On the Completeness of Verifying Message Passing Programs under Bounded Asynchrony*

Ahmed Bouajjani[1], Constantin Enea[1], Kailiang Ji[1], and Shaz Qadeer[3]

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

Abstract. We address the problem of verifying message passing programs, defined as a set of parallel 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 also show that reachability over k-synchronous computations and checking k-synchronizability are both PSPACE-complete. Furthermore, we introduce a class of programs called *flow-bounded* for which the problem of deciding whether there exists a $k > 0$ for which the program is k-synchronizable, is decidable.

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, web applications, 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. In general, sending messages is not blocking for the sender process, which means that the message buffers are supposed to be of unbounded size.

It is notorious that such programs are hard to get right. Indeed, asynchrony introduces a tremendous amount of new possible interleavings between actions of parallel processes, and makes very hard to apprehend the effect of all of their computations. Due to this complexity, expressing and verifying properties such as invariants for such systems is extremely hard. In particular, when buffer are ordered (FIFO buffers), the verification of invariants (or dually of reachability queries) is undecidable even when each of the processes is finite-state [10].

* 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).

Therefore, an important issue is the design of verification approaches that avoid considering the full sets 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., [4,13,6,25,17]. 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 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. Our intuition comes from the conviction we have that the behaviors of well designed systems 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 we consider on computations is based on a notion of *trace* corresponding to a *happens-before* relation that captures the program order (the order of actions in the code of a process) and the precedence order between send actions and their corresponding receive actions. Two computations are considered to be 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.

An important feature of our bounding concept is that it is possible to decide its completeness: 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* [3]. 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 with arbitrarily large configurations of the entry buffer of the consumer, the program is 1-synchronous because all its computations are equivalent to the computation where each message sent by the producer is immediately received by the consumer.

Importantly, we show that checking k-synchronizability of a program can be reduced in linear time to checking state reachability under the k-synchronous semantics (i.e., without considering all the program computations), which implies that checking k-synchronizability is PSPACE-complete. Thus, for k-synchronizable programs, it is possible to decide invariant properties without dealing with unbounded message buffers, and the overall complexity in this case is 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 might be the case 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. However, we are able to define a significant class of programs, including most examples in practice, for which this problem is decidable.

We have confronted our theory to a set of nontrivial examples. Some of these programs are given as motivating examples in the next section. All examples we have found are actually synchronizable (even if all of them are not flow-bounded), which confirms our conviction that non-synchronizability should correspond to an ill-designed system (and therefore it should be reported as an anomaly). Therefore, our approach always terminates and is complete for these systems.

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 nodes. The access to these nodes is controlled by a manager. Figure 2 shows an execution of this protocol. This system is

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

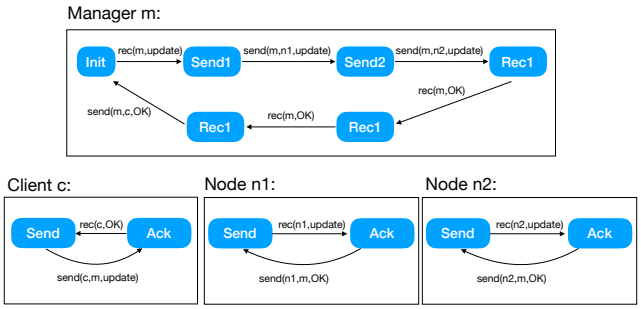


Fig. 1: A distributed commit protocol. Each process is defined as a labeled transition system. Transitions are labeled by send and receive actions, e.g., send(c, m, update) is a send from the client c to the manager m with payload update. Similarly, rec(c, OK) denotes process c receiving a message OK.

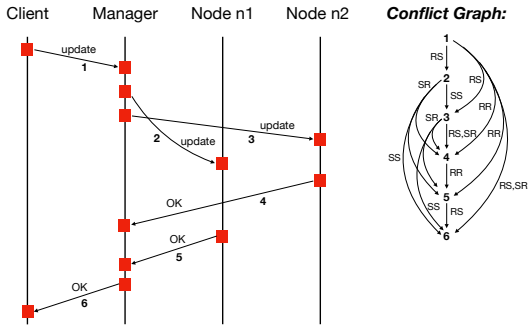


Fig. 2: An execution of the distributed commit protocol and its conflict graph.

1-synchronizable, i.e., every execution of this system 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 side to the other one 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 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 conflict graph being acyclic means that matching pairs of send-receive actions are "serializable", which implies that it is equivalent to an execution 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 for instance by a (family of) computations shown in Figure 4a of the elevator system shown in Figure 3 (a simplified version of the system described in [14]). In this execution, the user keeps sending requests for closing the door, which generates an unbounded

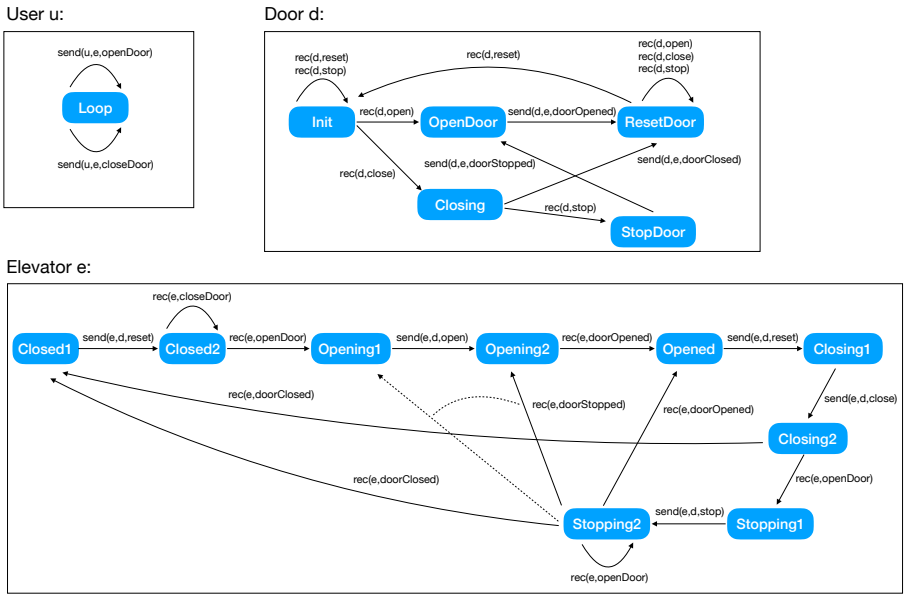
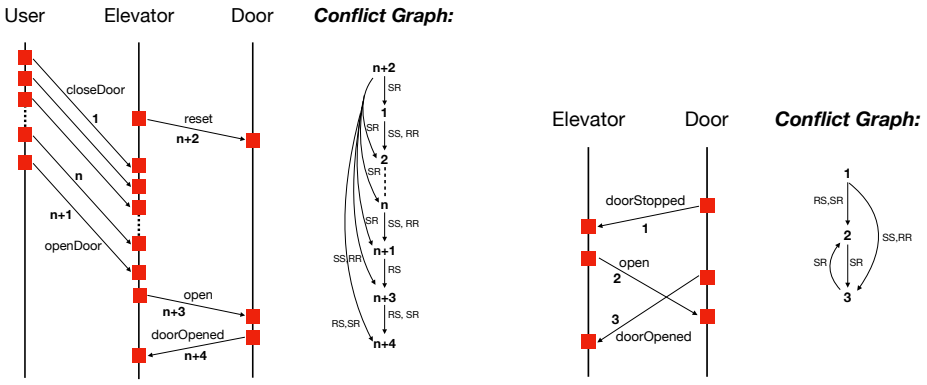


Fig. 3: The Elevator example

sequence of messages in the entry buffer of the elevator process. However, these computations are synchronizable since they are equivalent to a synchronous computation where the elevator receives immediately every message sent by the 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 shown in Figure 3 is a 1-synchronous system (without the dashed edge).

