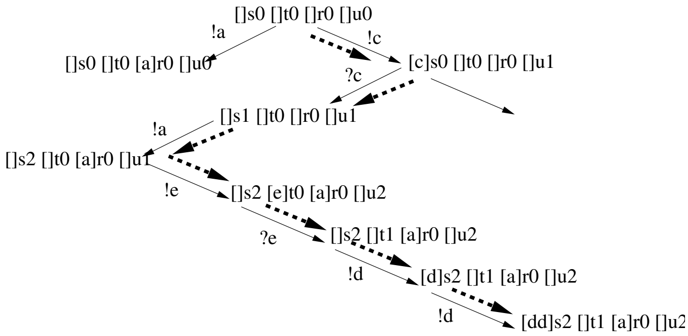
Consider that $\mathcal{P}_4$ in Figure 1 is replaced by $\mathcal{P}'_4$ such that $u_0 \xrightarrow{!e} u_1 \xrightarrow{!c} u_2$. In this case, Algorithm Explore returns false. The reason for returning false is due to the path where $\mathcal{P}_4$ sends e, which is consumed by $\mathcal{P}_2$ followed by sending of message d by $\mathcal{P}_2$. This results in a configuration $([d]s_0\ [\ ]t_1\ [\ ]r_0\ [\ ]u_1)$, which is visited by Algorithm Explore. At this configuration, there a cycle where $\mathcal{P}_2$ from state t_1 can send message d to $\mathcal{P}_1$; however, $\mathcal{P}_1$ at state s_0 cannot consume the messages from its receive queue. The Algorithm Explore returns false, i.e., $\neg[\exists k : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})]$.

4.3 Correctness of Algorithm Explore

Theorem 2. *Given $\mathcal{I} = (M, C, c_0, \Delta)$ over a set of n peers, $\textsc{Explore}(c_0, \epsilon, \emptyset, \emptyset)$ always terminates.*

Proof. The proof follows directly from the finiteness of state-space of each peer behavior. The set $\mathit{Visited}$ is always finite as there are finite combinations of local states of peers. Similarly, the set VObl is finite. □

Theorem 3. *Given $\mathcal{I} = (M, C, c_0, \Delta)$ over a set of n peers, the following holds: all configurations reachable from*



Configuration	**Obligations**	**Visited**	**SCCs**	**Receivers**
$[]s_0\ []t_0\ []r_0\ []u_0$	$\langle s_0t_0r_0u_0, s_0t_0r_0u_0\rangle$	$(s_0t_0r_0u_0, \epsilon)$	$s_0 \xrightarrow{!a} s_0$	$\mathcal{P}_3$
$[c]s_0\ []t_0\ []r_0\ []u_1$	$\langle s_0t_0r_0u_1, s_0t_0r_0u_1\rangle$	$(s_0t_0r_0u_1, !c)$	$s_0 \xrightarrow{!a} s_0$	$\mathcal{P}_3$
$[]s_1\ []t_0\ []r_0\ []u_1$	$\langle s_1t_0r_0u_1, s_1t_0r_0u_1\rangle$	$(s_1t_0r_0u_1, ?c)$	$s_1 \xrightarrow{!a} s_2 \xrightarrow{!a} s_1$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
$[]s_2\ []t_0\ [a]r_0\ []u_1$	$\langle s_2t_0r_0u_1, s_2t_0r_0u_1\rangle$	$(s_2t_0r_0u_1, !a)$	$s_2 \xrightarrow{!a} s_1 \xrightarrow{!a} s_2$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
$[]s_2\ [e]t_0\ [a]r_0\ []u_2$	$\langle s_2t_0r_0u_2, s_2t_0r_0u_2\rangle$	$(s_2t_0r_0u_2, !e)$	$s_2 \xrightarrow{!a} s_1 \xrightarrow{!a} s_2$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
$[]s_2\ []t_1\ [a]r_0\ []u_2$	$\langle s_2t_1r_0u_2, s_2t_1r_0u_2\rangle$	$(s_2t_1r_0u_2, ?e)$	$s_2 \xrightarrow{!a} s_1 \xrightarrow{!a} s_2$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
			$t_1 \xrightarrow{!d} t_1$	$\mathcal{P}_1$
$[d]s_2\ []t_1\ [a]r_0\ []u_2$	$\langle s_2t_1r_0u_2, s_2t_1r_0u_2\rangle$	$(s_2t_1r_0u_2, !d)$	$s_2 \xrightarrow{!a} s_1 \xrightarrow{!a} s_2$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
			$t_1 \xrightarrow{!d} t_1$	$\mathcal{P}_1$
$[dd]s_2\ []t_1\ [a]r_0\ []u_2$	$\langle s_2t_1r_0u_2, s_2t_1r_0u_2\rangle$	$(s_2t_1r_0u_2, !d)$	$s_2 \xrightarrow{!a} s_1 \xrightarrow{!a} s_2$ $s_1 \xrightarrow{?d} s_1\ s_2 \xrightarrow{?d} s_2$	$\mathcal{P}_3$
			$t_1 \xrightarrow{!d} t_1$	$\mathcal{P}_1$

