On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony

Ahmed Bouajjani[1], Constantin Enea[1](✉), Kailiang Ji[1], and Shaz Qadeer[2]

[1] IRIF, University Paris Diderot and CNRS, Paris, France
{abou,cenea,jkl}@irif.fr
[2] Microsoft Research, Redmond, USA
qadeer@microsoft.com

Abstract. We address the problem of verifying message passing programs, defined as a set of processes communicating through unbounded FIFO buffers. We introduce a bounded analysis that explores a special type of computations, called k-synchronous. These computations can be viewed as (unbounded) sequences of interaction phases, each phase allowing at most k send actions (by different processes), followed by a sequence of receives corresponding to sends in the same phase. We give a procedure for deciding *k-synchronizability* of a program, i.e., whether every computation is equivalent (has the same happens-before relation) to one of its k-synchronous computations. We show that reachability over k-synchronous computations and checking k-synchronizability are both PSPACE-complete.

1 Introduction

Communication with asynchronous message passing is widely used in concurrent and distributed programs implementing various types of systems such as cache coherence protocols, communication protocols, protocols for distributed agreement, device drivers, etc. An asynchronous message passing program is built as a collection of processes running in parallel, communicating asynchronously by sending messages to each other via channels or message buffers. Messages sent to a given process are stored in its entry buffer, waiting for the moment they will be received by the process. Sending messages is not blocking for the sender process, which means that the message buffers are supposed to be of unbounded size.

Such programs are hard to get right. Asynchrony introduces a tremendous amount of new possible interleavings between actions of parallel processes, and makes it very hard to apprehend the effect of all of their computations. Due

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 678177).

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10982, pp. 372–391, 2018.
https://doi.org/10.1007/978-3-319-96142-2_23

to this complexity, verifying properties (invariants) of such systems is hard. In particular, when buffers are ordered (FIFO buffers), the verification of invariants (or dually of reachability queries) is undecidable even when each process is finite-state [10].

Therefore, an important issue is the design of verification approaches that avoid considering the full set of computations to draw useful conclusions about the correctness of the considered programs. Several such approaches have been proposed including partial-order techniques, bounded analysis techniques, etc., e.g., [4,6,13,16,23]. Due to the hardness of the problem and its undecidability, these techniques have different limitations: either applicable only when buffers are bounded (e.g., partial-order techniques), or limited in scope, or do not provide any guarantees of termination or insight about the completeness of the analysis.

In this paper, we propose a new approach for the analysis and verification of asynchronous message-passing programs with unbounded FIFO buffers, which provides a decision procedure for checking state reachability for a wide class of programs, and which is also applicable for bounded-analysis in the general case.

We first define a bounding concept for prioritizing the enumeration of program behaviors. This concept is guided by our conviction that the behaviors of well designed programs can be seen as successions of *bounded interaction phases*, each of them being a sequence of send actions (by different processes), followed by a sequence of receive actions (again by different processes) corresponding to send actions belonging to the same interaction phase. For instance, interaction phases corresponding to *rendezvous communications* are formed of a single send action followed immediately by its corresponding receive. More complex interactions are the result of exchanges of messages between processes. For instance two processes can send messages to each other, and therefore their interaction starts with two send actions (in any order), followed by the two corresponding receive actions (again in any order). This exchange schema can be generalized to any number of processes. We say that an interaction phase is *k-bounded*, for a given $k > 0$, if its number of send actions is less than or equal to k. For instance rendezvous interactions are precisely 1-bounded phases. In general, we call *k-exchange* any k-bounded interaction phase. Given $k > 0$, we consider that a computation is *k-synchronous* if it is a succession of k-exchanges. It can be seen that, in k-synchronous computations the sum of the sizes of all messages buffers is bounded by k. However, as it will be explained later, boundedness of the messages buffers does not guarantee that there is a k such that all computations are k-synchronous.

Then, we introduce a new bounded analysis which for a given k, considers only computations that are *equivalent* to k-synchronous computations. The equivalence relation on computations is based on a notion of *trace* corresponding to a *happens-before* relation capturing the program order (the order of actions in the code of a process) and the precedence order between sends and their corresponding receives. Two computations are equivalent if they have the same trace, i.e., they differ only in the order of causally independent actions. We show that this analysis is PSPACE-complete when processes have a finite number of states.

An important feature of our bounding concept is that it is possible to decide its completeness for systems composed of finite-state processes, but with unbounded message buffers: For any given k, it is possible to decide whether every computation of the program (under the asynchronous semantics) is equivalent to (i.e., has the same trace as) a k-synchronous computation of that program. When this holds, we say that the program is *k-synchronizable*[1]. Knowing that a program is k-synchronizable allows to conclude that an invariant holds for all computations of the program if no invariant violations have been found by its k-bounded exchange analysis. Notice that k-synchronizability of a program *does not* imply that all its behaviours use bounded buffers. Consider for instance a program with two processes, a producer that consists of a loop of sends, and a consumer that consists of a loop of receives. Although there are computations where the entry buffer of the consumer is arbitrarily large, the program is 1-synchronizable because all its computations are equivalent to computations where each message sent by the producer is immediately received by the consumer.

Importantly, we show that checking k-synchronizability of a program, with possibly infinite-state processes, can be reduced in linear time to checking state reachability under the k-synchronous semantics (i.e., without considering all the program computations). Therefore, for finite-state processes, checking k-synchronizability is PSPACE and it is possible to decide invariant properties without dealing with unbounded message buffers when the programs are k-synchronizable (the overall complexity being PSPACE).

Then, a method for verifying asynchronous message passing programs can be defined, based on iterating k-bounded analyses with increasing value of k, starting from $k = 1$. If for some k, a violation (i.e., reachability of an error state) is detected, then the iteration stops and the conclusion is that the program is not correct. On the other hand, if for some k, the program is shown to be k-synchronizable and no violations have been found, then again the iteration terminates and the conclusion is that the program is correct.

However, it is possible that the program is *not* k-synchronizable for any k. In this case, if the program is correct then the iteration above will not terminate. Thus, an important issue is to determine whether a program is *synchronizable*, i.e., *there exists a k such that the program is k-synchronizable.* This problem is hard, and we believe that it is undecidable, but we do not have a formal proof.

We have applied our theory to a set of nontrivial examples, two of them being presented in Sect. 2. All the examples are synchronizable, which confirms our conviction that non-synchronizability should correspond to an ill-designed system (and therefore it should be reported as an anomaly).

An extended version of this paper with missing proofs can be found at [9].

[1] A different notion of synchronizability has been defined in [4] (see Sect. 8).