Consider now a slightly different version of the elevator system where the transition from Stopping2 to Opening2 is moved to target Opening1 instead of Opening2 (see the dashed transition in Figure 3). It can be seen that this version has the same state space as the previous one. Indeed, moving that transition from Stopping2 to Opening1 gives the possibility to Elevator to send a message open to Door, but the latter can only be between StopDoor and ResetDoor at this point, and therefore it can (maybe after sending doorStoped and doorOpened) receive at state ResetDoor the message open and stay in the same state. However, this version of the system is not 1-synchronizable as it is shown in Figure 4b: Suppose that Door is at state StopDoor, and that Elevator is at state Stopping2. Then, Door can send a message doorStoped and move to the state OpenDoor. Next, Elevator can receive that message and move to state Opening1. At this point, Elevator and Door can only exchange messages: message doorOpened from Door to Elevator and message open from Elevator to Door. The conflict graph of this execution, shown on the right of Figure 4b, contains a cycle of size 2 between the two matching pairs of send-receive actions involved in the exchange interaction.



(a) A 1-synchronizable execution.

(b) A computation with a 2-exchange.

Fig. 4: Executions of the Elevator.

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. 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.

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 Figure 1 and Figure 3.

We fix an arbitrary 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 $\mathsf{msg}(s) = \mathsf{msg}(r)$.

An execution of a system $\mathcal{S}$ under the asynchronous semantics is a sequence of indexed actions that is obtained as an interleaving of processes in $\mathcal{S}$ (see

Appendix B for a formal definition). Every send action enqueues the message into the destination's buffer, and every receive action dequeues a message from the corresponding buffer. 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 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, i.e., for every $p \in \mathbb{P}$, $po \cap \{a : a \in A \text{ and } \mathsf{proc}(a) = p\}^2$ is a total order, 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 the tuple (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 Figure 2 and Figure 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 $\text{send}_1(p_1, q_1, \_)$ $\text{rec}_1(q_1, \_)$ $\text{send}_2(p_2, q_2, \_)$ $\text{rec}_2(q_2, \_)$.

A direct consequence of the definitions is that the set of executions having the same trace are permutations of one another. Also, a system $\mathcal{S}$ cannot distinguish between permutations 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 Figure 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 Figure 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

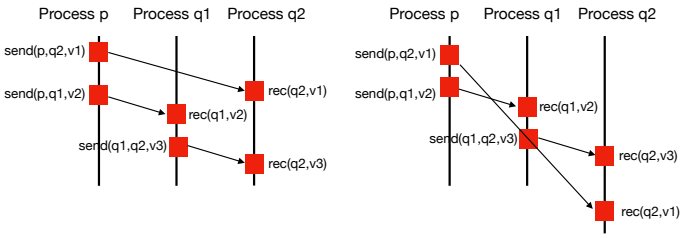


Fig. 5: A trace satisfying causal delivery (on the left) and a trace violating causal delivery (on the right).

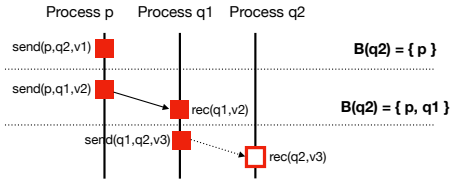


Fig. 6: An execution of the 1-synchronous semantics.

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.

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

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

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 rendezvous 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 smaller than k. For instance, the execution $\text{send}_1(p_1, q, \_)\ \text{send}_2(p_2, q, \_)\ \text{rec}_1(q, \_)\ \text{rec}_2(q, \_)$ is an instance of a 2-exchange. Actually, 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 ($\text{rec}_2(q, \_)$) is also an instance of a 2-exchange. The

k-EXCHANGE

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

Fig. 7: The synchronous semantics of a message passing system $\mathcal{S}$. Above, $\boldsymbol{\epsilon}$ denotes a vector where all the components are ϵ.

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 execution in Figure 6 which can be produced by a sequence of 1-exchanges. The receive action $(\mathrm{rec}(q_2, v_3))$ 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. Following the sequence of 1-exchanges in this execution, the unmatched $\mathrm{send}(p, q2, v1)$ 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 $\mathrm{send}(q1, q2, v_3)$ the receive $\mathrm{rec}(q_2, v_3)$ is disabled because $q1 \in B(q2)$.

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 Figure 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 $\rightarrow$. 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. In particular, this implies that such problems are decidable for finite-state k-synchronizable systems [4] Appendix C shows that the problem of detecting deadlocks in a k-synchronizable system can also be solved on the k-synchronous semantics instead of the asynchronous one.

5 Characterizing Synchronous Traces

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 [29]. 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 for which the conflict graph of its trace is acyclic, e.g., the execution in Figure 2, is "equivalent" to an execution where every receive immediately follows the matching send. In general, 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 Figure 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 characterize precisely the k-synchronous traces. Consider for instance the trace on the left of Figure 8. Its conflict-graph contains a cycle of size 4 (shown on the right), 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. We show that excluding such cycles, in addition to cycles of size strictly bigger than k, is a precise characterization of k-synchronous traces.

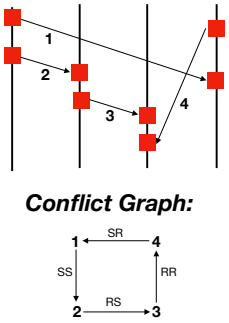


Fig. 8

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 $a \in \text{act}(v)$ and $a' \in \text{act}(v')$ such that $(a, a') \in po$ (where $\text{act}(v)$ 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 $a' \in Y_{id}$.

[4] A system is called *finite-state* when the number of local states of every process is bounded.

A direct consequence of previous results on conflict serializability [29] 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 is a characterization of k-synchronous traces (see Appendix D for a proof).

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 8 can be used to define a runtime monitor checking for k-synchronizability. The monitor maintains 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 in a system that executes under the *k-synchronous* semantics (where message buffers are bounded). We show that every *borderline* synchronizability violation (for which every strict prefix is synchronizable) of a system $\mathcal{S}$ can be "simulated" by the synchronous semantics of a system $\mathcal{S}'$ where the reception of exactly one message is delayed (w.r.t. the synchronous semantics of $\mathcal{S}$). Then, we give a monitor which observes executions of $\mathcal{S}'$ and identifies synchronizability violations (there exists a run of this monitor that goes to error whenever such a violation exists). Proofs of the results in this section can be found in Appendix 6.

