Synchronizability for Verification of Asynchronously Communicating Systems[⋆]

Samik Basu[1], Tevfik Bultan[2], Meriem Ouederni[3]

[1] Iowa State University, sbasu@iastate.edu
[2] University of California, Santa Barbara, bultan@cs.ucsb.edu
[3] University of Malaga, meriem@lcc.uma.es

Abstract. Message-based communication is an increasingly common interaction mechanism used in concurrent and distributed systems where components interact with each other by sending and receiving messages. It is well-known that verification of systems that use asynchronous message-based communication with unbounded FIFO queues is undecidable even when the component behaviors are expressed using finite state machines. In this paper we show that there is a sub-class of such systems, called synchronizable systems, for which certain reachability properties (over send actions and over states with no pending receives) remain unchanged when asynchronous communication is replaced with synchronous communication. Hence, if a system is synchronizable, then the verification of these reachability properties can be done on the synchronous version of the system and the results hold for the asynchronous case. We present a technique for deciding if a given system is synchronizable. Our results are applicable to a variety of domains including verification and analysis of interactions among processes at the OS level, coordination in service-oriented computing and interactions among distributed programs. In this paper we focus on analysis of channel contracts in the Singularity OS. Our experimental results show that almost all channel contracts in the Singularity OS are synchronizable, and, hence, their properties can be analyzed using synchronous communication semantics.

1 Introduction

The asynchronous message-based communication model has been receiving increasing system support [17, 21, 2, 20] and it is getting increasing attention in a diverse set of areas for handling a variety of issues such as process isolation at the OS level [8], coordination in service-oriented computing [6, 27], and interactions in distributed programs [1]. Unfortunately, in general verification problems are undecidable for such systems since a set of finite-state machines that communicate with unbounded FIFO message queues can simulate Turing Machines [5].

[⋆] The authors thank Gwen Salaün for fruitful discussions on the CADP implementation. This work has been partially supported by the US National Science Foundation grants CCF1117708, CCF1116836, CCF0702758, and project TIN2008-05932 funded by the Spanish Ministry of Innovation and Science and FEDER.

We present a class of asynchronously communicating systems, called *synchronizable* systems, for which certain reachability properties can be verified automatically, and we show that we can automatically check if an asynchronously communicating system is in this class.

Intuitively, an asynchronously communicating system is synchronizable if executing that system with synchronous communication instead of asynchronous communication preserves its behaviors. We focus on two types of behaviors: 1) the sequences of messages that are sent, and 2) the set of reachable configurations where message queues are empty, i.e., configurations with no pending receives. If a system is synchronizable, then we can check properties about its message sequences or about the reachability of its global configurations with empty message queues, using the synchronous version of the system. Since we are focusing on systems where component behaviors are specified using finite state machines, the synchronous version of the system has a finite state space and its properties can be verified using well-known model checking techniques.

The important question is: *is it possible to check synchronizability automatically*? In this paper we show the following: A system is synchronizable if and only if the behaviors for the synchronous version of the system and the 1-bounded-asynchronous version of the system are equivalent with respect to sent messages and reachable configurations with empty message queues. The 1-bounded-asynchronous version corresponds to the case where all message queues are replaced with queues of size one (hence, if there is an unconsumed message in a message queue any send action to that queue blocks until message is consumed). Since both synchronous and 1-bounded asynchronous versions of a system have finite state space, the equivalence check of their behavior, and therefore, synchronizability check, can be done automatically.

In order to demonstrate the practical value of our results, we have developed a prototype implementation leveraging CADP toolbox [12] and have applied our approach to analyzing channel contracts in Singularity OS [8, 16]. A channel contract is a state machine that specifies the allowable ordering of messages exchanged between processes in the Singularity OS. In this paper we show that almost all of the channel contracts in Singularity OS are synchronizable, hence, their reachability properties can be automatically verified.

2 Motivation: Singularity Channel Contracts

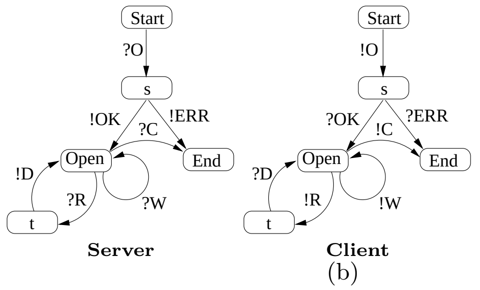
Singularity [23] is an experimental operating system developed by Microsoft Research to explore new approaches to OS design in order to improve the dependability of software systems. Process isolation is a chief design principle of the Singularity OS, where processes are not allowed to share memory with each other or the kernel. All inter-process communication occurs via asynchronous message exchange in bidirectional channels. Communication through Singularity channels corresponds to asynchronous communication via FIFO queues. When a process sends a message through a channel, the message is appended to a message queue. A message that is at the head of a message queue is removed from

```
public contract IoStream {
...
state Start: {
  O? -> {
      OK! -> Open;
      ERR! -> End;
  }
}

state Open: {
  R? -> D! -> Open;
  W? -> Open;
  C? -> End;
}

state End;
...
```

(a)



(b)

O: request to open
R: request to read
W: request to write
D: respond with data
C: request to close

Fig. 1. (a) An example channel contract; and (b) corresponding state machines for the Client and the Server

the message queue when a receive action is executed by the receiving process at the other end of the channel.