Figure 3: System Configurations explored in Algorithm Explore (partial view)

c_0 satisfy $\neg\varphi$ (see Theorem 1 for definition of φ) if and only if EXPLORE$(c_0, \epsilon, \emptyset, \emptyset)$ *returns true.*

Proof. **To prove:** All configurations satisfy $\neg\varphi$ implies that EXPLORE$(c_0, \epsilon, \emptyset, \emptyset)$ returns true. This follows trivially as EXPLORE algorithm visits subset of the reachable configurations.

To prove: There exists a reachable configuration satisfying φ implies that EXPLORE$(c_0, \epsilon, \emptyset, \emptyset)$ returns false.

Assume that φ holds in some reachable configuration and EXPLORE$(c_0, \epsilon, \emptyset, \emptyset)$ returns true. Therefore, there exists a path π in $\mathcal{I}$ of the form: $c_0, c_1, \ldots, c_m$ such that $\forall i \geq 0$: $c_i \in C$ and at c_m, some subset of peers $\mathcal{PS}$ can produce an unbounded send sequence for some peer $\mathcal{P}$ and $\mathcal{P}$ from the configuration c_m cannot consume all the messages present in its receive queue (negation of the antecedent of the implication in φ_2 in Theorem 1) or after consuming all the messages present in its receive queue $\mathcal{P}$ moves to a state that does not send-simulate the state $c_m \downarrow^{st}_{\mathcal{P}}$ (consequent of the implication in φ_2 in Theorem 1). As per our **assumption**, such a configuration is not visited in the Algorithm EXPLORE.

Without loss of generality, we can assume that there are two peers in the system: peer $\mathcal{P}$ and its environment E. In other words, at the configuration c_m, E is capable of sending unbounded number of messages to $\mathcal{P}$.

As c_m is not visited along the path π in Algorithm EXPLORE, there must exist some configurations c_i and c_j such that

1. $i < j < m$

2. the local states are identical: $c_i \downarrow^{st} = c_j \downarrow^{st}$; and have the same actions leading to the configurations c_i and c_j

3. WLOG assuming E is responsible for unbounded sends to $\mathcal{P}$, the state $c_i \downarrow_{\mathcal{P}}^{st}$ (same as $c_j \downarrow_{\mathcal{P}}^{st}$) can move to the same state s' after consuming all messages in $c_i \downarrow_{\mathcal{P}}^{qu}$ and in $c_j \downarrow_{\mathcal{P}}^{qu}$ (recall that, we are considering deterministic peer behaviors). Furthermore, s' send-simulates $c_i \downarrow_{\mathcal{P}}^{st}$ ($c_j \downarrow_{\mathcal{P}}^{st}$).

In other words, there is a cycle in $\mathcal{P}$ reachable from state $c_i \downarrow_{\mathcal{P}}^{st}$ which can consume unbounded number of messages sent to it from the cycle in E starting from state $c_i \downarrow_{E}^{st}$. Therefore, the repetitions of the behavior of E and $\mathcal{P}$ that causes the sequence $c_i, c_{i+1}, \ldots, c_j$ does not affect the future behavior.