6.1 Borderline Synchronizability Violations

For a system $\mathcal{S}$, a violation to k-synchronizability e 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 Figure 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 $\mathcal{S}$. 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.*

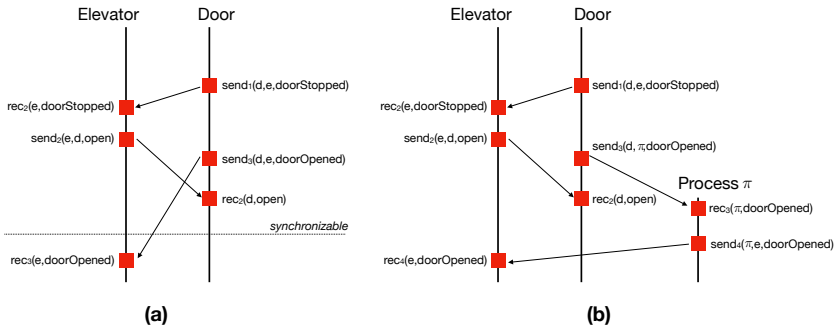


Fig. 9: A borderline violation to 1-synchronizability.

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 message, chosen nondeterministically: $\mathcal{S}'$ contains an additional process π and exactly one message sent by a process in $\mathcal{S}$ can be non-deterministically redirected to π which sends it to the original destination non-deterministically at a later time. 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 Figure 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 $e = e_1 \cdot \mathrm{send}_i(p,q,v) \cdot e_2 \cdot \mathrm{rec}_i(q,v)$ be a borderline violation to k-synchronizability of $\mathcal{S}$. Then, $\mathrm{sEx}_k(\mathcal{S}')$ contains an execution e' of the form:*

$$e' = e'_1 \cdot \mathrm{send}_i(p,\pi,(q,v)) \cdot \mathrm{rec}_i(\pi,(q,v)) \cdot e'_2 \cdot \mathrm{send}_j(\pi,q,v) \cdot \mathrm{rec}_j(q,v)$$

such that $e'_1 \cdot \mathrm{send}_i(p,q,v) \cdot e'_2$ is a permutation of $e_1 \cdot \mathrm{send}_i(p,q,v) \cdot e_2$.

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

$$\mathrm{send}_i(p,\pi,(q,v)) \cdot \mathrm{rec}_i(\pi,(q,v)) \cdot \mathrm{send}_{i'}(p,q,v') \cdot \mathrm{rec}_{i'}(q,v') \cdot \mathrm{send}_j(\pi,q,v) \cdot \mathrm{rec}_j(q,v)$$

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 $\mathrm{send}_i(p,q,v)) \cdot \mathrm{send}_{i'}(p,q,v') \cdot \mathrm{rec}_{i'}(q,v')\mathrm{rec}_i(q,v)$ which violates causal delivery and it is thus not possible under the asynchronous semantics of $\mathcal{S}$. Our

solution to this problem is to define a monitor $\mathcal{M}_{causal}$ which excludes such executions of $\mathcal{S}'$ when run under the synchronous semantics, i.e., it goes to an error state whenever applying the homomorphism σ leads to 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 defining a monitor $\mathcal{M}_{viol}(k)$ which observes executions in $\mathcal{S}'_k \,||\, \mathcal{M}_{causal}$ and checks whether they represent violations to k-synchronizability. We show that there exists a run of $\mathcal{M}_{viol}(k)$ that 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 constructs a conflict graph cycle, interpreting the sequence $\mathrm{send}_i(p, \pi, (q, v))\mathrm{rec}_i(\pi, (q, v))$ as in the original system $\mathcal{S}$, i.e., as $\mathrm{send}_i(p, q, v)$, and $\mathrm{send}_i(\pi, q, v)\mathrm{rec}_i(q, v)$ as $\mathrm{rec}_i(q, v)$. By Lemma 4, 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 founds such a path it goes to an error state. The set of executions in $\mathcal{S}'_k \,||\, \mathcal{M}_{causal}$ for which $\mathcal{M}_{viol}(k)$ 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*

$$\mathcal{S}'_k \,||\, \mathcal{M}_{causal} \,||\, \neg\mathcal{M}_{viol}(k) = \emptyset$$

7 Decidability results

We investigate several decidability and asymptotic complexity questions concerning the synchronous semantics and synchronizability. Proofs of the results in this section can be found in Appendix F.

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 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.*

We now give a syntactical criterion that imposes an upper bound on the number k for which a system could be k-synchronizable. In general, there are two reasons for which a system is not k-synchronizable, for every k. It either admits

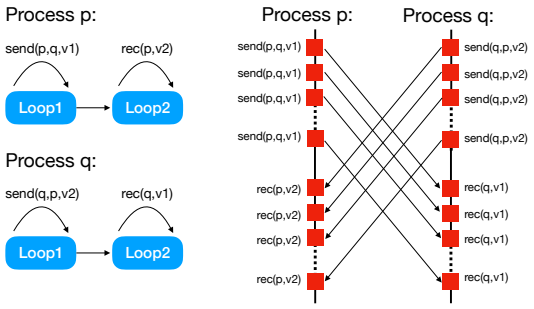


Fig. 10: An example of a system which is not k-synchronizable, for every k.

an execution with a bad conflict-graph cycle (e.g., the execution in Figure 8), or it admits executions with infinitely increasing conflict-graph cycles. If a system admits a bad conflict-graph cycle, then there exists a k for which it can be shown to be non k-synchronizable (a coarse upper bound for k is the length of the execution containing this cycle). The second case is exemplified by the system in Figure 10: the two loops in each process allow to create executions with unboundedly many send actions before any receive is enabled. However, the systems we have encountered in practice do not contain such scenarios.

In fact, the large majority of the processes composing practical systems, e.g., systems developed in the P language [5], perform a bounded number of consecutive receives, and a bounded number of sends before a receive. If all processes in the system would satisfy this constraint, then there exists a bound k_s on the number of sends that are enabled before a receive, and a bound k_r on the number of receives that are enabled before a send, which would imply that the system is k-synchronizable for some k iff it is k-synchronizable for some $k \leq (k_s + k_r) \times |\mathbb{P}|$. However, there exist processes which don't satisfy this constraint, e.g., a consumer in a standard producer-consumer scenario and the process `Elevator` in Figure 3, which performs an unbounded number of consecutive receive actions. While in the first case, the system would be 1-synchronous, in the second case, the unbounded number of receives is just an "optimization" that doesn't change the set of reachable local state vectors. The self loop where an unbounded number of messages `closeDoor` can be received from the `User` means that all these messages can be ignored since the door of the elevator is anyway closed. This unbounded interaction between `Elevator` and `User` will leave both processes in exactly the same state. Removing this self loop and considering executions where the `User` sends exactly one message `closeDoor` instead of an unbounded sequence (before a message `openDoor`) will allow to discover all the reachable local state vectors. Ignoring the self-loops in `Door` can be motivated in the same way.

Let $\mathcal{S} = ((L_p, \delta_p, l_p^0) \mid p \in \mathbb{P})$ be a message passing system. A process p is called *k-receive bounded* when it can perform at most k consecutive receives, i.e., for every sequence $w \in (S \cup R)^*$ accepted by the labeled transition system (L_p, δ_p, l_p^0), there exists no decomposition $w = w_1 \cdot w_2 \cdot w_3$ where $w_2 \in R^*$, and the

[5] Available at `https://github.com/p-org`.

length of w_2 is strictly bigger than k. A process p is called *k-send bounded* when it can perform at most k consecutive sends before a receive, i.e., for every sequence $w \in (S \cup R)^*$ accepted by the labeled transition system (L_p, δ_p, l_p^0), there exists no decomposition $w = w_1 \cdot w_2 \cdot r \cdot w_3$, where $r \in R$, $w_2 \in S^*$, and the length of w_2 is strictly bigger than k. For instance, all the processes in the distributed commit protocol in Figure 1 are 2-receive bounded and 2-send bounded. The system $\mathcal{S}$ is called *flow-bounded* when there exists a constant k such that every process $p \in \mathbb{P}$ is k-receive bounded and k-send bounded. Note that verifying flow-boundedness is reducible to a reachability problem for a single process and thus, decidable for finite-state processes.