2 Motivating Examples

We provide in this section examples illustrating the relevance and the applicability of our approach. Figure 1 shows a *commit protocol* allowing a client to update a memory that is replicated in two processes, called *nodes*. The access to the nodes is controlled by a manager. Figure 2 shows an execution of this protocol. This system is 1-synchronizable, i.e., every execution is equivalent to one where only rendezvous communication is used. Intuitively, this holds because mutually interacting components are never in the situation where messages sent from one to the other are crossing messages sent in the other direction (i.e., the components are "talking" to each other at the same time). For instance, the execution in Fig. 2 is 1-synchronizable because its *conflict graph* (shown in the same figure) is acyclic. Nodes in the conflict graph are matching send-receive pairs (numbered from 1 to 6 in the figure), and edges correspond to the program order between actions in these pairs. The label of an edge records whether the actions related by program order are sends or receives, e.g., the edge from 1 to 2 labeled by RS represents the fact that the receive of the send-receive pair 1

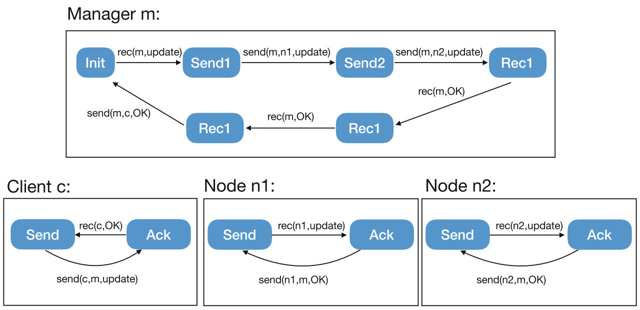


Fig. 1. A distributed commit protocol. Each process is defined as a labeled transition system. Transitions are labeled by send and receive actions, e.g., send(c, m, update) is a send from the client c to the manager m with payload update. Similarly, rec(c, OK) denotes process c receiving a message OK.

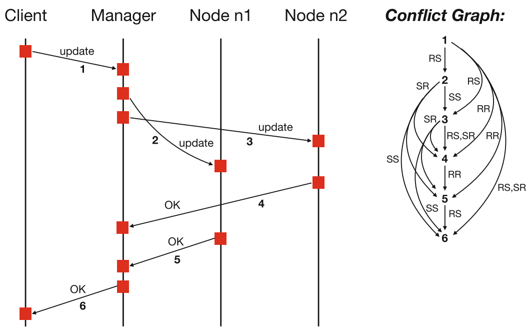


Fig. 2. An execution of the distributed commit protocol and its conflict graph.

is before the send of the send-receive pair 2, in program order. For the moment, these labels should be ignored, their relevance will be discussed in Sect. 5. The conflict graph being acyclic means that matching pairs of send-receive actions are “serializable”, which implies that this execution is equivalent to one where every send is immediately followed by the matching receive (as in rendezvous communication).

Although the message buffers are bounded in all the computations of the commit protocol, this is not true for every 1-synchronizable system. There are asynchronous computations where buffers have an arbitrarily big size, which are equivalent to synchronous computations. This is illustrated by a (family of) computations shown in Fig. 4a of the system modeling an elevator described in Fig. 3 (a simplified version of the system described in [14]). This system consists of three processes: User models the user of the elevator, Elevator models the elevator’s controller, and Door models the elevator’s door which reacts to commands received from the controller. The execution in Fig. 4a models an interaction where the user sends an unbounded number of requests for closing the door, which generates an unbounded number of messages in the entry buffer of Elevator. These computations are 1-synchronizable since they are equivalent to a 1-synchronous computation where Elevator receives immediately every message sent by User. This is witnessed by the acyclicity of the conflict graph of this computation (shown on the right of the same figure). It can be checked that the elevator system without the dashed edge is a 1-synchronous system.

Consider now a slightly different version of the elevator system where the transition from Stopping2 to Opening2 is moved to target Opening1 instead (see the dashed transition in Fig. 3). It can be seen that this version reaches exactly the same set of configurations (tuples of process local states) as the previous one. Indeed, modifying that transition enables Elevator to send a message open to Door, but the latter can only be at StopDoor, OpenDoor, or ResetDoor at this point, and therefore it can (maybe after sending doorStopped and doorOpened) receive at state ResetDoor the message open. However, receiving this message doesn’t change Door’s state, and the set of reachable configurations of the system remains the same. This version of the system is not 1-synchronizable as it is shown in Fig. 4b: once the doorStopped message sent by Door is received by Elevator[2], these two processes can send messages to each other at the same time (the two send actions happen before the corresponding receives). This mutual interaction consisting of 2 parallel send actions is called a 2-*exchange* and it is witnessed by the cycle of size 2 in the execution’s conflict graph (shown on the right of Fig. 4b). In general, it can be shown that every execution of this version of the elevator system has a conflict graph with cycles of size at most 2, which implies that it is 2-synchronizable (by the results in Sect. 5).

[2] Door sends the message from state StopDoor, and Elevator is at state Stopping2 before receiving the message.

3 Message Passing Systems

We define a message passing system as the composition of a set of processes that exchange messages, which can be stored in FIFO buffers before being received (we assume one buffer per process, storing incoming messages from all the other processes). Each process is described as a state machine that evolves by executing send or receive actions. An execution of such a system can be represented abstractly using a partially-ordered set of events, called a *trace*. The partial order in a trace represents the causal relation between events. We show that these systems satisfy *causal delivery*, i.e., the order in which messages are received by a process is consistent with the causal relation between the corresponding sendings.

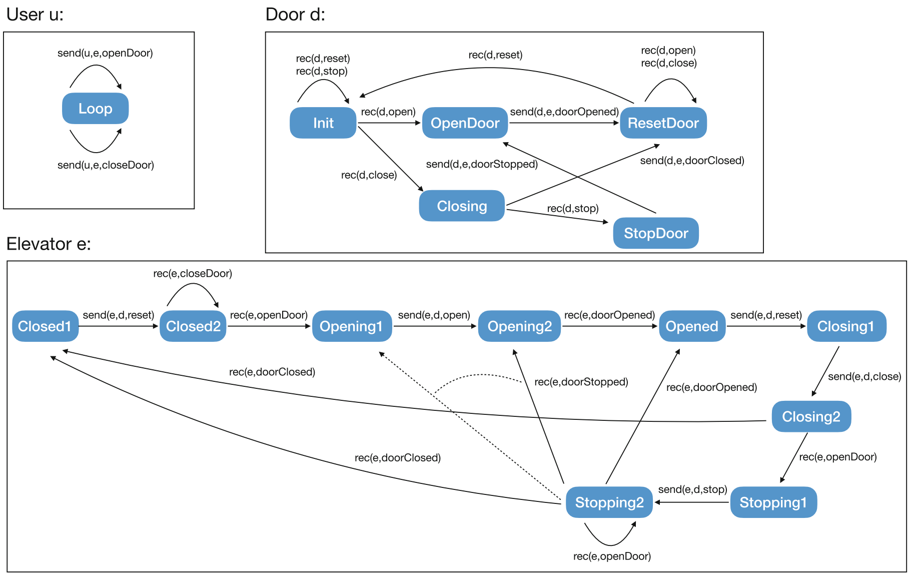


Fig. 3. A system modeling an elevator.