The above statement implies that we can remove this repetition in the behavior of E and generate a shorter path π' ($|\pi'| < |\pi|$) in $\mathcal{I}$ of the form $c_0, c_1, \ldots, c_i, c'_{i+1}, \ldots, c'_n$ such that the local states at c_m and c'_n are identical.

Therefore, the state $c'_n \downarrow_{E}^{st}$ can produce unbounded send sequence for $\mathcal{P}$ and the state $c'_n \downarrow_{\mathcal{P}}^{st}$ of $\mathcal{P}$ cannot consume the messages in its receive queue $c'_n \downarrow_{\mathcal{P}}^{qu}$, or can consume all the messages in its receive queue but ends up in a state which does not send-simulate $c'_n \downarrow_{\mathcal{P}}^{st}$.

If π' is not explored till c'_n by the Algorithm Explore, then the above steps can be repeated to obtain a even smaller path π'' with the same property. Proceeding further, one can construct a short enough path σ where the last configuration is equal to c_m with respect to local states of $\mathcal{P}$ and E and Algorithm Explore visits and checks the last configuration. This contradicts our assumption. □

Theorems 2 and 3 complete the proof of correctness of the Algorithm Explore. The algorithm allows to automatically determine the existence of a k such that $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I})$. Note that, while the existence of k is guaranteed when the algorithm returns true, it does not give us the actual value of k. However, one can easily identify k for which $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_k)$ by successively checking for equality of $\mathcal{L}(\mathcal{I}_i)$ and $\mathcal{L}(\mathcal{I}_{i+1})$ starting from $i = 1$. The process is guaranteed to terminate with a k such that $\mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$ (see Proposition 1). Finally, the properties over the send sequences in $\mathcal{I}$ can be automatically verified by model checking the behavior of $\mathcal{I}_k$ using traditional model checking tools (e.g., Spin [14]).

5. EXPERIMENTS

We have implemented a prototype of our technique in the XSB tabled logic programming environment [23]. XSB's declarative language allows us to easily encode the transition relations of the peers and systems, while XSB's tabling strategy allows for easy computation of least fixed point models, which is directly applied to compute send-simulation using the strategy presented in [5].

We have used three types of specifications in our experiments. The first type is service choreography specifications [10]: MetaConversation is a two peer protocol to decide the initiator of a conversation; ReservationSession is a client-server protocol where the client is requesting a session, waiting for response from the server (cancel, fail or success) and asynchronously sending message to server to cancel the request.

The second type of specifications are the channel contracts from the Singularity OS (an experimental OS developed at Microsoft to allow process isolation) [19]: TpmContract (see Figure 4(a)), TcpContract and KeyboardContract. In this case, the processes interact asynchronously (using FIFO message buffers) following the contract specification.

The third type of specifications are from the example suit of the Spin model checker and includes the Alternating Bit protocol (with a variant that does not include re-sending of messages when the sender times out before receiving acknowledgment) and the Snoopy Cache protocol. In addition, we have also used the specification of a simple StockBroker example from [10] (see Figure 4(b)). Note that there is no equivalent bounded buffer behavior for this example because the first peer can send unbounded sequences of "raw" messages to the second peer, which cannot consume them before sending a "data" message (satisfying condition φ_2 of unboundedness in Theorem 1).

The Table 1 presents the results of our experiments. The second column shows the number of peers in the system, the third column presents a set of tuples, where each tuple (X, Y) denote the number of states X and transitions Y in the peer, the fourth denotes whether or not the behavior with unbounded buffer ($\mathcal{I}$) can be mimicked by some finitely bounded buffer behavior. The fifth column shows the number of configurations explored by our algorithm before terminating with a definitive answer and the sixth column shows the time in seconds. The largest system that we have analyzed in this preliminary study is the Snoopy cache protocol. We encoded the Spin specification (ignoring the buffer restriction) in XSB's input language using logical assertions and rules. Our analysis shows that the protocol behavior cannot be represented using finitely bounded buffers. The time taken for the analysis primarily depends on the size and number of cycles present in the peers and the corresponding systems (experiments are conducted on 1.8GHz Intel Core i7 with 4GB memory). All examples and our prototype implementation are made available at `http://fmg.cs.iastate.edu/project-pages/async/`.