Theorem 4. *For a flow-bounded system $\mathcal{S}$, the problem of checking if there exists some k such that $\mathcal{S}$ is k-synchronizable, is decidable.*

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 which cannot be produced by rendezvous communication). 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, e.g., [12,22,21,28]. Our class of synchronizable systems differs also from other 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 particular classes of concurrent push-down systems [25,20].

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., [18], 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 in fact initiated by the work on bounded-context switch analysis [31,30,27]. For shared-memory programs, this work has been extended to an unbounded number of threads or larger classes of behaviors, e.g., [9,15,23,26]. Few bounded analyses incomparable to ours have been proposed for message passing systems, e.g., [25,6]. Contrary to our work, all these works on bounded

analyses in general do not propose decision procedures for checking if the analysis is complete, i.e., it covers the whole set of reachable states. The only exception that we are aware of is [26], which however concerns shared-memory programs.

Partial-order reduction techniques, e.g., [1,17], 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 notion of conflict-graph is similar to the one used for defining conflict serializability [29]. However, our algorithms and proof techniques are very different from those used in this context, e.g., [3,7,16]. Our approach considers several classes of cycles in these graphs and focuses on showing that these cycles can be detected without exploring all the behaviors of a system.

The approach we adopt in this work is related to robustness checking [5,8]. 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, i.e., by avoiding the consideration of the weak semantics that is in general far more complex to deal with than the strong one. For instance, in the case of our work, 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 (i.e., as hard as state reachability in programs with bounded channels). However, the results in [5,8] can not be applied to solve the question of synchronizability we consider in this paper; in each of [5], [8], and our work, the considered classes of programs and their strong/weak semantics are very different (shared-memory concurrent programs running over a relaxed memory model in [5], and shared-memory concurrent programs with dynamic asynchronous process creation in [8]), and the corresponding robustness checking algorithms are based on distinct concepts and techniques.

9 Experimental Evaluation

Name	Proc	Loc	Instr	k	Time
Elevator	3	94	481	2	13m
Two-phase commit	4	145	381	1	10m
Replication Storage	5	243	515	4	15m
German Protocol	5	300	637	2	25m
OSR	4	154	464	1	22m

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]. In the absence of an ex-

[6] Available at `https://github.com/p-org`.

haustive model-checker for this language, we have rewritten these examples in the Promela language and used the Spin model checker [7] for discharging the reachability queries. For a given Promela program, its k-synchronous semantics is implemented as an instrumentation which uses additional boolean variables to enforce that sends and receives interleave in k-exchange phases. Then, the monitors defined in Section 6 are defined as additional processes which observe the sequence of k-exchanges in an execution and update their state accordingly. Finding a conflict-graph cycle which witnesses non k-synchronizability corresponds to violating an assertion.

The experimental data is listed in Figure 11: Proc, resp., Loc, is the number of processes, resp., the number of lines of code (loc) of the original program, Instr is the number of loc added by the instrumentation, k is the *minimal* integer for which the program is k-synchronizable, and Time gives the number of minutes needed for this check. The first three examples are the ones presented in Section 2 and Appendix A. The German protocol is a modelization of the cache-coherence protocol with the same name, and OSR is a modelization of a device driver.

[7] Available at `http://spinroot.com`

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 '14, San Diego, CA, USA, January 20-21, 2014. pp. 373–384. ACM (2014), http://doi.acm.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. Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. 160(1-2), 167–188 (2000), https://doi.org/10.1006/inco.1999.2847
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.) Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7214, pp. 451–465. Springer (2012), https://doi.org/10.1007/978-3-642-28756-5_31
7. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 290–309. Springer (2013), https://doi.org/10.1007/978-3-642-37036-6_17
8. Bouajjani, A., Emmi, M., Enea, C., Ozkan, B.K., Tasiran, S.: Verifying robustness of event-driven asynchronous programs against concurrency. In: Yang, H. (ed.) Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10201, pp. 170–200. Springer (2017), https://doi.org/10.1007/978-3-662-54434-1_7
9. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6887, pp. 129–145. Springer (2011), https://doi.org/10.1007/978-3-642-23702-7_13
10. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983), http://doi.acm.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., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 194–213. Springer (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, October 20-24, 2014. pp. 709–725. ACM (2014), http://doi.acm.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 '13, Seattle, WA, USA, June 16-19, 2013. pp. 321–332. ACM (2013), http://doi.acm.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, January 26-28, 2011. pp. 411–422. ACM (2011), http://doi.acm.org/10.1145/1926385.1926432
16. Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta and Malik [19], pp. 52–65, https://doi.org/10.1007/978-3-540-70545-1_8
17. 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, January 12-14, 2005. pp. 110–121. ACM (2005), http://doi.acm.org/10.1145/1040305.1040315
18. 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
19. Gupta, A., Malik, S. (eds.): Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, Lecture Notes in Computer Science, vol. 5123. Springer (2008), https://doi.org/10.1007/978-3-540-70545-1
20. Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. Logical Methods in Computer Science 8(3) (2012), https://doi.org/10.2168/LMCS-8(3:23)2012
21. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1381, pp. 122–138. Springer (1998), https://doi.org/10.1007/BFb0053567
22. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM 63(1), 9:1–9:67 (2016), http://doi.acm.org/10.1145/2827695

23. Kidd, N., Jagannathan, S., Vitek, J.: One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling. In: van de Pol, J., Weber, M. (eds.) Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6349, pp. 245–261. Springer (2010), https://doi.org/10.1007/978-3-642-16164-3_18
24. Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 254–266. IEEE Computer Society (1977), https://doi.org/10.1109/SFCS.1977.16
25. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 299–314. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_21
26. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6174, pp. 629–644. Springer (2010), https://doi.org/10.1007/978-3-642-14295-6_54
27. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta and Malik [19], pp. 37–51, https://doi.org/10.1007/978-3-540-70545-1_7
28. 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, January 15-17, 2015. pp. 221–232. ACM (2015), http://doi.acm.org/10.1145/2676726.2676964
29. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
30. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 93–107. Springer (2005), https://doi.org/10.1007/978-3-540-31980-1_7
31. 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, June 9-11, 2004. pp. 14–24. ACM (2004), http://doi.acm.org/10.1145/996841.996845

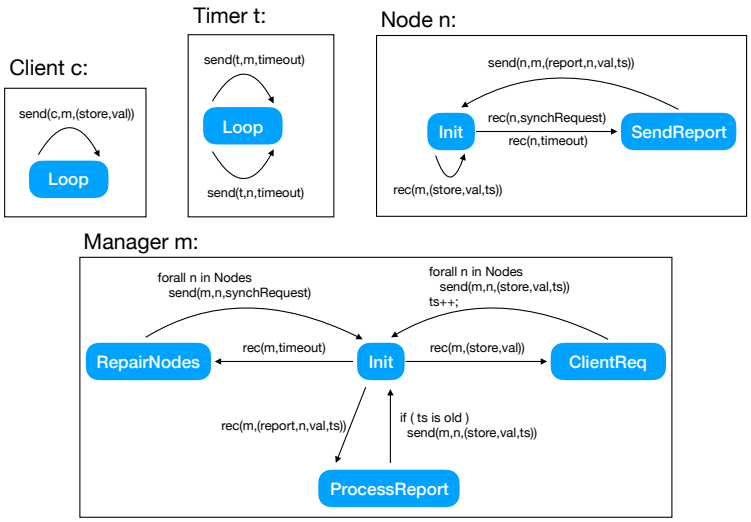


Fig. 12: A replication storage protocol.