In Singularity, channel contracts (written in an extension of C#, called Sing#) specify the allowable ordering of message exchanges between the processes [8, 23]. Figure 1(a) shows a contract governing a channel used by Singularity for communicating between a process (client in this case) and the file server [18]. (The full contract specification also includes the message declarations which are omitted in the figure.) Singularity contracts are written from the perspective of the server, where send actions by the server are appended with `!` to denote communication from the server to the client and receive actions by the server are appended with `?` to denote communication from the client to the server. The contract states that the file server receives a request (`O`) for opening a file and it responds with either `OK` or `ERR`; the destination states are `Open` or `End`. In the `Open` state, the file server can either receive a read request (`R`), a write request (`W`) or a close request (`C`). In the first case, the server responds with the data from the opened file and the destination state of the contract remains `Open`; in the second case, the destination state also remains `Open`; and in the final case, the destination state becomes `End`. The behaviors of the client and the server constructed on the basis of this contract are presented in Figure 1(b). Each local configuration in the client and the server is annotated with the state of the contract; note that there are two temporary/transient states `s` and `t`.

Verification Objectives: Properties of Interest. There are several questions that are of interest in this setting. Does the system obtained from the asynchronously communicating client and server (Figure 1(b)) produces exactly the same behavior (in terms of send actions) as depicted in the channel contract (Figure 1(a))? Does the system conform to some pre-specified desired properties expressed in temporal logic? For instance, a property of interest can be: the `C` (close) send action is eventually followed by a configuration where the client and the server are both at state `End` and their message queues are empty (i.e., there are no pending receives). Another example property can be: every read send action (`R`) is eventually followed by a configuration where the client and the server states are both `Open` and their message queues are empty. These types of properties can be suitably expressed in linear temporal logic.

Verification Challenge. Unfortunately, for finite state processes that communicate asynchronously with unbounded message queues, verification of these types of properties is undecidable in general. Observe that, the system obtained from the asynchronously communicating client and server in Figure 1(b) exhibits behavior with infinite state-space due to existence of potentially unbounded number of `!W` actions from the client before the server consumes via `?W` action.

Our Solution. In this paper, we show that we can automatically check if the asynchronous system under consideration is synchronizable, and, if it is, we can verify the above properties on the synchronized-version of the system using traditional model checking techniques. Verification of properties of asynchronously communicating systems is decidable when the system is synchronizable and we present here the necessary and sufficient condition for synchronizability, which can be efficiently checked using existing equivalence checking techniques that work for systems with finite state-space.

It should be noted that in order to statically determine the amount of memory required for each message buffer, Singularity OS imposes a restriction on channel contracts that bounds the sizes of the message buffers. Such a restriction, therefore, finitizes the behavior of the asynchronous system. Even with such a restriction the results presented in this paper are useful since they allow us to remove the message queues completely during verification. Since the state space of an asynchronously communicating system with bounded queues can be exponential in the size of the queues, our results can be used to avoid state space explosion for such bounded systems. Our experiments show that in fact most of the Singularity channel contracts are synchronizable.

3 Preliminaries

3.1 Behaviors as State Machines

We use finite state machines to describe the behaviors of components or *peers* that asynchronously communicate via messages (sends and receives). The behavior of a system resulting from such communicating peers is described by state machines (with potentially infinite state-space).

Definition 1 (Peer Behavior). *A peer behavior or simply a peer, denoted by $\mathcal{P}$, is a state machine (M, T, s_0, δ) where M is the union of finite input (M^{in}) and finite 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 $m_2 \in M^{in}$, and (3) an ϵ-transition of the form (t_1, ϵ, t_2). We write $t \xrightarrow{a} t'$ to denote that $(t, a, t') \in \delta$.

Figures 2(a, b, c) present state machines representing three communicating peers. The start states (s_0, t_0 and r_0) are denoted by arrows with no source state. Each transition is labeled with the action (send or receive) performed when the peer moves from the source state to the destination state of the transition.

We will consider systems that consist of a finite 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 $M_i = M_i^{\text{in}} \cup M_i^{\text{out}}$, such that $\forall i : M_i^{\text{in}} \cap M_i^{\text{out}} = \emptyset$, $\forall i, j : i \neq j \Rightarrow M_i^{\text{in}} \cap M_j^{\text{in}} = M_i^{\text{out}} \cap M_j^{\text{out}} = \emptyset$.

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*

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* $c_0 = (\epsilon, s_{01}, \epsilon, s_{02} \ldots, \epsilon, s_{0n})$
4. $\Delta \subseteq C \times M \times C$, *and for* $c = (Q_1, t_1, \ldots Q_n, t_n)$ *and* $c' = (Q'_1, t'_1, \ldots Q'_n, t'_n)$
   (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)* $\forall k \in [1..n] : k \neq i \Rightarrow t'_k = t_k$

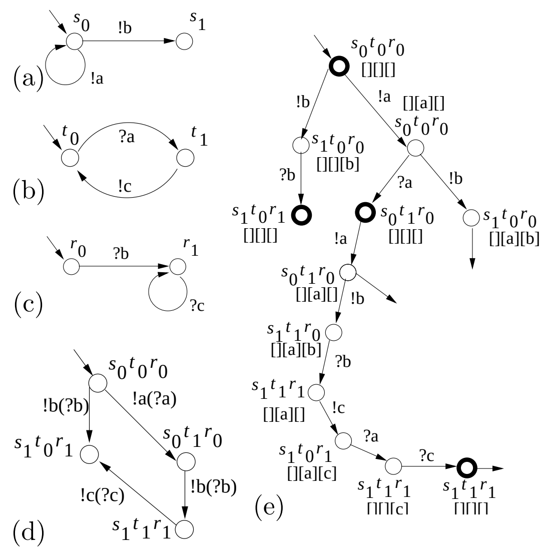


Fig. 2. Peers (a, b, c); Synchronous Behavior (d); (partial view of) Asynchronous Behavior (e).