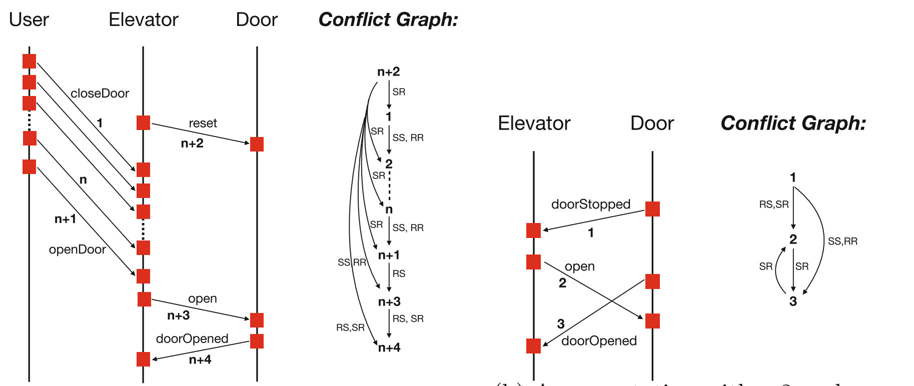
We fix sets $\mathbb{P}$ and $\mathbb{V}$ of process ids and message payloads, and sets $S = \{\text{send}(p,q,v) : p,q \in \mathbb{P}, v \in \mathbb{V}\}$ and $R = \{\text{rec}(q,v) : q \in \mathbb{P}, v \in \mathbb{V}\}$ of *send actions* and *receive actions*. Each send $\text{send}(p,q,v)$ combines two process ids p,q denoting the sender and the receiver of the message, respectively, and a message payload v. Receive actions specify the process q receiving the message, and the message payload v. The process executing an action $a \in S \cup R$ is denoted $\mathsf{proc}(a)$, i.e., $\mathsf{proc}(a) = p$ for all $a = \text{send}(p,q,v)$ or $a = \text{rec}(p,v)$, and the destination q of a send $s = \text{send}(p,q,v) \in S$ is denoted $\mathsf{dest}(s)$. The set of send, resp., receive, actions a of process p, i.e., with $\mathsf{proc}(a) = p$, is denoted by S_p, resp., R_p.

A *message passing system* is a tuple $\mathcal{S} = ((L_p, \delta_p, l_p^0) \mid p \in \mathbb{P})$ where L_p is the set of local states of process p, $\delta_p \subseteq L \times (S_p \cup R_p) \times L$ is a transition relation describing the evolution of process p, and l_p^0 is the initial state of process p. Examples of message passing systems can be found in Figs. 1 and 3.

We fix a set $\mathbb{M}$ of message identifiers, and the sets $S_{id} = \{s_i : s \in S, i \in \mathbb{M}\}$ and $R_{id} = \{r_i : r \in R, i \in \mathbb{M}\}$ of indexed actions. Message identifiers are used to pair send and receive actions. We denote the message id of an indexed send/receive action a by $\mathsf{msg}(a)$. Indexed send and receive actions $s \in S_{id}$ and $r \in R_{id}$ are *matching*, written $s \mapsto r$, when $\mathsf{msg}(s) = \mathsf{msg}(r)$.

A configuration $c = \langle \boldsymbol{l}, \boldsymbol{b} \rangle$ is a vector $\boldsymbol{l}$ of local states along with a vector $\boldsymbol{b}$ of message buffers (sequences of message payloads tagged with message identifiers). The transition relation $\xrightarrow{a}$ (with label $a \in S_{id} \cup R_{id}$) between configurations is defined as expected. Every send action enqueues the message into the destination's buffer, and every receive dequeues a message from the buffer. An execution of a system $\mathcal{S}$ under the asynchronous semantics is a sequence of indexed actions which corresponds to applying a sequence of transitions from the initial configuration (where processes are in their initial states and the buffers are empty). Let $\text{asEx}(\mathcal{S})$ denote the set of these executions. Given an execution e, a send action s in e is called an *unmatched send* when e contains no receive action r such that $s \mapsto r$. An execution e is called *matched* when it contains no unmatched send.

Traces. Executions are represented using traces which are sets of indexed actions together with a *program order* relating every two actions of the same process and a *source* relation relating a send with the matching receive (if any).



(a) A 1-synchronizable execution.

(b) A computation with a 2-exchange.

Fig. 4. Executions of the elevator.

Formally, a *trace* is a tuple $t = (A, po, src)$ where $A \subseteq S_{id} \cup R_{id}$, $po \subseteq A^2$ defines a total order between actions of the same process, and $src \subseteq S_{id} \times R_{id}$ is a relation s.t. $src(a, a')$ iff $a \mapsto a'$. The *trace* $tr(e)$ of an execution e is (A, po, src) where A is the set of all actions in e, $po(a, a')$ iff $\mathsf{proc}(a) = \mathsf{proc}(a')$ and a occurs before a' in e, and $src(a, a')$ iff $a \mapsto a'$. Examples of traces can be found in Figs. 2

and 4. The union of po and src is acyclic. Let $\text{asTr}(\mathcal{S}) = \{tr(e) : e \in \text{asEx}(\mathcal{S})\}$ be the set of traces of $\mathcal{S}$ under the asynchronous semantics.

Traces abstract away the order of non-causally related actions, e.g., two sends of different processes that could be executed in any order. Two executions have the same trace when they only differ in the order between such actions. Formally, given an execution $e = e_1 \cdot a \cdot a' \cdot e_2$ with $tr(e) = (A, po, src)$, where $e_1, e_2 \in (S_{id} \cup R_{id})^*$ and $a, a' \in S_{id} \cup R_{id}$, we say that $e' = e_1 \cdot a' \cdot a \cdot e_2$ is derived from e by a *valid swap* iff $(a, a') \notin po \cup src$. A permutation e' of an execution e is *conflict-preserving* when e' can be derived from e through a sequence of valid swaps. For simplicity, whenever we use the term permutation we mean conflict-preserving permutation. For instance, a permutation of $\text{send}_1(p_1, q, \_)$ $\text{send}_2(p_2, q, \_)$ $\text{rec}_1(q, \_)$ $\text{rec}_2(q, \_)$ is $\text{send}_1(p_1, q, \_)$ $\text{rec}_1(q, \_)$ $\text{send}_2(p_2, q, \_)$ $\text{rec}_2(q, \_)$ and a permutation of the execution $\text{send}_1(p_1, q_1, \_)$ $\text{send}_2(p_2, q_2, \_)$ $\text{rec}_2(q_2, \_)$ $\text{rec}_1(q_1, \_)$ is $\text{send}_1(p_1, q_1, \_)$ $\text{rec}_1(q_1, \_)$ $\text{send}_2(p_2, q_2, \_)$ $\text{rec}_2(q_2, \_)$.