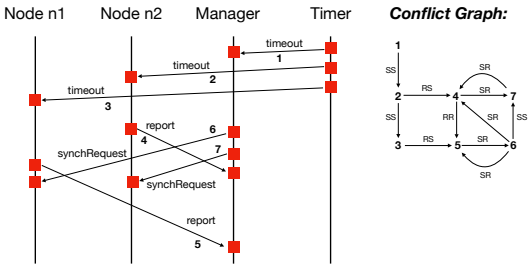


Fig. 13: An execution of the replication storage protocol and its conflict graph.

A Motivating Examples

We present an example illustrating the fact that exchanges can involve not only 2 processes but several ones, maybe all the processes of a system. Figure 12 shows a replication storage protocol. A client can send update requests to the manager who is in charge of maintaining several storage replicas on different nodes. When the manager receives an update request from the client, it forwards this message to all the nodes. However, since messages can be delayed, the information in the nodes can be different at various points in time. Then, a mechanism is used to force regularly synchronization between the data versions stored in the different nodes. This mechanism is based on (1) using time-stamps for each message that is sent by the manager to the nodes, and (2) a timer that triggers synchronizations: the timer can send timeout messages to either the manager or to the nodes. When a node receives such a timeout message, it sends a report to the manager with its

$$\text{SEND}\quad \frac{m = (i, v) \qquad i \in \mathbb{M} \text{ fresh} \qquad (\boldsymbol{l}_p, \text{send}(p, q, v), l) \in \delta_p}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\text{send}_i(p,q,v)} \boldsymbol{l}[\boldsymbol{l}_p \leftarrow l], \boldsymbol{b}[\boldsymbol{b}_q \leftarrow \boldsymbol{b}_q \cdot m]}$$

$$\text{RECEIVE}\quad \frac{\boldsymbol{b}_q = (i, v) \cdot b \qquad (\boldsymbol{l}_q, \text{rec}(q, v), l) \in \delta_q}{\boldsymbol{l}, \boldsymbol{b} \xrightarrow{\text{rec}_i(q,v)} \boldsymbol{l}[\boldsymbol{l}_q \leftarrow l], \boldsymbol{b}[\boldsymbol{b}_q \leftarrow b]}$$

Fig. 14: The asynchronous semantics of a message passing system $\mathcal{S}$. For a vector $\boldsymbol{x}$, $\boldsymbol{x}[\boldsymbol{x}_p \leftarrow E]$ is the vector $\boldsymbol{y}$ with $\boldsymbol{y}_q = \boldsymbol{x}_q$, for every $q \neq p$, and $\boldsymbol{x}_p = E$. Also, $\cdot$ denotes the concatenation of two sequences.

current data value, and when the manager receives the timeout message, it sends to all nodes a message requesting a synchronization. When a node receives the synchronization request, it sends to the manager a report with its last value. After each reception of a report from a node, the manager checks if the received value is up-to-date using its time-stamp, and if not, it sends the most recent value to the node. Now, since the nodes and the manager may all receive timeout messages from the timer, nodes can start sending reports to the manager while the latter is already sending them synchronization requests. This leads to an exchange that may involve all the nodes, in addition to the manager. This situation is shown in Figure 13. The conflict graph shown on the right of the figure contains a cycle of size 4, which is in this case the number of involved processes (two nodes and one manager), plus 1, which means that the considered computation is not 3-synchronizable. It can be checked that the system is actually 4-synchronizable.

B Asynchronous Semantics of Message Passing Systems

Formally, configuration $c = \langle \boldsymbol{l}, \boldsymbol{b} \rangle$ is a vector $\boldsymbol{l}$ of local states together with a vector $\boldsymbol{b}$ of message buffers that are represented as sequences of message payloads tagged with unique identifiers. The identifiers are used only for technical reasons, to identify a "matching" relation between sends and receives. These two vectors are indexed by elements of $\mathbb{P}$. For a vector $\boldsymbol{x}$, let $\boldsymbol{x}_p$ denote the element of $\boldsymbol{x}$ of index p. The initial configuration c_0 of the system $\mathcal{S}$ is the tuple of initial local states together with empty message buffers, i.e., $c_0 = \langle \boldsymbol{l}, \boldsymbol{b} \rangle$ where $\boldsymbol{l}_p = l_p^0$ and $\boldsymbol{b} = \epsilon$ for each $p \in \mathbb{P}$.

The transition relation $\rightarrow$ in Figure 14 is determined by a message passing system $\mathcal{S}$, and maps a configuration c_1 to another configuration c_2 and indexed action $a \in S_{id} \cup R_{id}$. The effect of a SEND transition is to enqueue the message payload tagged with a fresh identifier to the buffer of the destination, and the effect of a RECEIVE transition is to dequeue a message from the local buffer.

An *execution* of a system $\mathcal{S}$ under the asynchronous semantics to configuration c_n is a sequence of indexed actions $a_1 \dots a_n$ such that there exists a configuration

sequence $c_0 c_1 \dots c_n$ with $c_m \xrightarrow{a_{m+1}} c_{m+1}$ for all $0 \leq m < n$. We say that c_n is reachable in $\mathcal{S}$ under the asynchronous semantics. The *reachable local state vectors* of $\mathcal{S}$, denoted by $\mathrm{asSt}(\mathcal{S})$, is the set of local state vectors in reachable configurations. The set of executions of $\mathcal{S}$ under the asynchronous semantics is denoted by $\mathrm{asEx}(\mathcal{S})$. In the following, we don't distinguish between executions obtained by a consistent renaming of the message identifiers.

C Detecting deadlocks

In addition to assertion/invariant checking, we show that the problem of detecting deadlocks in a k-synchronizable system can also be solved using the k-synchronous semantics instead of the asynchronous one. For a process p, a state $l \in L_p$ is called *receiving* when $(l, a, l') \in \delta_p$, for some l', implies that $a \in R_p$. For instance, the state `Init` of the process `Manager` in Figure 1 is receiving. The state l is called *final* when there exists no l' and a such that $(l, a, l') \in \delta_p$.