In the above, Qs describe the message queues associated with each peer in the system. The messages sent to a peer are appended to the tail of its message queue. A peer can perform a receive action if the corresponding message is present at the head of its message queue. After the receive action is performed, the received message is removed from the head of the message queue.

Figure 2(e) presents a snapshot of the behavior of the system realized from the asynchronous composition of the peers shown in Figures 2(a, b, c). Each state is annotated with the local states of the peers and the contents of their message queues. For instance, the state

$s_1t_0r_0$ has the associated message queues $[][][b]$, denoting that the message queue of the third peer has a pending receive b and the message queues of the other peers are empty.

3.2 Verification Objective

We refer to the states where all peers have empty message queues as the *synchronized states* (shown in bold in Figure 2(e)). Note that, start state of the system is a synchronized state (e.g., $s_0t_0r_0$ $[][][]$ in Figure 2(e)). Verification of the above systems may involve checking for properties describing certain desired temporal ordering of send actions and reachability of synchronized states. In this paper, we focus on the following types of global properties:

1. reachability of a synchronized state via a sequence of send actions.
2. existence of a sequence of send actions.

Note that, it is reasonable to ignore the ordering of the receive actions as they are performed *locally* by the peers by consuming messages from their respective message queues. Similarly, it is reasonable to ignore the temporal ordering of states that are not synchronized since these states can be viewed as "transient" states where one or more peers are yet to consume messages and, therefore, have not reacted to the messages sent to them.

4 Synchronizability

We define the notion of send- and synchronized-traces described over the sequence of send actions and synchronized states. Formally,

Definition 3 (Send- & Synchronized-Trace). *A* ***send-trace*** *of a system $\mathcal{I} = (M, C, c_0, \Delta)$ is a sequence of send actions starting from c_0. This is obtained by projecting a trace of $\mathcal{I}$ starting from c_0 to the send actions (by ignoring labels of all the other transitions).*

A ***synchronized-trace*** *of a system, on the other hand, corresponds to a send-trace that starts from c_0 and ends in a synchronized state. A synchronized-trace also includes the start state and the synchronized state reached at the end of the trace (in addition to the sequence of send actions).*

The union of the set of send-traces and the set of synchronized-traces of $\mathcal{I}$ is denoted by $\mathcal{L}(\mathcal{I})$.

$(s_0t_0r_0[][][])aabc(s_1t_1r_1[][][])$ is a synchronized-trace of the system in Figure 2(e). We will denote such a trace as follows: $s_0t_0r_0 \overset{aabc}{\rightsquigarrow} s_1t_1r_1$ (as the message queues of the peers in synchronized states are empty, we omit them). On the other hand, the send-traces of the system include b, a, aa, aab, $aabc$, $aabc\ldots$, etc. We will denote the send-trace as follows $\cdot \overset{a}{\Longrightarrow} \cdot \overset{a}{\Longrightarrow} \cdot \overset{b}{\Longrightarrow} \cdot \overset{c}{\Longrightarrow} \ldots$, where $\overset{m}{\Longrightarrow}$ denotes a transition-sequence containing zero or more receive actions and a single send action $!m$.

Next, we describe synchronizability in terms of a system and its synchronous variant. In the synchronous variant, all peers communicate synchronously, that is, all peers *immediately* consume the messages sent to them.

Definition 4 (Synchronous System Behavior). *The synchronous system behavior containing 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* $\mathcal{I}_0 = (M, C, c_0, \Delta)$ *where*
1. $M = \cup_i M_i$ *2.* $C \subseteq T_1 \times T_2 \ldots \times T_n$
3. $c_0 \in C$ *such that* $c_0 = (s_{01}, s_{02} \ldots, s_{0n})$
4. $\Delta \subseteq C \times M \times C$ *and for* $c = (t_1, t_2, \ldots, t_n)$ *and* $c' = (t'_1, t'_2, \ldots, t'_n)$

1. $c \xrightarrow{!m} c' \in \Delta$ *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)* $t_j \xrightarrow{?m} t'_j \in \delta_j$, *(iii)* $\forall k \in [1..n] : k \neq i \wedge k \neq j \Rightarrow t'_k = t_k$
2. $c \xrightarrow{\epsilon} c' \in \Delta$ *if* $\exists i \in [1..n]$,
(i) $t_i \xrightarrow{\epsilon} t'_i \in \delta_i$, *(ii)* $\forall k \in [1..n] : k \neq i \Rightarrow t'_k = t_k$

Figure 2(d) presents the behavior of the system realized from synchronous composition of peers in Figure 2(a, b, c). Each transition is annotated with the send action; the corresponding receive action which happens synchronously is shown in parenthesis. Note that, in synchronous behavior, there is no pending receives and system states are represented by the tuples of the participating peers' local states. Finally, synchronizability is formally defined as:

Definition 5 (Trace Synchronizability). *The system* $\mathcal{I}$ *over a set of peers* $\langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$ *is said to be trace synchronizable if and only if* $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{I}_0)$, *where* $\mathcal{I}_0$ *is the synchronous system over the same set of peers.*

Verification of properties described in Section 3.2 is decidable for trace synchronizable systems, where such verification can be performed using synchronous version of the system (which does not have message queues and therefore has a finite state-space) using standard model checking techniques. The system in Figure 2(e) is not trace synchronizable as it contains a synchronized trace $s_0t_0r_0 \overset{aabc}{\rightsquigarrow} s_1t_1r_1$ which is not present in its synchronous variant in Figure 2(d).