Note that the set of executions having the same trace are permutations of one another. Also, a system $\mathcal{S}$ cannot distinguish between permutations of executions or equivalently, executions having the same trace.

Causal Delivery. The asynchronous semantics ensures a property known as *causal delivery*, which intuitively, says that the order in which messages are received by a process q is consistent with the "causal" relation between them. Two messages are causally related if for instance, they were sent by the same process p or one of the messages was sent by a process p after the other one was received by the same process p. This property is ensured by the fact that the message buffers have a FIFO semantics and a sent message is instantaneously enqueued in the destination's buffer. For instance, the trace (execution) on the left of Fig. 5 satisfies causal delivery. In particular, the messages $v1$ and $v3$ are causally related, and they are received in the same order by $q2$. On the right of Fig. 5, we give a trace where the messages v_1 and v_3 are causally related, but received in a different order by $q2$, thus violating causal delivery. This trace is not valid because the message $v1$ would be enqueued in the buffer of $q2$ before $\text{send}(p, q1, v2)$ is executed and thus, before $\text{send}(q1, q2, v3)$ as well.

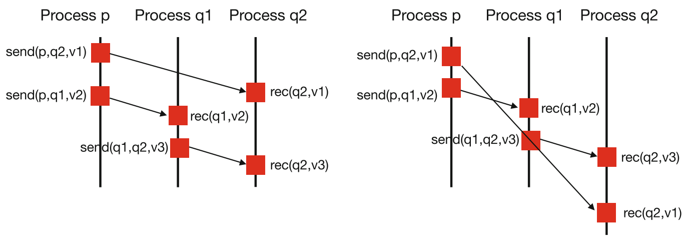


Fig. 5. A trace satisfying causal delivery (on the left) and a trace violating causal delivery (on the right).

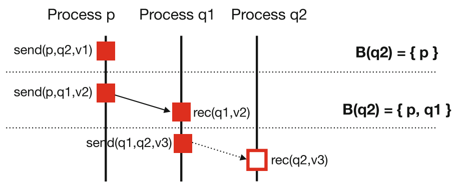


Fig. 6. An execution of the 1-synchronous semantics.

Formally, for a trace $t = (A, po, src)$, the transitive closure of $po \cup src$, denoted by $\leadsto_t$, is called the *causal relation* of t. For instance, for the trace t on the left of Fig. 5, we have that $\text{send}(p, q2, v1) \leadsto_t \text{send}(q1, q2, v3)$. A trace t satisfies *causal delivery* if for every two send actions s_1 and s_2 in A,

$$(s_1 \leadsto_t s_2 \wedge \mathsf{dest}(s_1) = \mathsf{dest}(s_2)) \implies (\nexists r_2 \in A.\ s_2 \mapsto r_2) \vee \\ (\exists r_1, r_2 \in A.\ s_1 \mapsto r_1 \wedge s_2 \mapsto r_2 \wedge (r_2, r_1) \notin po)$$

It can be easily proved that every trace $t \in \text{asTr}(\mathcal{S})$ satisfies causal delivery.

4 Synchronizability

We define a property of message passing systems called *k-synchronizability* as the equality between the set of traces generated by the asynchronous semantics and the set of traces generated by a particular semantics called *k-synchronous*.

The k-synchronous semantics uses an extended version of the standard rendez-vous primitive where more than one process is allowed to send a message and a process can send multiple messages, but all these messages must be received before being allowed to send more messages. This primitive is called *k-exchange* if the number of sent messages is at most k. For instance, the execution $\text{send}_1(p_1, p_2, \_)\ \text{send}_2(p_2, p_1, \_)\ \text{rec}_1(p_2, \_)\ \text{rec}_2(p_1, \_)$ is an instance of a 2-exchange. To ensure that the k-synchronous semantics is prefix-closed (if it admits an execution, then it admits all its prefixes), we allow messages to be dropped during a k-exchange transition. For instance, the prefix of the previous execution without the last receive ($\text{rec}_2(p_1, \_)$) is also an instance of a 2-exchange. The presence of unmatched send actions must be constrained in order to ensure that the set of executions admitted by the k-synchronous semantics satisfies causal delivery. Consider for instance, the sequence of 1-exchanges in Fig. 6, a 1-exchange with one unmatched send, followed by two 1-exchanges with matching pairs of send/receives. The receive action ($\text{rec}(q2, v3)$) pictured as an empty box needs to be disabled in order to exclude violations of causal delivery. To this, the semantics tracks for each process p a set of processes $B(p)$ from which it is forbidden to receive messages. For the sequence of 1-exchanges in Fig. 6, the unmatched $\text{send}(p, q2, v1)$ disables any receive by $q2$ of a message sent by p (otherwise, it will be even a violation of the FIFO semantics of $q2$'s buffer). Therefore, the first 1-exchange results in $B(q2) = \{p\}$. The second 1-exchange (the message from p to $q1$) forbids $q2$ to receive any message from $q1$. Otherwise, this message will be necessarily causally related to $v1$, and receiving it will lead to a violation of causal delivery. Therefore, when reaching send$(q1, q2, v_3)$ the receive rec(q_2, v_3) is disabled because $q1 \in B(q2)$.

k-EXCHANGE

$$\dfrac{\begin{array}{c} e \in S_{id}^* \cdot R_{id}^* \qquad |e| \le 2 \cdot k \\ (\boldsymbol{l}, \boldsymbol{\epsilon}) \xrightarrow{e} (\boldsymbol{l'}, \boldsymbol{b}),\ \text{for some } \boldsymbol{b} \qquad \forall s, r \in e.\ s \mapsto r \implies \mathsf{proc}(s) \notin B(\mathsf{dest}(s)) \\ B'(q) = B(q) \cup \{p : \exists s \in e \cap S_{id}.\ ((\nexists r \in e.\ s \mapsto r) \wedge p = \mathsf{proc}(s) \wedge q = \mathsf{dest}(s)) \\ \vee (\mathsf{proc}(s) \in B(q) \wedge \mathsf{dest}(s) = p)\} \end{array}}{(\boldsymbol{l}, B) \overset{e}{\Rightarrow}_k (\boldsymbol{l'}, B')}$$

Fig. 7. The synchronous semantics. Above, $\boldsymbol{\epsilon}$ is a vector where all the components are ϵ, and $\xrightarrow{e}$ is the transition relation of the asynchronous semantics.