We consider several notions of deadlock: a configuration $c = (\boldsymbol{l}, \boldsymbol{b})$ is called

- *empty-buffer deadlock* when all the buffers are empty, there exists at least one process waiting for a message, and all the other processes are either in a final or receiving state, i.e., $\boldsymbol{b} = \boldsymbol{\epsilon}$, there exists $p \in \mathbb{P}$ such that $(\boldsymbol{l}_p, r, l') \in \delta_p$ for some $r \in R$, and for all $q \in \mathbb{P}$, $\boldsymbol{l}_q$ is receiving or final,
- *orphan message configuration* when there is at least a non-empty buffer and each process is in a final state, i.e., $\boldsymbol{b} \neq \boldsymbol{\epsilon}$ and for all $p \in \mathbb{P}$, $\boldsymbol{l}_p$ is final,
- *unspecified reception* when some process is prevented from receiving any message from its buffer, i.e., there exists $p \in \mathbb{P}$ such that $\boldsymbol{l}_p$ is receiving, and for all $\mathrm{rec}(p, v) \in R$, if $(\boldsymbol{l}_p, \mathrm{rec}(p, v), l') \in \delta_p$, for some l', then $\boldsymbol{b}_p \notin v\mathbb{V}^*$.

We show that reachability of such configurations in the original asynchronous semantics can be reduced to reachability problems over the synchronous semantics, provided that the system is k-synchronizable. The constraints over the buffers of the asynchronous configurations are replaced by constraints over the executions (traces) of the synchronous semantics. For instance, an execution reaching an empty-buffer configuration is "equivalent" to a synchronous *matched* execution where every sent message has been received (assuming k-synchronizability).

We extend the notion of empty-buffer deadlock to configurations of the synchronous semantics by removing the condition that the buffers are empty.

Theorem 5. *A k-synchronizable system $\mathcal{S}$ reaches an empty-buffer deadlock configuration under the asynchronous semantics iff the k-synchronous semantics of $\mathcal{S}$ admits a matched execution to an empty-buffer deadlock configuration.*

Proof. We prove the only-if direction, the reverse being similar. Let e be an execution in $\mathrm{asEx}(\mathcal{S})$ to an empty-buffer deadlock configuration $(\boldsymbol{l}, \boldsymbol{\epsilon})$. Since the buffers are empty, by the definition of the asynchronous semantics, we get that e is matched. By k-synchronizability, there exists a permutation e' of e that belongs to $\mathrm{sEx}_k(\mathcal{S})$. Then, by Lemma **??**, e' is an execution to a configuration $(\boldsymbol{l}, B)$, for some B, which finishes the proof.

A configuration $(\boldsymbol{l}, B)$ of the synchronous semantics is called *final* when every local state $\boldsymbol{l}_p$ with $p \in \mathbb{P}$ is final. The proof of the following result is similar to that of Theorem 5, the only addition being that an asynchronous execution to a configuration with non-empty buffers corresponds to a synchronous execution with unmatched send actions (provided that the system is k-synchronizable).

Theorem 6. *A k-synchronizable system $\mathcal{S}$ reaches an orphan message configuration under the asynchronous semantics iff the k-synchronous semantics of $\mathcal{S}$ admits an execution containing at least one unmatched send to a final configuration.*

A local state l of a process p is called *V-receiving* when it is receiving and the set of messages that can be received in l is exactly V, i.e., for all v, $v \in V$ iff there exists $l' \in L_p$ such that $(l, \mathrm{rec}(p, v), l') \in \delta_p$. A configuration $(\boldsymbol{l}, B)$ of the synchronous semantics is called *(p, V)-receiving* when $\boldsymbol{l}_p$ is V-receiving. Given an execution e, let $\mathit{minUnmatched}(e, p)$ be the set of unmatched send actions in e which are minimal in the causal relation of $tr(e)$ among unmatched send actions with destination p, i.e., $\mathit{minUnmatched}(e, p)$ is the set of unmatched send actions $\mathrm{send}_i(p', p, v)$ in e such that for every other unmatched send action $\mathrm{send}_j(p'', p, v)$ in e we have that $\mathrm{send}_i(p'', p, v) \not\rightsquigarrow_{tr(e)} \mathrm{send}_i(p', p, v)$.

Theorem 7. *A k-synchronizable system $\mathcal{S}$ reaches an unspecified reception configuration under the asynchronous semantics iff there exists some $p \in \mathbb{P}$ such that the k-synchronous semantics of $\mathcal{S}$ admits an execution e to a (p, V)-receiving state and*

$$\{v : \exists \mathrm{send}_i(p', p, v) \in \mathit{minUnmatched}(e, p)\} \setminus V \neq \emptyset.$$

Proof. For the only-if direction, let e be an execution in $\mathrm{asEx}(\mathcal{S})$ to an unspecified reception configuration $(\boldsymbol{l}, \boldsymbol{b})$. Then, there exists $p \in \mathbb{P}$ such that $\boldsymbol{l}_p$ is V-receiving, for some $V \in \mathbb{V}$, and $v_p \notin V$, where v_p is the head of $\boldsymbol{b}_p$ (the first message to be dequeued). Therefore, e contains an unmatched send action $\mathrm{send}_i(p', p, v_p)$ which is also the first among unmatched send actions with destination p (otherwise, v_p would not be the first message in the buffer of p). Therefore, $\mathrm{send}_i(p', p, v_p) \in \mathit{minUnmatched}(e, p)$. By k-synchronizability, there exists a permutation e' of e that belongs to $\mathrm{sEx}_k(\mathcal{S})$. By Lemma **??**, e' is an execution to a configuration $(\boldsymbol{l}, B)$, for some B, which is (p, V)-receiving. Since e and e' have the same trace, we get that $\mathrm{send}_i(p', p, v_p) \in \mathit{minUnmatched}(e', p)$ as well, which finishes the proof of this direction.

For the if direction, assume that the k-synchronous semantics of $\mathcal{S}$ admits an execution e as above. Let $\mathrm{send}_i(p', p, v)$ be an action in $\mathit{minUnmatched}(e, p)$ such that $v \notin V$. Because $\mathrm{send}_i(p', p, v)$ is minimal among unmatched send actions with destination p, there exists a permutation e' of e where it is the first unmatched send with destination p. As a direct consequence of the definitions, we get that e is an execution to an unspecified reception configuration.

D Proofs of Section 5

Theorem 8. *A trace t satisfying causal delivery is k-synchronous iff every cycle in its conflict-graph is good and of size at most k.*

Proof. For the only if direction, t is the trace of an execution $e \in \mathrm{sEx}_k(\mathcal{S})$ for some system $\mathcal{S}$. The execution e is obtained through a sequence of k-EXCHANGE transitions. The set of actions of every node v of CG_t appear in the label of a single such transition. Moreover, for every cycle in CG_t, the actions corresponding to the nodes in this cycle occur in the label of a single k-EXCHANGE transition. Therefore, every cycle in CG_t is good and of size at most k.

For the if direction, we first show that any strongly-connected component C of CG_t is k-synchronous. Since all the cycles in CG_t are of size at most k, we get that C contains at most k nodes. The case of strongly-connected components formed of a single node v is trivial. The actions corresponding to v are either a matching pair of send/receive actions, which can be generated through a 1-EXCHANGE transition, or an unmatched send, which can also be generated through a 1-EXCHANGE transition. Now, let C be formed of a set of nodes $v_1,\ldots, v_n$, for some $2 \leq n \leq k$ such that $s_i \in act(v_i)$, for all $1 \leq i \leq n$. W.l.o.g., assume that the indexing of the nodes in C is consistent with the edges labeled by SS (note that there is no cycle formed only of edges labeled by SS), i.e., for every $1 \leq i_1 < i_2 \leq n$, C doesn't contain an edge labeled by SS from i_2 to i_1, and for every $1 \leq i_1 < i_2 < i_3 \leq n$, if $\mathsf{proc}(s_{i_1}) = \mathsf{proc}(s_{i_3})$, then $\mathsf{proc}(s_{i_1}) = \mathsf{proc}(s_{i_2})$. Let $i_1,\ldots,i_m$ be the maximal subsequence of $1,\ldots,n$ such that $r_{i_j} \in act(v_i)$ for every $1 \leq j \leq m$. We have that C is the trace of the execution $e = s_1 \cdot \ldots \cdot s_n \cdot r_{i_1} \cdot \ldots \cdot r_{i_m}$. The fact that all sends can be executed before the receives is a consequence of the fact that C doesn't contain edges labeled by RS. Then, the order between receives is consistent with the one between sends because C satisfies causal delivery. By definition, e is the label of an n-EXCHANGE transition, and therefore, C is k-synchronous.