5 Deciding Trace Synchronizability

We will show that the necessary and sufficient condition for synchronizability involves the equivalence between the synchronous system behavior and the system behavior using bounded asynchronous communication with message queues of size 1 for each participating peer.

Definition 6 (k-bounded System). *For any* $k \geq 1$, *a* k*-bounded system (denoted by* $\mathcal{I}_k$*) is a system where the length of message queue for any peer is at most* k. *The description of* k*-bounded system behavior is, therefore, realized by augmenting condition 4(a) in Definition 2 to include the condition* $|Q_j| < k$, *where* $|Q_j|$ *denotes the number of pending receives in the queue for peer* j.

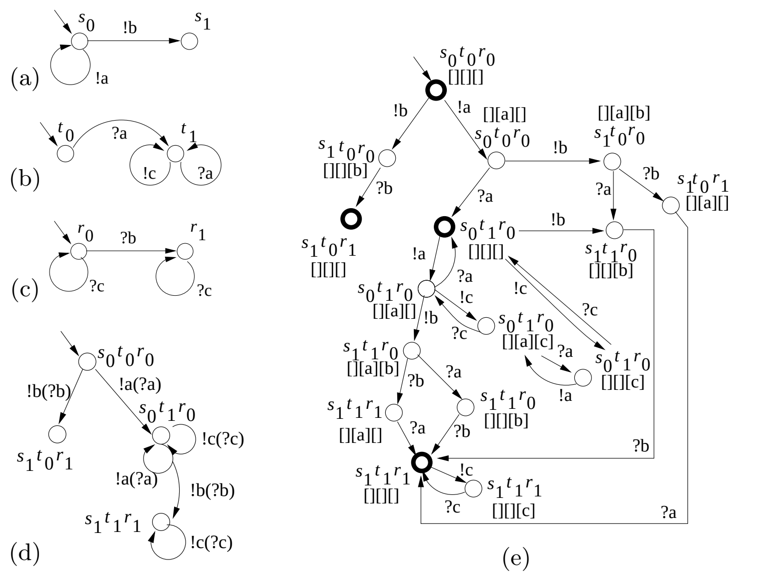


Fig. 3. Peers (a, b, c); Synchronous (d); 1-bounded Asynchronous Behavior (e).

Figure 3(e) shows the 1-bounded system behavior obtained from asynchronously communicating peers in Figures 3(a, b, c). Note that the peer behavior in Figure 3(a) is identical to that in Figure 2(a), while the two peers in Figures 3(b, c) are modified versions of the ones presented in Figures 2(b, c).

Recall that, the synchronous system behavior is denoted by $\mathcal{I}_0$ (Definition 4). In the rest of the section, we will assume that $\mathcal{I}$ and $\mathcal{I}_k$ ($\forall k$) are described over the same set of peers.

Proposition 1. $\forall k \geq 0 : [\mathcal{L}(\mathcal{I}_k) \subseteq \mathcal{L}(\mathcal{I}_{k+1})]$

Proof. For any $k \geq 0$, every move of $\mathcal{I}_k$ can be matched by $\mathcal{I}_{k+1}$ by avoiding the send actions that make the receiving peers' pending receives to exceed k. $\square$

Theorem 1. $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1) \Rightarrow \forall k \geq 0 : \mathcal{L}(\mathcal{I}_k) = \mathcal{L}(\mathcal{I}_{k+1})$.

We prove the theorem by contradiction. We assume that there exists $k > 1$ such that $\mathcal{L}(\mathcal{I}_k) \neq \mathcal{L}(\mathcal{I}_1)$. Therefore, there exists a finite trace (either a send-trace or a synchronized-trace) in $\mathcal{I}_k$ (as $\mathcal{L}(\mathcal{I}_1) \subseteq \mathcal{L}(\mathcal{I}_k)$, by Proposition 1) distinguishing $\mathcal{I}_k$ from $\mathcal{I}_1$. The following Lemmas 1 and 2 contradict the above assumption.

Lemma 1. $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1) \Rightarrow$ *all send-traces in* $\mathcal{I}_k$ *for all* $k > 1$ *are present in* $\mathcal{I}_0$ *and* $\mathcal{I}_1$.

Proof. This lemma follows directly from the result in [3], where we have proved that $\mathcal{I}_0$ and $\mathcal{I}_1$ have the same set of send-traces if and only if the sets of send-traces in $\mathcal{I}_0$ and $\mathcal{I}$ are identical. $\square$

Before proceeding with the proof of Lemma 2, we informally describe the concepts that will be used in the proof. A synchronized-trace is realized by a system that consists of a set of peers, if each peer follows a path in its behavioral state machine that is consistent with the synchronized-trace and reaches a state where its messages queue is empty. In such a path, we will consider the sequence of send and receive actions leading to the local state of the peer with empty message queue. We refer to such a sequence when we say that a peer moves along a trace to realize the synchronized-trace. Similarly, we say that a set of peers move along a trace to realize a synchronized-trace to refer to the sequence of send and receive actions performed by the peers to reach their respective local states describing the synchronized-state of the system. For instance, in Figure 3(e), consider the synchronized trace $s_0t_0r_0 \overset{aabc}{\rightsquigarrow} s_1t_1r_1$. We say that the synchronized trace is realized when the first peer (Figure 3(a)) moves along the trace $(!a!a!b)s_1$; while the other peers (Figures 3(b, c)) move along the traces $(?a?a?b)t_1r_1$ or $(?a?b?a)t_1r_1$.