Formally, a configuration $c' = (\boldsymbol{l}, B)$ in the synchronous semantics is a vector $\boldsymbol{l}$ of local states together with a function $B : \mathbb{P} \to 2^{\mathbb{P}}$. The transition relation $\Rightarrow_k$ is defined in Fig. 7. A k-EXCHANGE transition corresponds to a sequence of transitions of the asynchronous semantics starting from a configuration with empty buffers. The sequence of transitions is constrained to be a sequence of at most k sends followed by a sequence of receives. The receives are enabled depending on previous unmatched sends as explained above, using the function B. The semantics defined by $\Rightarrow_k$ is called the k-synchronous semantics.

Executions and traces are defined as in the case of the asynchronous semantics, using $\Rightarrow_k$ for some fixed k instead of $\to$. The set of executions, resp., traces, of $\mathcal{S}$ under the k-synchronous semantics is denoted by $\mathrm{sEx}_k(\mathcal{S})$, resp., $\mathrm{sTr}_k(\mathcal{S})$. The executions in $\mathrm{sEx}_k(\mathcal{S})$ and the traces in $\mathrm{sTr}_k(\mathcal{S})$ are called k-synchronous.

An execution e such that $tr(e)$ is k-synchronous is called k-synchronizable. We omit k when it is not important. The set of executions generated by a system $\mathcal{S}$ under the k-synchronous semantics is prefix-closed. Therefore, the set of its k-synchronizable executions is prefix-closed as well. Also, k-synchronizable and k-synchronous executions are undistinguishable up to permutations.

Definition 1. *A message passing system $\mathcal{S}$ is called* k-synchronizable *when* $\mathrm{asTr}(\mathcal{S}) = \mathrm{sTr}_k(\mathcal{S})$.

It can be easily proved that k-synchronizable systems reach exactly the same set of local state vectors under the asynchronous and the k-synchronous semantics. Therefore, any assertion checking or invariant checking problem for a k-synchronizable system $\mathcal{S}$ can be solved by considering the k-synchronous semantics instead of the asynchronous one. This holds even for the problem of detecting deadlocks. Therefore, all these problems become decidable for finite-state k-synchronizable systems, whereas they are undecidable in the general case (because of the FIFO message buffers).

5 Characterizing Synchronous Traces

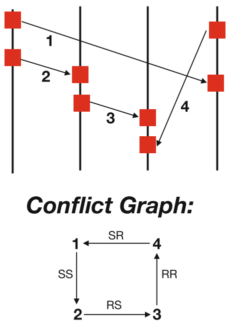


Fig. 8. A trace and its conflict graph.

We give a characterization of the traces generated by the k-synchronous semantics that uses a notion of *conflict-graph* similar to the one used in conflict serializability [27]. The nodes of the conflict graph correspond to pairs of matching actions (a send and a receive) or to unmatched sends, and the edges represent the program order relation between the actions represented by these nodes.

For instance, an execution with an acyclic conflict graph, e.g., the execution in Fig. 2, is "equivalent" to an execution where every receive immediately follows the matching send. Therefore, it is an execution of the 1-synchronous semantics. For arbitrary values of k, the conflict graph may contain cycles, but of a particular form. For instance, traces of the 2-synchronous semantics may contain a cycle of size 2 like the one in Fig. 4(b). More generally, we show that the conflict graph of a k-synchronous trace cannot contain cycles of size strictly bigger than k. However, this class of cycles is not sufficient to precisely characterize the k-synchronous traces. Consider for instance the trace on top of Fig. 8. Its conflict-graph contains a cycle of size 4 (shown on the bottom), but the trace is not 4-synchronous. The reason is that the messages tagged by 1 and 4 must be sent during the same exchange transition, but receiving message 4 needs that the message 3 is sent after 2 is received. Therefore, it is not possible to schedule all the send actions before all the receives. Such scenarios correspond to cycles in the conflict graph where at least one receive is before a send in the program order (witnessed by the edge labeled by RS). We show that excluding such cycles, in addition to cycles of size strictly bigger than k, is a precise characterization of k-synchronous traces.

The *conflict-graph* of a trace $t = (A, po, src)$ is the labeled directed graph $CG_t = \langle V, E, \ell_E \rangle$ where: (1) the set of nodes V includes one node for each pair of matching send and receive actions, and each unmatched send action in t, and (2) the set of edges E is defined by: $(v, v') \in E'$ iff there exist actions $a \in \mathrm{act}(v)$ and $a' \in \mathrm{act}(v')$ such that $(a, a') \in po$ (where $\mathrm{act}(v)$ is the set of actions of trace t corresponding to the graph node v). The label of the edge (v, v') records whether a and a' are send or receive actions, i.e., for all $X, Y \in \{S, R\}$, $XY \in \ell(v, v')$ iff $a \in X_{id}$ and $a' \in Y_{id}$.

A direct consequence of previous results on conflict serializability [27] is that a trace is 1-synchronous whenever its conflict-graph is acyclic. A cycle of a conflict graph CG_t is called *bad* when it contains an edge labeled by RS. Otherwise, it is called *good*. The following result is a characterization of k-synchronous traces.

Theorem 1. *A trace t satisfying causal delivery is k-synchronous iff every cycle in its conflict-graph is good and of size at most k.*

Theorem 1 can be used to define a runtime monitoring algorithm for k-synchronizability checking. The monitor records the conflict-graph of the trace produced by the system and checks whether it contains some bad cycle, or a cycle of size bigger than k. While this approach requires dealing with unbounded message buffers, the next section shows that this is not necessary. Synchronizability violations, if any, can be exposed by executing the system under the *synchronous* semantics.

6 Checking Synchronizability

We show that checking k-synchronizability can be reduced to a reachability problem under the *k-synchronous* semantics (where message buffers are bounded). This reduction holds for arbitrary, possibly infinite-state, systems. More precisely, since the set of (asynchronous) executions of a system is prefix-closed, if a system $\mathcal{S}$ admits a synchronizability violation, then it also admits a *borderline* violation, for which every strict prefix is synchronizable. We show that every *borderline* violation can be "simulated"[3] by the synchronous semantics of an instrumentation of $\mathcal{S}$ where the receipt of exactly one message is delayed (during every execution). We describe a monitor that observes executions of the instrumentation (under the synchronous semantics) and identifies synchronizability violations (there exists a run of this monitor that goes to an error state whenever such a violation exists).

6.1 Borderline Synchronizability Violations

For a system $\mathcal{S}$, a violation e to k-synchronizability is called *borderline* when every strict prefix of e is k-synchronizable. Figure 9(a) gives an example of a borderline violation to 1-synchronizability (it is the same execution as in Fig. 4(b)).

We show that every borderline violation e ends with a receive action and this action is included in every cycle of $CG_{tr(e)}$ that is bad or exceeds the bound k. Given a cycle $c = v, v_1, \ldots, v_n, v$ of a conflict graph CG_t, the node v is called a *critical* node of c when (v, v_1) is an SX edge with $X \in \{S, R\}$ and (v_n, v) is an YR edge with $Y \in \{S, R\}$.