To complete the proof we proceed by induction on the number of strongly connected components of CG_t. The base case is trivial. For the induction step, assume that the claim holds for every trace whose conflict-graph has at most n strongly connected components, and let t be a trace with $n+1$ strongly connected components. Let C be a strongly connected component of t such that C has no outgoing edges towards another strongly connected component of t. By the definition of the conflict-graph, t is the trace of an execution $e = e' \cdot e''$ where e'' contains all the actions corresponding to the nodes of C. We have shown above that e'' is k-synchronous, and by the induction hypothesis, e' is also k-synchronous. Therefore, e is k-synchronous.

E Proofs of Section 6

E.1 Borderline Synchronizability Violations

Lemma 3. *Let e be a borderline violation to k-synchronizability of $\mathcal{S}$. Then, $e = e' \cdot r$ for some $e' \in (S_{id} \cup R_{id})^*$ and $r \in R_{id}$.*

Proof. Assume by contradiction that $e = e' \cdot s$ for some $e' \in (S_{id} \cup R_{id})^*$ and $s \in S_{id}$. By definition, $CG_{tr(e)}$ contains no outgoing edge from the node representing s, which implies that any cycle of $CG_{tr(e)}$ is already contained in $CG_{tr(e')}$. This is a contradiction to the fact that e is borderline.

Lemma 4. *Let $e = e' \cdot r$, for some $e' \in (S_{id} \cup R_{id})^*$ and $r \in R_{id}$, be a borderline violation to k-synchronizability of $\mathcal{S}$. Then, 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.*

Proof. Let $v_0, v_1, \ldots, v_n, v_0$ be a cycle of $CG_{tr(e)}$ which is bad or of size bigger than k. We first show that $v = v_i$ for some $0 \leq i \leq n$. Assume by contradiction that this is not the case. Then, the execution e' is already a violation to k-synchronizability which violates the assumption that e is borderline.

For the following, w.l.o.g., we assume that $v = v_0$. Since r is the last action of e, the only outgoing edge of v is an edge labeled by SX with $X \in \{S, R\}$. Therefore (v, v_1) is an SX edge. Assuming by contradiction that the edge (v_n, v) is labeled by YS for some $Y \in \{S, R\}$ implies that e' is already a k-synchronizability violation, which contradicts the hypothesis.

E.2 Simulating Borderline Violations on the Synchronous Semantics

We define a system $\mathcal{S}'$ which "delays" the reception of exactly one message, which is chosen nondeterministically, by redirecting it to an additional process π which relays it to the original destination at a later time. We show that the synchronous semantics of $\mathcal{S}'$ "simulates" a permutation of every borderline violation of $\mathcal{S}$.

Formally, given $\mathcal{S} = ((L_p, \delta_p, l_p^0) \mid p \in \mathbb{P})$, we define $\mathcal{S}' = ((L_p, \delta'_p, l_p^0) | p \in \mathbb{P} \cup \{\pi\})$ where

- every send of a process p can be non-deterministically redirected to the process π (the message payload contains the destination process), i.e.,

  $\delta'_p(l, \text{send}(p, \pi, (q, v))) = \delta'_p(l, \text{send}(p, q, v))$, and
  $\delta'_p(l, a) = \delta_p(l, a)$ for all $p \in \mathbb{P}$, $l \in L_p$, and $a \notin \{\text{send}(p, \pi, v) | p \in \mathbb{P}, v \in \mathbb{V}\}$

- the process π stores the received message in its state and relays it later, i.e., $L_\pi = \{l_\pi^0, l_f\} \cup \{(q, v) \mid q \in \mathbb{P}, v \in \mathbb{V}\}$, and for all $q \in \mathbb{P}$ and $v \in \mathbb{V}$,

$$\delta'_p(l_\pi^0, \text{rec}(\pi, (q, v))) = \{(q, v)\} \text{ and } \delta'_p((q, v), \text{send}(\pi, q, v)) = l_f$$

Lemma 5. *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 $e_1 \cdot \text{send}_i(p, q, v) \cdot e_2$.

```
function cone: 2^ℙ
function receiver: ℙ ∪ {⊥}

rule s_1 · ... · s_n · r_1 · ... · r_m :
  if ( ∃i. proc(s_i) = π )
    pass
  if ( ∃i. s_i = send_(p, π, (q, v)) )
    cone := {p}
    receiver := q
  forall j with 1 ≤ j ≤ n
    if ( p ∈ cone ∧ ∃k. s_j ↦ r_k ∧ (∃i. dest(s_i) = π ⟹ (proc(s_i) = proc(s_j) ∧ i < j)) )
      cone := cone ∪ {dest(s_j)}
      assert dest(s_j) ≠ receiver
```

Fig. 15: The monitor $\mathcal{M}_{causal}$. Initially, `receiver` is $\bot$, and `cone` $= \emptyset$.

Proof. A direct consequence of the definition of $\mathcal{S}'$ is that $e' \in \text{asEx}(\mathcal{S}')$. We show that the trace of e' is k-synchronous. The conflict graph of $tr(e')$ can be obtained from the one of $tr(e)$ as follows:

- the node v representing the pair of actions $\{\text{send}_i(p,q,v), \text{rec}_i(q,v)\}$ is replaced by two nodes v' and v'' representing $\{\text{send}_i(p,\pi,(q,v)), \text{rec}_i(\pi,(q,v))\}$ and $\{\text{send}_j(\pi,q,v), \text{rec}_j(q,v)\}$, respectively,
- for every SX edge from v to a node w in $CG_{tr(e)}$, where $X \in \{S,R\}$, there exists an SX edge from v' to w in $CG_{tr(e')}$,
- v' is connected to v'' by an RS edge,
- there is no outgoing edge from v''.

Since all the cycles of $CG_{tr(e)}$ that are bad or exceed the size k pass trough v, we get that $CG_{tr(e')}$ contains no such cycle. Therefore, $tr(e')$ is k-synchronous.

This implies that $\text{sEx}_k(\mathcal{S}')$ contains a permutation of $e_1 \cdot \text{send}_i(p,\pi,(q,v)) \cdot \text{rec}_i(\pi,(q,v)) \cdot e_2 \cdot \text{send}_j(\pi,q,v) \cdot \text{rec}_j(q,v)$. Since there is no outgoing edge from v'', there exists such a permutation that ends in $\text{send}_j(\pi,q,v) \cdot \text{rec}_j(q,v)$ which concludes the proof.