Lemma 2. $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1) \Rightarrow$ *all synchronized-traces in* $\mathcal{I}_k$ *for all* $k > 1$ *are present in* $\mathcal{I}_0$ *and* $\mathcal{I}_1$.

Proof. Let $t_0^k \overset{\omega}{\rightsquigarrow} t_1^k \dots$ be a synchronized-trace belonging to $\mathcal{I}_k$ where t_0^k is the start state, t_1^k is a synchronized state and ω is a sequence of send actions.

$\mathcal{I}_0$ and $\mathcal{I}_1$ contain the send-trace ω as they contain all send-traces present in $\mathcal{I}_k$, for any $k > 1$ (Lemma 1). As $\mathcal{I}_0$ reaches one or more synchronized states via the send-trace ω, $\mathcal{I}_1$ also reaches the same set of synchronized states after ω (as $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$). We denote this set of states by T^{01}. To prove by contradiction, we assume that t_1^k is different from all the synchronized states in T^{01}. We will contradict this assumption by considering differences between t_1^k and the states in T^{01} in terms of the local states of the peers.

Consider that in $\mathcal{I}_k$, there exists a peer $\mathcal{P}_1$ that moves along a trace A_k and other peers move along a trace B_k to realize the synchronized-trace $t_0^k \overset{\omega}{\rightsquigarrow} t_1^k$. Further, consider that in $\mathcal{I}_1$, the peer $\mathcal{P}_1$ moves along the trace A_1 ($\neq A_k$) and the other peers move along a trace B_1 to realize a synchronized-trace with ω as the sequence of send actions. Let the synchronized state reached in $\mathcal{I}_1$ in this case be $t_1^{01} \in T^{01}$. Let B_1 and B_k eventually lead to identical local states for all peers other than $\mathcal{P}_1$. In short, we are considering the case where t_1^{01} and t_1^k differ only in terms of the states of $\mathcal{P}_1$. Figure 4 illustrates this situation.

We analyze the condition under which, in $\mathcal{I}_1$, the peer $\mathcal{P}_1$ cannot move along A_k when other peers are moving along B_k to realize the send-trace ω. The condition is that in A_k, the peer $\mathcal{P}_1$ has a full message queue (containing a pending receive a) and is trying to send a message m to some other peer; while in B_k, the other peers cannot move without sending a message b to $\mathcal{P}_1$; and $\dots abm \dots$ is present in ω. In other words, the peers cannot move without sending each other messages in a specific order and such sending is not possible as the buffer of $\mathcal{P}_1$ in $\mathcal{I}_1$ in the path A_k is full. That is,

- in A_k, $\mathcal{P}_1$ sends $!m$ when it has some pending receive action (say, a);
- in B_k, some peer sends $!b$ to $\mathcal{P}_1$; and

- $\dots abm \dots$ is present in ω.

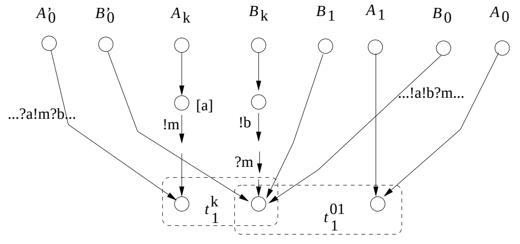
For simplicity, we consider the above scenario with the following assumptions:

Assumption 1: $\mathcal{P}_1$'s message queue contains two pending receives at most once when it moves along the trace A_k in $\mathcal{I}_k$ to realize the given synchronized-trace, and
Assumption 2: t_1^k differs from t_1^{01} in terms of local states of one peer ($\mathcal{P}_1$).

We will prove that the scenario is not possible with the above assumptions and later proceed to prove the same without the assumptions.

As $\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$, the peer $\mathcal{P}_1$ in $\mathcal{I}_0$ moves along a trace A_0 and the other peers move along a trace B_0 to reach t_1^{01} via $\omega = \dots abm \dots$. Note that, A_0 and A_1 end in identical local states for the peer $\mathcal{P}_1$, while B_0, B_1 and B_k end in identical local states for peers other than $\mathcal{P}_1$ according to Assumption 2 above (see Figure 4).

Furthermore, the peers moving along B_0 immediately consume any message sent to them (all sends are immediately received in $\mathcal{I}_0$). This implies that B_0 contains the subsequence $!a!b?m$. Therefore, in $\mathcal{I}_1$, the peer $\mathcal{P}_1$ can move along the trace A_k and other peers can move along B_0 to realize a send sequence $\dots amb \dots$ and reach the synchronized state t_1^k (see Figure 4). As $\mathcal{L}(\mathcal{I}_1) = \mathcal{L}(\mathcal{I}_0)$, this synchronized-trace is also present in $\mathcal{I}_0$. In other words, in $\mathcal{I}_0$, $\mathcal{P}_1$ can move along a trace A_0' and other peers can move along a trace B_0' such that the send sequence $\dots amb \dots$ is realized and the synchronized state t_1^k is reached. Therefore, the destination states for $\mathcal{P}_1$ along the traces A_0' and A_k are identical and the destination states for the peers other than $\mathcal{P}_1$ along the traces B_0', B_0, B_1 and B_k are identical (see Figure 4). Furthermore, there exists a subsequence $?a!m?b$ in A_0' as all sends are immediately consumed by $\mathcal{P}_1$ in $\mathcal{I}_0$.