Lemma 1. *Let e be a borderline violation to k-synchronizability of a system $\mathcal{S}$. Then, $e = e' \cdot r$ for some $e' \in (S_{id} \cup R_{id})^*$ and $r \in R_{id}$. Moreover, the node v of $CG_{tr(e)}$ representing r (and the corresponding send) is a critical node of every cycle of $CG_{tr(e)}$ which is bad or of size bigger than k.*

6.2 Simulating Borderline Violations on the Synchronous Semantics

Let $\mathcal{S}'$ be a system obtained from $\mathcal{S}$ by "delaying" the reception of exactly one nondeterministically chosen message: $\mathcal{S}'$ contains an additional process π and exactly one message sent by a process in $\mathcal{S}$ is non-deterministically redirected

[3] We refer to the standard notion of (stuttering) simulation where one system mimics the transitions of the other system.

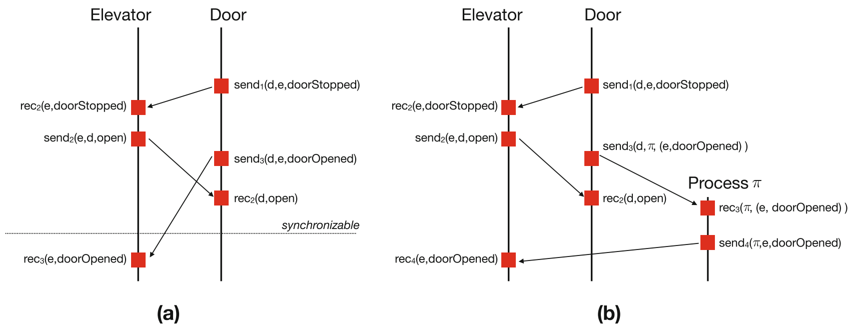


Fig. 9. A borderline violation to 1-synchronizability.

to π[4], which sends it to the original destination at a later time[5]. We show that the synchronous semantics of $\mathcal{S}'$ "simulates" a permutation of every borderline violation of $\mathcal{S}$. Figure 9(b) shows the synchronous execution of $\mathcal{S}'$ that corresponds to the borderline violation in Fig. 9(a). It is essentially the same except for delaying the reception of doorOpened by sending it to π who relays it to the elevator at a later time.

The following result shows that the k-synchronous semantics of $\mathcal{S}'$ "simulates" all the borderline violations of $\mathcal{S}$, modulo permutations.

Lemma 2. *Let* $e = e_1 \cdot \text{send}_i(p,q,v) \cdot e_2 \cdot \text{rec}_i(q,v)$ *be a borderline violation to* k*-synchronizability of* $\mathcal{S}$*. Then,* $\text{sEx}_k(\mathcal{S}')$ *contains an execution* e' *of the form:*

$$e' = e_1' \cdot \text{send}_i(p,\pi,(q,v)) \cdot \text{rec}_i(\pi,(q,v)) \cdot e_2' \cdot \text{send}_j(\pi,q,v) \cdot \text{rec}_j(q,v)$$

such that $e_1' \cdot \text{send}_i(p,q,v) \cdot e_2'$ *is a permutation of* $e_1 \cdot \text{send}_i(p,q,v) \cdot e_2$*.*

Checking k-synchronizability for $\mathcal{S}$ on the system $\mathcal{S}'$ would require that every (synchronous) execution of $\mathcal{S}'$ can be transformed to an execution of $\mathcal{S}$ by applying an homomorphism σ where the send/receive pair with destination π is replaced with the original send action and the send/receive pair initiated by π is replaced with the original receive action (all the other actions are left unchanged). However, this is not true in general. For instance, $\mathcal{S}'$ may admit an execution $\text{send}_i(p,\pi,(q,v))\cdot\text{rec}_i(\pi,(q,v))\cdot\text{send}_j(p,q,v')\cdot\text{rec}_j(q,v')\cdot\text{send}_{i'}(\pi,q,v)\cdot \text{rec}_{i'}(q,v)$ where a message sent after the one redirected to π is received earlier, and the two messages were sent by the same process p. This execution is possible under the 1-synchronous semantics of $\mathcal{S}'$. Applying the homomorphism σ, we get the execution $\text{send}_i(p,q,v) \cdot \text{send}_j(p,q,v') \cdot \text{rec}_j(q,v') \cdot \text{rec}_i(q,v)$ which violates causal delivery and therefore, it is not admitted by the asynchronous semantics

[4] Meaning that every transition labeled by a send action $\text{send}(p,q,v)$ is doubled by a transition labeled by $\text{send}(p,\pi,(q,v))$, and such a send to π is enabled only once throughout the entire execution.

[5] The process π stores the message (q,v) it receives in its state and has one transition where it can send v to the original destination q.

of $\mathcal{S}$. Our solution to this problem is to define a monitor $\mathcal{M}_{causal}$, i.e., a process which reads every transition label in the execution and advances its local state, which excludes such executions of $\mathcal{S}'$ when run under the synchronous semantics, i.e., it blocks the system $\mathcal{S}'$ whenever applying some transition would lead to an execution which, modulo the homomorphism σ, is a violation of causal delivery. This monitor is based on the same principles that we used to exclude violations of causal delivery in the synchronous semantics in the presence of unmatched sends (the component B from a synchronous configuration).

6.3 Detecting Synchronizability Violations

We complete the reduction of checking k-synchronizability to a reachability problem under the k-synchronous semantics by describing a monitor $\mathcal{M}_{viol}(k)$, which observes executions in the k-synchronous semantics of $\mathcal{S}' \,||\, \mathcal{M}_{causal}$ and checks whether they represent violations to k-synchronizability; $\mathcal{M}_{viol}(k)$ goes to an error state whenever such a violation exists.

Essentially, $\mathcal{M}_{viol}(k)$ observes the sequence of k-exchanges in an execution and tracks a conflict graph cycle, if any, interpreting $\mathrm{send}_i(p, \pi, (q, v)) \cdot \mathrm{rec}_i(\pi, (q, v))$ as in the original system $\mathcal{S}$, i.e., as $\mathrm{send}_i(p, q, v)$, and $\mathrm{send}_i(\pi, q, v) \cdot \mathrm{rec}_i(q, v)$ as $\mathrm{rec}_i(q, v)$. By Lemma 2, every cycle that is a witness for *non* k-synchronizability includes the node representing the pair $\mathrm{send}_i(p, q, v)$, $\mathrm{rec}_i(q, v)$. Moreover, the successor of this node in the cycle represents an action that is executed by p and the predecessor an action executed by q. Therefore, the monitor searches for a conflict-graph path from a node representing an action of p to a node representing an action of q. Whenever it finds such a path it goes to an error state.