E.3 Excluding Executions Violating Causal Delivery

The monitor $\mathcal{M}_{causal}$ is essentially a labeled transition system whose transitions are labeled by sequences of actions in $S_{id}^* \cdot R_{id}^*$ corresponding to k-exchange transitions of the synchronous semantics. For readability, we define it as an abstract state machine in Figure 15. $\mathcal{M}_{causal}$ tracks the set of processes `cone` who perform a send that is causally after the send to π, and checks that the message they sent is not received by the same process to whom π must send a message. It performs this check before π sends a message and therefore, any failure will correspond to an execution which is not possible in $\mathcal{S}$ (violating causal delivery).

The set of executions of the k-synchronous semantics of $\mathcal{S}'$ for which $\mathcal{M}_{causal}$ doesn't go to an error state (the assertion in $\mathcal{M}_{causal}$ passes at every step) is denoted by $\mathcal{S}'_k \,||\, \mathcal{M}_{causal}$. The following result shows that every such execution is correct with respect to the asynchronous semantics of $\mathcal{S}$.

Lemma 6. *For every execution* $e \in \mathcal{S}'_k \,||\, \mathcal{M}_{causal}$*, we have that* $\sigma(e) \in \text{asEx}(\mathcal{S})$.

The reverse of the lemma above is also true, modulo permutations.

Lemma 7. *For every borderline violation* $e \in \text{asEx}(\mathcal{S})$ *to* k*-synchronizability, there exists an execution* $e' \in \mathcal{S}'_k \,||\, \mathcal{M}_{causal}$*, such that* $\sigma(e')$ *is a permutation of* e.

E.4 Detecting Synchronizability Violations

```
function conflict: ℙ ∪ {⊥}
function lastIsRec: 𝔹
function sawRS: 𝔹
function count: ℕ

rule send_i(p, π, (q, v)) · rec_i(π, (q, v)):
  conflict := p
  count := k

//for every i, dest(s_i) ≠ π and proc(s_i) ≠ π
rule s_1 · ... · s_n · r_1 · ... · r_m:
  for i = 1 to n
    if ( * ∧ ∃j. s_i ↦ r_j ∧ conflict ∈ {proc(s_i), dest(s_i
      if ( * )
        conflict := proc(s_i)
        if (lastIsRec)
          sawRS = true
        lastIsRec := false
      else
        conflict := dest(s_i)
        lastIsRec := true
      count--
    if ( * ∧ proc(s_i) = conflict ∧ ∀j. ¬s_i ↦ r_j )
      count--
      lastIsRec := false

rule send_i(π, q, v) · rec_i(q, v):
  assert conflict = q ⟹ (count > 0 ∧ ¬sawRS)
```

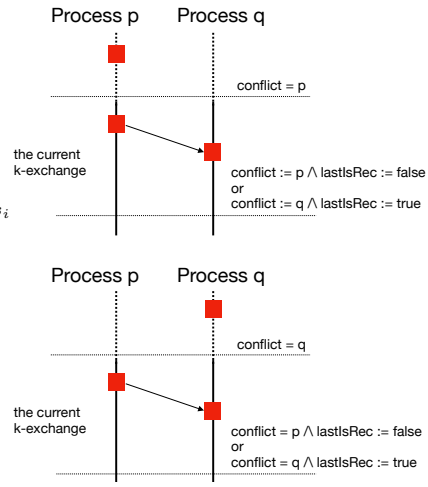


Fig. 16: The monitor $\mathcal{M}_{viol}(k)$. We use $\mathbb{B}$ to denote the set of Boolean values and $\mathbb{N}$ the set of natural numbers. Initially, `conflict` and `receiver` are $\perp$, while `lastIsRec` and `sawRS` are `false`.

Figure 16 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 Figure 16 (advancing the conflict-graph path with an unmatched send doesn't modify the value of `conflict`, it just decrements the value of `count`). In principle, 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 process 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 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.

F Proofs of Section 7

Theorem 9. *For a finite-state system $\mathcal{S}$, the reachability problem under the k-synchronous semantics is decidable and PSPACE-complete.*

Proof. A consequence of the fact that the product emptiness problem (checking if the product of a set of finite state automata has an empty language) is PSPACE-complete [24]. The evolution of the B component of a synchronous configuration and the set of messages sent during a k-exchange transition can be modeled using an additional labeled transition system that is composed with the processes in the system.

Theorem 10. *The problem of checking k-synchronizability of a finite-state system $\mathcal{S}$ is decidable and PSPACE-complete.*

Proof. Theorem 2 and Theorem 9 imply that the problem is in PSPACE. Moreover, PSPACE-hardness follows from the fact that the product emptiness problem can be reduced to checking 1-synchronizability. Given a set of finite state automata A_1, ..., A_n, we define a message passing system $\mathcal{S}$ containing one process p_i for each automaton A_i, which "simulates" the product. Essentially, p_1 is obtained from A_1 by rewriting every transition label a to $\text{send}(p_1, p_2, a_1) \cdot \text{rec}(p_1, a_n)$, the process p_i with $1 < i < n$ is obtained from A_i by rewriting every transition label a to $\text{rec}(p_i, a_{i-1}) \cdot \text{send}(p_i, p_{i+1}, a_i)$, and p_n is obtained from A_n by rewriting every transition label a to $\text{rec}(p_n, a_{n-1}) \cdot \text{send}(p_n, p_1, a_n)$. This rewriting ensures that every transition of the product of $A_1 \times \ldots \times A_n$ is simulated precisely by a sequence of sends/receives:

$$\text{send}(p_1, p_2, a_1) \cdot \text{rec}(p_2, a_1) \cdot \text{send}(p_2, p_3, a_2) \cdot \ldots \cdot \text{rec}(p_n, a_{n-1})\text{send}(p_n, p_1, a_n) \cdot \text{rec}(p_1, a_n)$$

Note that every execution admitted by this system is 1-synchronous. Augmenting this system with new states and transitions to ensure that it produces a violation of 1-synchronizability exactly when each process p_i is in a final state of A_i, leads to a system which is *not* 1-synchronizable iff the product $A_1 \times \ldots \times A_n$ has a non-empty language. Therefore, the product emptiness problem is polynomial-time reducible to checking 1-synchronizability.

Theorem 11. *For a flow-bounded system $\mathcal{S}$, the problem of checking if there exists some k such that $\mathcal{S}$ is k-synchronizable, is decidable.*

Proof. First, assume that there exists an execution e of $\mathcal{S}$ such that the corresponding conflict graph contains a bad cycle. Then, $\mathcal{S}$ is not k-synchronizable for $k = |e|$ (where $|e|$ denotes the length of e), and finding this k through a procedure that checks k-synchronizability for increasing values of k is clearly possible.

Now, assume that $\mathcal{S}$ admits no such execution. Then, every execution e of $\mathcal{S}$ can be permuted to a k-synchronous execution e', for some k (the lack of conflict-graph cycles with an RS labeled edge implies that the execution can be permuted to a sequence of k-exchange transition labels). Let K be a constant such that every process in $\mathcal{S}$ is K-receive bounded and K-send bounded (this constant exists because $\mathcal{S}$ is flow-bounded). We get that the number of consecutive receives in e' is bounded by $K \times |\mathbb{P}|$, and the number of consecutive sends by $K \times |\mathbb{P}|$. Otherwise, there would exist a process that performs more than K consecutive receives or more than K consecutive sends before a receive, which contradicts the definition of K. Therefore, e' can be executed by a sequence of $K \times |\mathbb{P}|$-exchange transitions.