$\mathcal{P}_1$ Path	Others Path	Send-trace	System
A_k	B_k	$\dots abm \dots$	$\mathcal{I}_k$
A_1	B_1	$\dots abm \dots$	$\mathcal{I}_1$
A_0	B_0	$\dots abm \dots$	$\mathcal{I}_0$
A_k	B_0	$\dots amb \dots$	$\mathcal{I}_1$
A_0'	B_0'	$\dots amb \dots$	$\mathcal{I}_0$
A_0'	B_0	$\dots abm \dots$	$\mathcal{I}_1$

Fig. 4. Proof Schema 1 for Lemma 2

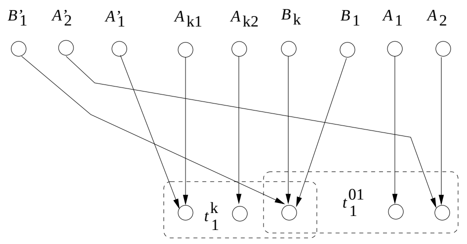
Proceeding further, in $\mathcal{I}_1$, the peer $\mathcal{P}_1$ can move along the trace A_0' and the other peers can move along the trace B_0 to realize the send sequence $\omega = \dots abm \dots$ and reach the destination synchronized state t_1^k (see Figure 4). This is because, in $\mathcal{I}_1$, each peer has a message queue of size 1. This contradicts the assumption that $\mathcal{I}_k$ can reach a synchronized state t_1^k via ω that is not reachable by $\mathcal{I}_1$ via the same send sequence.

Addressing Assumption 1. Recall the two assumptions made for simplifying the arguments of the proof. The arguments hold even when the first assumption is not considered. This is because, if $\mathcal{P}_1$ considers $n > 2$ pending receives in A_k, then we can construct a path for $\mathcal{I}_1$ where $\mathcal{P}_1$ consumes $n-1$ pending receives before the send action $!m$ and reaches the same state as in trace A_k.

Similarly, if $\mathcal{P}_1$ considers $n > 2$ pending receives multiple times along the trace A_k before sending m_0, m_1, etc., we can construct a trace for $\mathcal{P}_1$ in $\mathcal{I}_1$, where $\mathcal{P}_1$ consumes $n-1$ pending receives before performing the send actions $!m_0, !m_1$, etc. and reaches the same destination state as in A_k.

Addressing Assumption 2. Next, we discard the second assumption that t_1^k differ from $t_1^{01}(\in T^{01})$ due to only the local states of $\mathcal{P}_1$. Let the difference between t_1^k and t_1^{01} be due to two peers $\mathcal{P}_1$ and $\mathcal{P}_2$. In $\mathcal{I}_k$, $\mathcal{P}_1$ moves along the trace A_{k1}, $\mathcal{P}_2$ moves along the trace A_{k2}, and peers other than $\mathcal{P}_1$ and $\mathcal{P}_2$ move along the trace B_k. On the other hand, in $\mathcal{I}_1$, $\mathcal{P}_1$ moves along A_1 ($\neq A_{k1}$), $\mathcal{P}_2$ moves along A_2 ($\neq A_{k2}$) and other peers move along B_1 (destination states of these peers in B_1 and B_k are identical). Figure 5 illustrates this scenario.

$\mathcal{I}_k$ has a synchronized-trace with send sequence ω where $\mathcal{P}_1$ moves along A_{k1}, $\mathcal{P}_2$ moves along A_2 and the rest of the peers move along B_1. This synchronized-trace is possible because the size of the message queues of peers in $\mathcal{I}_k$ is greater than those in $\mathcal{I}_1$. Therefore, $\mathcal{I}_k$ and $\mathcal{I}_1$ reach two different synchronized states via send sequence ω, where the destination states differ only in terms of local states of $\mathcal{P}_1$ (see Figure 5). We have already proved that this is not possible. Therefore, there exists a path (with send sequence ω) in $\mathcal{I}_1$ such that $\mathcal{P}_1$ moves along A_1', $\mathcal{P}_2$ moves along A_2' and other peers move along B_1', where the destination states in A_1' and A_{k1} are identical, the destination states in A_2' and A_2 are identical, and the destination states in B_1' and B_1 are identical (see Figure 5).



$\mathcal{P}_1$ Path	$\mathcal{P}_2$ Path	Others Path	Send-trace	System
A_{k1}	A_{k2}	B_k	ω	$\mathcal{I}_k$
A_1	A_2	B_1	ω	$\mathcal{I}_1$
A_{k1}	A_2	B_1	ω	$\mathcal{I}_k$
A_1'	A_2'	B_1'	ω	$\mathcal{I}_1$

Fig. 5. Proof Schema 2 for Lemma 2

Next, consider this newly constructed synchronized-trace for $\mathcal{I}_1$ and the original synchronized-trace for $\mathcal{I}_k$ ($\mathcal{P}_1$ moves along A_{k1}, $\mathcal{P}_2$ moves along A_{k_2} and other peers move along B_k). The synchronized states reached via the same send sequence (ω) differ only in terms of local states of $\mathcal{P}_2$. We have proved this is not possible. Therefore, there exists a synchronized-trace (with send sequence ω) in $\mathcal{I}_1$ such that $\mathcal{P}_1$ moves along A_1'', $\mathcal{P}_2$ moves along A_2'' and others move along B_1'' where the destination states of A_1'' and A_{k1} are identical, the destination states of A_2'' and A_{k2} are identical, and the destination states of B_1'' and B_1 are identical. This contradicts our assumption.

The above arguments also hold when differences in synchronized states are due to local states of more than two peers participating in the system. $\square$

The proof for Theorem 1 directly follows from Lemmas 1 and 2.