Figure 10 lists the definition of $\mathcal{M}_{viol}(k)$ as an abstract state machine. By the construction of $\mathcal{S}'$, we assume w.l.o.g., that both the send to π and the send from π are executed in isolation as an instance of 1-exchange. When observing the send to π, the monitor updates the variable `conflict`, which in general stores the process executing the last action in the cycle, to p. Also, a variable `count`, which becomes 0 when the cycle has strictly more than k nodes, is initialized to k. Then, for every k-exchange transition in the execution, $\mathcal{M}_{viol}(k)$ non-deterministically picks pairs of matching send/receive or unmatched sends to continue the conflict-graph path, knowing that the last node represents an action of the process stored in `conflict`. The rules for choosing pairs of matching send/receive to advance the conflict-graph path are pictured on the right of Fig. 10 (advancing the conflict-graph path with an unmatched send doesn't modify the value of `conflict`, it just decrements the value of `count`). There are two cases depending on whether the last node in the path conflicts with the send or the receive of the considered pair. One of the two processes involved in this pair of send/receive equals the current value of `conflict`. Therefore, `conflict` can either remain unchanged or change to the value of the other process. The variable `lastIsRec` records whether the current conflict-graph path ends in a conflict due to a receive action. If it is the case, and the next conflict is between

```
function conflict: $\mathbb{P} \cup \{\bot\}$
function lastIsRec: $\mathbb{B}$
function sawRS: $\mathbb{B}$
function count: $\mathbb{N}$

rule $\text{send}_i(p, \pi, (q, v)) \cdot \text{rec}_i(\pi, (q, v))$:
  conflict := $p$
  count := k
//for every i, $\text{dest}(s_i) \neq \pi$ and $\text{proc}(s_i) \neq \pi$
rule $s_1 \cdot \ldots \cdot s_n \cdot r_1 \cdot \ldots \cdot r_m$:
  for i = 1 to $n$
    if ( $* \wedge \exists j.\ s_i \mapsto r_j \wedge$ conflict $\in \{\text{proc}(s_i), \text{dest}(s_i)\}$)
      if ( * )
        conflict := $\text{proc}(s_i)$
        if (lastIsRec) sawRS := true
        lastIsRec := false
      else
        conflict := $\text{dest}(s_i)$
        lastIsRec := true
      count--
    if ( * $\wedge$ $\text{proc}(s_i)$ = conflict $\wedge \forall j.\ \neg s_i \mapsto r_j$ )
      count--
      lastIsRec := false
rule $\text{send}_i(\pi, q, v) \cdot \text{rec}_i(q, v)$:
  assert conflict = $q \implies$ (count $> 0 \wedge \neg$sawRS)
```

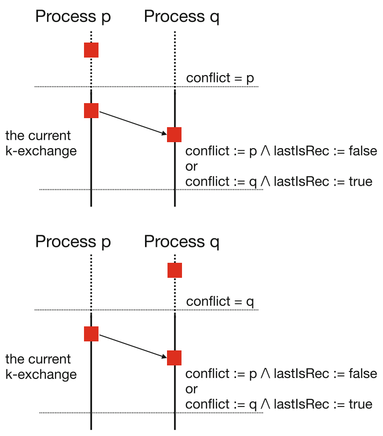

Process p
Process q
conflict = p
the current k-exchange
conflict := p ∧ lastIsRec := false
or
conflict := q ∧ lastIsRec := true
Process p
Process q
conflict = q
the current k-exchange
conflict := p ∧ lastIsRec := false
or
conflict := q ∧ lastIsRec := true

Fig. 10. The monitor $\mathcal{M}_{viol}(k)$. $\mathbb{B}$ is the set of Booleans and $\mathbb{N}$ is the set of natural numbers. Initially, conflict is $\bot$, while lastIsRec and sawRS are false.

this receive and a send, then sawRS is set to true to record the fact that the path contains an RS labeled edge (leading to a potential bad cycle).

When π sends its message to q, the monitor checks whether the conflict-graph path it discovered ends in a node representing an action of q. If this is the case, this path together with the node representing the delayed send forms a cycle. Then, if sawRS is true, then the cycle is bad and if count reached the value 0, then the cycle contains more than k nodes. In both cases, the current execution is a violation to k-synchronizability.

The set of executions in the k-synchronous semantics of $\mathcal{S}'$ composed with $\mathcal{M}_{causal}$ and $\mathcal{M}_{viol}(k)$, in which the latter goes to an error state, is denoted by $\mathcal{S}'_k \,||\, \mathcal{M}_{causal} \,||\, \neg\mathcal{M}_{viol}(k)$.

Theorem 2. *For a given k, a system $\mathcal{S}$ is k-synchronizable iff the set of executions $\mathcal{S}'_k \,||\, \mathcal{M}_{causal} \,||\, \neg\mathcal{M}_{viol}(k)$ is empty.*

Given a system $\mathcal{S}$, an integer k, and a local state l, *the reachability problem under the k-synchronous semantics* asks whether there exists a k-synchronous execution of $\mathcal{S}$ reaching a configuration $(\boldsymbol{l}, B)$ with $l = \boldsymbol{l}_p$ for some $p \in \mathbb{P}$. Theorem 2 shows that checking k-synchronizability can be reduced to a reachability problem under the k-synchronous semantics. This reduction holds for arbitrary (infinite-state) systems, which implies that k-synchronizability can be checked using the existing assertion checking technology. Moreover, for finite-state systems, where each process has a finite number of local states (message buffers can still be unbounded), it implies that checking this property is PSPACE-complete.

Theorem 3. *For a finite-state system $\mathcal{S}$, the reachability problem under the k-synchronous semantics and the problem of checking k-synchronizability of $\mathcal{S}$ are decidable and PSPACE-complete.*

7 Experimental Evaluation

Name	Proc	Loc	k	Time
Elevator	3	90	2	64.3s
OSR	4	63	1	1.28s
German	5	335	2	38m
Two-phase commit	4	57	1	1.43s
Replication storage	6	100	4	92.8s

Fig. 11. Experimental results.

As a proof of concept, we have applied our procedure for checking k-synchronizability to a set of examples extracted from the distribution of the P language[6]. Two-phase commit and Elevator are presented in Sect. 2, German is a model of the cache-coherence protocol with the same name, OSR is a model of a device driver, and Replication Storage is a model of a protocol ensuring eventual consistency of a replicated register. These examples cover common message communication patterns that occur in different domains: distributed systems (Two-phase commit, Replication storage), device drivers (Elevator, OSR), cache-coherence protocols (German). We have rewritten these examples in the Promela language and used the Spin model checker[7] for discharging the reachability queries. For a given program, its k-synchronous semantics and the monitors defined in Sect. 6 are implemented as ghost code. Finding a conflict-graph cycle which witnesses non k-synchronizability corresponds to violating an assertion.