6. RELATED WORK

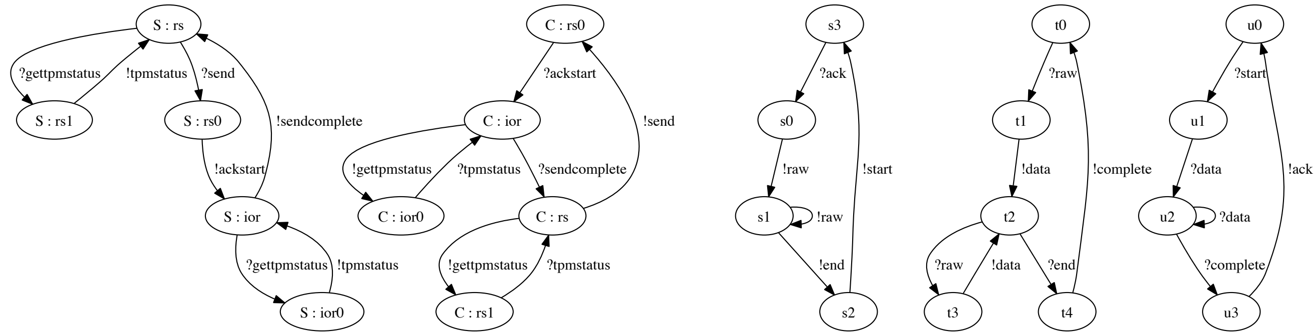
A number of solutions were provided to address the problem of verifying asynchronous systems where peers communicate using unbounded receive queues.

One class of solutions focuses on identifying restricted communication patterns that can render the problem decidable. For instance, in [7], the authors consider half-duplex communication paradigm containing two peers, one where at most one receive queue is non-empty. In [20, 13] the authors relate decidability results with the type of communication topologies (e.g., trees). In this paper, we do not restrict the communication pattern or the communication topology. It should be noted, however, that [20, 13] consider a more expressive peer behavioral model: communicating well-queuing push-down systems (as opposed to our finite state model). We conjecture that our results can be extended to well-queuing push-down systems as well. In the context of push-down systems, recently the authors in [2] present sufficient conditions for decidability of reachability which require the queue contents to be visibly pushdown. In contrast, we focus on algorithmically finding whether a given system behavior can be represented using finite buffers, which, in turn, ensures automatic verifiability.

Another line of work [15, 16] stems from session types, where correct interaction between peers is reduced to a typing problem. The restriction imposed on the communicating peers is that the peers cannot have send and receive

Case Study	# of Peers	# of Peer-States, Transitions	Equivalence	Configs. Explored	Time
MetaConversation	2	(4, 6), (4, 6)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_2)$	13	0.63s
Reservation Session	2	(6, 8), (6, 8)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_2)$	20	0.70s
TpmContract	2	(5, 7), (5, 7)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_2)$	17	0.76s
TcpContract	2	(4, 4), (4, 4)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_1)$	9	0.36s
Keyboard Contract	2	(4, 7), (4, 7)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_1)$	12	0.50s
Stock Broker Example	3	(4, 5), (5, 6), (4, 5)	$\forall k : \mathcal{L}(\mathcal{I}) \neq \mathcal{L}(\mathcal{I}_k)$	3	0.21s
ABP	4	(4, 8), (6, 8), (3, 8), (3, 8)	$\forall k : \mathcal{L}(\mathcal{I}) \neq \mathcal{L}(\mathcal{I}_k)$	37	1.97s
ABP (w/o timeout)	4	(4, 6), (6, 8), (3, 8), (3, 8)	$\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_1)$	21	0.92s
Snoopy Cache	6	(4, 6), (4, 6), (34, 56), (34, 56), (6, 7), (14, 25)	$\forall k : \mathcal{L}(\mathcal{I}) \neq \mathcal{L}(\mathcal{I}_k)$	974	70.52s

Table 1: Summary of Results for the case studies



(a) Start States $S : rs0$ and $C : rs0$

(b) Start States $s0, t0$ and $u0$

Figure 4: (a) Tpm Singularity Contract, (b) Stock Broker example from [10]

actions from the same state. This condition, referred to as the *autonomous condition*, is also used in [11, 12] to identify the sufficient conditions under which the behavior of asynchronously interacting peers can be mimicked by the behavior of the same peers interacting synchronously (i.e., with receive queue size 0). This type of equivalence between asynchronous and synchronous interactions is called *synchronizability* and the system resulting from the interactions is referred to as *synchronizable*. In [3, 4], we prove that synchronizability is decidable. In this paper, we present the conditions under which interactions of peers in the system can be automatically verified; our results hold even for systems that are not synchronizable.