Theorem 2. *$\mathcal{L}(\mathcal{I}_0) = \mathcal{L}(\mathcal{I}_1)$ if and only if $\mathcal{I}$ is trace synchronizable.*

Proof. Follows from Theorem 1, Definition 3 and Proposition 1. □

The system in Figure 2(e) is not synchronizable as its 1-bounded asynchronous version is not trace equivalent to its synchronous counterpart (Figure 2(d)). The 1-bounded asynchronous system contains traces (e.g., send trace $\overset{a}{\Longrightarrow}\overset{a}{\Longrightarrow}$ and synchronized trace $s_0t_0r_0 \overset{aabc}{\rightsquigarrow} s_1t_1r_1$) which are absent in the synchronous version. Figure 3(d) and (e) shows the synchronous and 1-bounded asynchronous system realized from the peers in Figures 3(a, b, c). These two systems are trace equivalent and as such the corresponding asynchronous system is trace synchronizable.

Note that, we have proved that synchronizability can be decided by checking the equivalence between two finite-state systems, $\mathcal{I}_0$ and $\mathcal{I}_1$. This can be performed automatically. Once an asynchronous system (with possibly infinite state-state) has been classified as trace synchronizable, we can verify reachability properties over its send actions and synchronized states using the synchronous variant of the system.

6 Experiments with Singularity Channel Contracts

We automated our approach for analyzing Singularity channel contracts by implementing a translator which takes a Singularity channel contract specification as input and generates two LOTOS specifications that correspond to the synchronous and 1-bounded-asynchronous versions of the input contract. Then we use the CADP toolbox [12] to check the equivalence of the synchronous and 1-bounded-asynchronous versions.

Synchronous Model. Given a Singularity channel contract, the state machine of the participating peers (a client and a server) is obtained as follows. For every transition between a state s to a state t, in the contract with label `m!`, a send transition labeled with `m` is added to the state machine of server peer from its local state corresponding to s to its local state corresponding to t; a receive action `m` is added to the state machine of client peer from its local state corresponding to s to its local state corresponding to t. The dual strategy is used for actions of the form `m?` in the contract (the server peer receives `m` sent by the client peer).

The state machines for the peers are encoded in LOTOS using process constructs which allows sequential (ordering), branching (choice) and loop specifications. The synchronous system is constructed from the peer specifications in LOTOS by using the composition operator in LOTOS, which specifies synchronous communication between processes over pre-specified channels.

Asynchronous Model. The LOTOS language does not support asynchronous communication directly. In order to generate the 1-bounded asynchronous model in LOTOS we create a bounded FIFO queue process (which can store at most one message) for each message queue. The FIFO queue process representing the message queue of a peer $\mathcal{P}$ synchronously receives messages from peers sending messages to $\mathcal{P}$, and it synchronously sends these messages to $\mathcal{P}$. The messages sent from the FIFO queue process of peer $\mathcal{P}$ are essentially receive actions by $\mathcal{P}$ which are not considered in send- and synchronized-traces. These actions are, therefore, hidden during the composition process and they become internal transitions (τ-transitions in LOTOS).

Equivalence Checking. After generating the LOTOS specifications for the synchronous and 1-bounded asynchronous models, we generate the two corresponding LTSs using the state space generation tools in the CADP toolbox. During the equivalence check the only visible events are the message send events from any peer since the receive events are hidden. To optimize the equivalence check we reduce the resulting LTS modulo the hidden actions (using the τ-confluence relation). This reduces the transition system without modifying the send- and synchronized-traces of the system. Then we check the equivalence of the reduced LTSs for the synchronous and 1-bounded asynchronous systems. If two LTSs are equivalent, the system (i.e, the system obtained from the peers participating in the given Singularity contract) is synchronizable; otherwise it is not.

The construction of LTSs from LOTOS specifications, the reduction of the LTSs and their equivalence checking are performed automatically using SVL scripts [11] and by using the Reductor and the Bisimulator tools that are part of the CADP toolbox [12].

We applied our approach to 86 channel contracts that are available in the Singularity code base. The size of the synchronous systems obtained from the projected peers of these contracts ranges between 2 to 23 states and 1 to 60 transitions. The size of the 1-bounded asynchronous variant, on the other hand, ranges between 3 to 99 states and 2 to 136 transitions. The time taken to reduce the asynchronous model is on an average 10 secs and the equivalence checking time is on an average 3 secs. We have found that all channel contracts in the Singularity code base are synchronizable except two. The two contracts that fail the synchronizability test are faulty (allow deadlocks, as was previously reported and confirmed by the Singularity developers [24]). Hence, if we ignore these two faulty contracts, *all* channel contracts in the Singularity code base are synchronizable, i.e., their properties concerning the sequence of send actions and reachability of synchronized states can be verified automatically.

7 Related Work

The synchronizability problem was first proposed in [9, 10] in the context of analyzing interactions among web services. Synchronizability definition in these papers only considered sequence of send actions, i.e., send-traces. The synchronizability conditions given in [9, 10] are sufficient but not necessary conditions. One of the synchronizability conditions used in [9, 10] is called autonomous condition, and this condition prevents a process from having a send and a receive transition from the same state. This condition sometimes fails for protocols that are synchronizable. In [3] it is argued that synchronizability analysis can be used for checking the conformance of a set of web services to a given global interaction protocol (called a choreography specification in the web services domain). The synchronizability analysis presented in [3] provides a necessary and sufficient condition for synchronizability when only send-traces are considered. Recent results reported in [4] also build on the results from [3] to show the decidability of the choreography realizability problem.