The experimental data is listed in Fig. 11: Proc, resp., Loc, is the number of processes, resp., the number of lines of code (loc) of the original program, k is the *minimal* integer for which the program is k-synchronizable, and Time gives the number of minutes needed for this check. The ghost code required to check k-synchronizability takes 250 lines of code in average.

8 Related Work

Automatic verification of asynchronous message passing systems is undecidable in general [10]. A number of decidable subclasses has been proposed. The class of systems, called *synchronizable* as well, in [4], requires that a system generates the same sequence of send actions when executed under the asynchronous semantics as when executed under a synchronous semantics based on rendezvous communication. These systems are all 1-synchronizable, but the inclusion is strict (the 1-synchronous semantics allows unmatched sends). The techniques proposed in [4] to check that a system is synchronizable according to their definition cannot be extended to k-synchronizable systems. Other classes of systems that are 1-synchronizable have been proposed in the context of session types,

[6] Available at https://github.com/p-org.
[7] Available at http://spinroot.com.

e.g., [12,20,21,26]. A sound but incomplete proof method for distributed algorithms that is based on a similar idea of avoiding reasoning about all program computations is introduced in [3]. Our class of synchronizable systems differs also from classes of communicating systems that restrict the type of communication, e.g., lossy-communication [2], half-duplex communication [11], or the topology of the interaction, e.g., tree-based communication in concurrent pushdowns [19,23].

The question of deciding if all computations of a communicating system are equivalent (in the language theoretic sense) to computations with bounded buffers has been studied in, e.g., [17], where this problem is proved to be undecidable. The link between that problem and our synchronizability problem is not (yet) clear, mainly because non synchronizable computations may use bounded buffers.

Our work proposes a solution to the question of defining adequate (in terms of coverage and complexity) parametrized bounded analyses for message passing programs, providing the analogous of concepts such as context-bounding or delay-bounding defined for shared-memory concurrent programs. Bounded analyses for concurrent systems was initiated by the work on bounded-context switch analysis [25,28,29]. For shared-memory programs, this work has been extended to unbounded threads or larger classes of behaviors, e.g., [8,15,22,24]. Few bounded analyses incomparable to ours have been proposed for message passing systems, e.g., [6,23]. Contrary to our work, these works on bounded analyses in general do not propose decision procedures for checking if the analysis is complete (covers all reachable states). The only exception is [24], which concerns shared-memory.

Partial-order reduction techniques, e.g., [1,16], allow to define equivalence classes on behaviors, based on notions of action independence and explore (ideally) only one representative of each class. This has lead to efficient algorithmic techniques for enhanced model-checking of concurrent shared-memory programs that consider only a subset of relevant action interleavings. In the worst case, these techniques will still need to explore all of the interleavings. Moreover, these techniques are not guaranteed to terminate when the buffers are unbounded.

The work in [13] defines a particular class of schedulers, that roughly, prioritize receive actions over send actions, which is complete in the sense that it allows to construct the whole set of reachable states. Defining an analysis based on this class of schedulers has the same drawback as partial-order reductions, in the worst case, it needs to explore all interleavings, and termination is not guaranteed.

The approach in this work is related to robustness checking [5,7]. The general paradigm is to decide that a program has the same behaviors under two semantics, one being weaker than the other, by showing a polynomial reduction to a state reachability problem under the stronger semantics. For instance, in our case, the class of message passing programs with unbounded FIFO channels is Turing powerful, but still, surprisingly, k-synchronizability of these programs is decidable and PSPACE-complete. The results in [5,7] cannot be applied in our context: the class of programs and their semantics are different, and the corresponding robustness checking algorithms are based on distinct concepts and techniques.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.F.: Optimal dynamic partial order reduction. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20-21 January 2014, pp. 373–384. ACM (2014). https://doi.org/10.1145/2535838.2535845
2. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Inf. Comput. **127**(2), 91–101 (1996). https://doi.org/10.1006/inco.1996.0053
3. Bakst, A., von Gleissenthall, K., Kici, R.G., Jhala, R.: Verifying distributed programs via canonical sequentialization. PACMPL **1**(OOPSLA), 110:1–110:27 (2017). https://doi.org/10.1145/3133934
4. Basu, S., Bultan, T.: On deciding synchronizability for asynchronously communicating systems. Theor. Comput. Sci. **656**, 60–75 (2016). https://doi.org/10.1016/j.tcs.2016.09.023
5. Bouajjani, A., Derevenetc, E., Meyer, R.: Robustness against relaxed memory models. In: Hasselbring, W., Ehmke, N.C. (eds.) Software Engineering 2014, Kiel, Deutschland. LNI, vol. 227, pp. 85–86. GI (2014). http://eprints.uni-kiel.de/23752/
6. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 451–465. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_31
7. Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S.: Verifying robustness of event-driven asynchronous programs against concurrency. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 170–200. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_7
8. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_13
9. Bouajjani, A., Enea, C., Ji, K., Qadeer, S.: On the completeness of verifying message passing programs under bounded asynchrony. arXiv:1804.06612 [cs.PL]
10. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
11. Cécé, G., Finkel, A.: Verification of programs with half-duplex communication. Inf. Comput. **202**(2), 166–190 (2005). https://doi.org/10.1016/j.ic.2005.05.006
12. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_10
13. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, 20–24 October 2014, pp. 709–725. ACM (2014). https://doi.org/10.1145/2660193.2660211

14. Desai, A., Gupta, V., Jackson, E.K., Qadeer, S., Rajamani, S.K., Zufferey, D.: P: safe asynchronous event-driven programming. In: Boehm, H., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 321–332. ACM (2013). https://doi.org/10.1145/2462156.2462184
15. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 411–422. ACM (2011). https://doi.org/10.1145/1926385.1926432
16. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Palsberg, J., Abadi, M. (eds.) Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, 12–14 January 2005, pp. 110–121. ACM (2005). https://doi.org/10.1145/1040305.1040315
17. Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. Fundam. Inform. **80**(1–3), 147–167 (2007). http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09
18. Gupta, A., Malik, S. (eds.): CAV 2008. LNCS, vol. 5123. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1
19. Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. Logical Methods Comput. Sci. **8**(3) (2012). https://doi.org/10.2168/LMCS-8(3:23)2012
20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
21. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
22. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 245–261. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16164-3_18
23. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_21
24. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_54
25. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta and Malik [18], pp. 37–51. https://doi.org/10.1007/978-3-540-70545-1_7
26. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 221–232. ACM (2015). https://doi.org/10.1145/2676726.2676964
27. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)

28. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_7
29. Qadeer, S., Wu, D.: KISS: keep it simple and sequential. In: Pugh, W., Chambers, C. (eds.) Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, 9–11 June 2004, pp. 14–24. ACM (2004). https://doi.org/10.1145/996841.996845