In [17, 18, 21], the authors discuss deadlock freedom and local behavior conformance in MPI programs, where concurrently executing processes interact via message passing. For MPI programs, deadlock-freedom ensures the conformance to desired local behavior. In contrast, we are focusing on global interaction behavior. Additionally, our work does not impose restrictions on the dependencies between send and receive actions, which are natural for the MPI programs.

Our results significantly broaden the scope of automatic verification of asynchronous systems and subsumes the existing results for asynchronously communicating peers when they are represented as finite state machines.

7. CONCLUSIONS

In this paper, we focus on analyzing interactions of asynchronously communicating systems. Since verification of asynchronously communicating systems is undecidable in general, previous results in this area identified subclasses (synchronizable, half-duplex) for which the analysis of interactions is decidable. We significantly improve on these results by presenting a larger decidable class. The key to our approach is identifying if the interactions of a given system (with unbounded receive queues) can be mimicked by the same system when the queues are bounded. We present a prototype implementation and discuss the applicability of our technique in different types of case studies where asynchronous interaction plays an important role.

As part of future work, in addition to focusing on efficient implementation (primary overhead lies in detecting unbounded send sequences) of our technique, we plan to augment existing model checkers with our tool. This will further broaden the application of automated techniques for verifying interaction properties of certain class of asynchronous systems that were previously deemed un-verifiable automatically.

8. REFERENCES

[1] J. Armstrong. Getting Erlang to Talk to the Outside World. In *Proc. ACM SIGPLAN Work. on Erlang*, pages 64–72, 2002.

[2] D. Babić and Z. Rakamarić. Asynchronously communicating visibly pushdown systems. In *FORTE'13: 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems*, volume 7892 of *LNCS*, pages 225 – 242. Springer, 2013.

[3] S. Basu and T. Bultan. Choreography conformance via synchronizability. In *20th International World Wide Web Conference*, 2011.

[4] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In *13th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 56–71, 2012.

[5] S. Basu, M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. M. Verma. Local and symbolic bisimulation using tabled constraint logic programming. In *International Conference on Logic Programming*, 2001.

[6] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[7] G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202:166–190, November 2005.

[8] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[9] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os. In *Proc. 2006 EuroSys Conf.*, pages 177–190, 2006.

[10] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. of the 8th Int. Conf. on Implementation and Application of Automata (CIAA)*, 2003.

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proc. 13th Int. World Wide Web Conf.*, pages 621 – 630, 2004.

[12] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.

[13] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. In *13th Int. Conf. on Foundations of Software Science and Computational Structures (FOSSACS)*, pages 267–281, 2010.

[14] G. Holzman. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Westley, 2003.

[15] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symp. on Programming Languages and Systems (ESOP)*, pages 122–138, 1998.

[16] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *Proceedings of Symposium Principles of Programming Languages*, pages 273–284, 2008.

[17] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Mathematics of Program Construction, (MPC)*, pages 272–285, 1998.

[18] S. F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In *6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 413–429, 2005.

[19] Singularity design note 5 : Channel contracts. singularity rdk documentation (v1.1). http://www.codeplex.com/singularity, 2004.

[20] S. L. Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 299–314, 2008.

[21] S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby. Precise dynamic analysis for slack elasticity: adding buffering without adding bugs. In *17th Euro. MPI Conf. Advances in Message Passing Interface*, pages 152–159, 2010.

[22] Web Service Choreography Description Language (WS-CDL). http://www.w3.org/TR/ws-cdl-10/, 2005.

[23] XSB. http://xsb.sourceforge.net/, 2013.