The synchronizability definitions used in these earlier papers do not correspond to the synchronizability definition we use in this paper since they do not take into account synchronized state reachability. In particular, the main result presented in [3] corresponds to the Lemma 1 from this paper. In this paper we present a non-trivial and important extension to this earlier result and introduce the synchronized-state reachability by proving Lemma 2. This extension allows for verification of reachability properties over send actions and configurations where the message queues are empty. Moreover, the synchronizability analysis presented in [3] is not implemented, whereas we implement the proposed synchronizability analysis and apply it to the Singularity channel contracts.

The work on session types [14, 15] focuses on conformance of an interaction to a predefined protocol and formulates this as a typing problem. The idea is to first define a global type for interaction behavior and then to check if each local peer implementation is "typable" with respect to the global type. If that is the case then the typing rules ensure that when the peers are executed, they conform to the interaction protocol specification that corresponds to the global type. Interestingly, the type system for session types contains an analogue of the autonomous condition from [9, 10] and therefore is more restrictive then the synchronizability condition presented in this paper.

In [7], the authors presented various decidability results for half-duplex asynchronous systems containing two peers, one where at any system state at most one message queue is non-empty. The authors proved that half-duplex systems have a recognizable reachability set which can be computed in polynomial time, and which makes it possible to verify in polynomial time the reachability of system states. The authors proved that determining whether an asynchronous system with two peers is half-duplex is decidable. Finally, the authors showed that systems with more than two peers and participating in pair-wise half-duplex communication can simulate a Turing machines, and therefore, reachability analysis of such systems is undecidable, in general. In this paper, we examined a different subclass of asynchronous systems, namely synchronizable systems. Synchronizability does not require half-duplex communication and is applicable for systems containing more than two peers.

In [25, 13], the authors discuss the type of communication topologies (e.g., trees) that leads to decidability of reachability analysis in communicating systems, including communicating push-down systems. Our results hold for any communication topology. We conjecture that our results also hold for well-queuing push-down systems considered in [25]; a well-queuing push-down system is one where communications occur when the execution stack is empty. We plan to investigate synchronizability of such communicating push-down systems.

In the context of parallel programming, where concurrently executing processes communicate via message passing (MPI programs), several papers (e.g., [19, 22, 26]) discuss the impact of buffering on the behavior in terms of deadlock freedom and conformance to local sequence of actions. Specifically, these works discuss how buffering can lead to deadlock when there are "wildcard" receives (states in the peer behavior where any receive action of that peer is possible), and address the problem of deadlock detection efficiently using partial order reduction [22] or using "happens before" relation [26]. There is one main difference between our work and these earlier results. We are concerned with the global ordering of send actions and reachability of synchronized states as opposed to local ordering of actions. As a result, deadlock-freedom in the synchronous and asynchronous variants does not imply that these variants are trace equivalent (Definition 3). Hence, the premise of the work on MPI programming that deadlock-freedom ensures conformance to desired behavior does not hold in our setting. Additionally, [22] imposes certain MPI domain-specific restrictions regarding dependencies between sends and receives, whereas our approach does not depend on such conditions.

8 Conclusion

In this paper we introduced a notion of synchronizability that identifies a class of asynchronously communicating systems for which the sequences of sent messages and the set of reachable synchronized states (i.e., states with empty-message queues) remain the same when asynchronous communication is replaced with synchronous communication. We showed that synchronizability of a system can be determined by checking the equivalence between its synchronous and 1-bounded asynchronous models. We applied this approach to Singularity channel contracts and our experimental results show that all Singularity channel contracts that are not faulty are synchronizable. Hence, their properties can be verified using the synchronous communication model.

References

1. J. Armstrong. Getting Erlang to talk to the outside world. In *Proc. ACM SIGPLAN Workshop on Erlang*, pages 64–72, 2002.
2. G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proc. 13th Int. Symp. Distributed Computing (DISC)*, pages 1–18, 1999.
3. S. Basu and T. Bultan. Choreography conformance via synchronizability. In *Proc. 20th Int. World Wide Web Conf. (WWW)*, 2011.
4. S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *Proc. 39th Symp. Principles of Programming Languages (POPL)*, 2012.
5. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
6. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming.

7. G. Cécé and A. Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202:166–190, November 2005.
8. 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.
9. 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.
10. X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. Software Eng.*, 31(12):1042–1055, 2005.
11. H. Garavel and F. Lang. SVL: A Scripting Language for Compositional Verification. In *Proc. of FORTE*, pages 377–394, 2001.
12. H. Garavel, R. Mateescu, F. Lang, and W. Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes. In *Proc. 18th Int. Conf. on Computer Aided Verification (CAV)*, 2006.
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. 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.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. 35th Symp. Prin. Programming Languages (POPL)*, pages 273–284, 2008.
16. G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.
17. Java Message Service. http://java.sun.com/products/jms/.
18. J. Larus and G. Hunt. Using the singularity research development kit, 2008. Tutorial, Int. Conf. Arch. Support for Prog. Lang. and OS.
19. R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Mathematics of Program Construction, (MPC)*, pages 272–285, 1998.
20. D. A. Menascé. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing*, 9(2):90–93, 2005.
21. Microsoft Message Queuing Service. http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx.
22. 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.
23. Singularity design note 5 : Channel contracts. singularity rdk documentation (v1.1). http://www.codeplex.com/singularity, 2004.
24. Z. Stengel and T. Bultan. Analyzing singularity channel contracts. In *Proc. 18th Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 13–24, 2009.
25. 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.
26. 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.
27. Web Service Choreography Description Language (WS-CDL). http://www.w3.org/TR/ws-cdl-10/, 2005.