Analysis and verification of concurrent programs under mailbox semantics

Romain Delpy, University of Bordeaux

December 12, 2023

My internship

My internship took place at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI), which is a public IT research laboratory, linked to the *Université de Bordeaux*. I worked among the research department of *Formal Methods and Models*, in the team *Models & Technologies for Verification*. The main goal of the team is to ensure the accuracy and reliability of complex software and systems by using formal methods. It contributes to various fields like cybersecurity, embedded systems, and artificial intelligence.

I started my internship by reading the article *On the completeness of verifying message passing programs under bounded asynchrony* [2]. I used this article to get familiar with the notions of mailbox semantics, and of k-synchronizability. I also prepared a 40 minutes oral presentation on the content of this paper for a course.

I then worked on formal definitions for the mailbox semantics. With these new definitions, we obtained new results on k-synchronizability. I also worked on a comparison of the mailbox semantics with the peer-to-peer semantics. I also assisted to several seminars within the team.

Research in my area has no direct societal or environmental impact, but I became interested in this kind of actions in the larger context of LaBRI. In this respect, several actions have been taken within LaBRI to better understand the environmental challenges and to better address issues of inclusiveness and equity.

A dedicated committee directed by Marthe Bonamy was created in 2021 to answer some of these challenges. Since then, the *Equity, Diversity and Inclusion Charter* has been written by this comitee. Its goal is to emphasize undesirable situations and behaviors, to help people spot and avoid them.

This committee also drives an environmental impact analysis of the LaBRI, to emphasize the elements of a researcher's daily life that have the biggest impact on the environment. With different poster campaign, the committee gives hints of how we can reduce our own impact, for example by preferring train to plane when travelling for a conference.

Contents

0.1 Introduction

With the last two decades' progress on new processors, multi-threaded programs are becoming more widespread. Threads are lightweight processes that interact with each other via shared memory, synchronization mechanisms (such as locks), or asynchronous message passing. In asynchronous message passing systems, we have multiple processes running simultaneously that can send messages to one another, without any guarantee on the time of reception. The action of sending a message for a process is non-blocking, meaning it does not wait for the message to be received before continuing its actions. Using multiple threads reduce the computation time, by parallelizing tasks. However, this principle makes conception and verification of multithreaded programs a challenge. In the general case, it is undecidable to check for reachability in asynchronous message passing systems [3]. However, it has been shown that if we have some restrictions over the system, we can have certain results on accessibility. For example, it has been shown by Bouajjani et.al. [2] that if the behavior of a system is composed of bounded interaction phases, then we can check for accessibility.

One particular aspect of the asynchronous message passing systems studied in [2] is that every process receives its messages in a *single* FIFO buffer. This so-called mailbox semantics can be found, for instance, in the concurrent programming languages `Erlang` and `Rust`, and it differs from the usual Peer-to-Peer semantics that can be found in distributed programming and that has already been studied [5].

During my internship I have read three papers on the subject of these bounded interaction phases [2, 4, 6], and I have extended the results presented in these papers. The main notion I worked on is k-synchronizability of a system, and its extensions. But the definition of these interaction phases in the previous works stipulated that in one phase, all the send actions are done before all corresponding receiving actions. The first task was to explore the results presented in these papers without this restriction, and clarify some of the definitions used in [2, 4, 6] . The second task consisted in making the decidability proof for k-synchronizability for *some* k given in [6], more accessible.

In this report I develop the ideas and progress that I made together with my supervisors. In Chapter 1, I give definitions for describing our systems that will be used throughout this paper. Chapter 2 deals with new definitions for the bounded interaction phases (called k-exchanges) that differ from the ones given in previous work on the subject, and explain how we can decide if a system is k-synchronizable or not, with a different proof from the previous papers and with less restriction on the structure of the k-exchanges (they don't need to have all their send actions before all their receive actions). We also prove that for a given system, finding a k such that it is k-synchronizable is undecidable in general. Finally, Chapter 3 presents our results on k-synchronizability using a definition of a k-exchange close to the one given in [2, 4, 6]. We prove that in this case, finding a k such that a system is SR-k-synchronizable is decidable in `PSPACE`, by showing how we can first check if a system is SR-synchronizable (i.e. has all

its executions equivalent to sequences of SR-exchanges), and then checking if the size of the SR-exchanges are bounded. We conclude by giving an overview of all the results on mailbox semantics we have seen in this paper, and present future works we intend to do on the subject.

Chapter 1

Definitions

In this chapter, we will introduce the basic elements that will be used throughout this manuscript, such as *communicating finite state machine* and *message sequence chart*.

1.1 Preliminaries

A communicating finite state machine [3] is a finite set of processes (represented as automata) that exchange messages. In this paper, we mainly focus on the Mailbox semantics. In this semantics, each process has a mailbox, which is a FIFO list. Messages are instantaneously stored in the mailbox when sent. Let $\mathcal{P}$ be a set of processes, and M be a set of message contents. The actions of the processes are either sending or receiving a message. For $p, q \in \mathcal{P}$, we denote a send from p to q by $p!q$, and a receive of q by $q?\_$. We then define a local alphabet for each $p \in \mathcal{P}$ for those actions, denoted by $\Sigma_p = \{\ p!q(m)\ \mid\ q \in \mathcal{P}\backslash\{p\}\,,\, m \in M\ \} \cup \{\ p?\_(m)\ \mid\ m \in M\ \}$. We also call $S_p = \{\ p!q(m)\ \mid\ q \in \mathcal{P}\backslash\{p\}\,,\, m \in M\ \}$ the set of all send actions on p and $R_p = \{\ p?\_(m)\ \mid\ m \in M\ \}$ for the receive actions on p. We will also need to specify actions directed to a certain process, so we define the set $S_{\to p} = \{q!p(m)\ \mid\ q \in \mathcal{P}\backslash\{p\}\,,\, m \in M\ \}$. We then write $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$. Moreover we define the two following subsets of Σ : $S = \bigcup_{p\in\mathcal{P}} S_p$ and $R = \bigcup_{p\in\mathcal{P}} R_p$.

Definition 1 (Communicating Finite-State Machine)
A *communicating finite-state machine* (CFM) is a tuple $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$, where $\mathcal{A}_p = (L_p, \to_p, i_p)$ is a finite labeled transition system over the alphabet Σ_p with set of states (locations) L_p and initial state $i_p \in L_p$. States from L_p are also denoted as *local states*.

An example of a CFM is shown in Figure 1.1, representing a lock of a tank, with two buttons, one activating the filling sequence, the other the emptying sequence of the tank.

One can then represent the possible configurations of a CFM by a sequential transition system (possibly infinite) whose states consist of a tuple containing the local state of each process and the content of each mailbox.

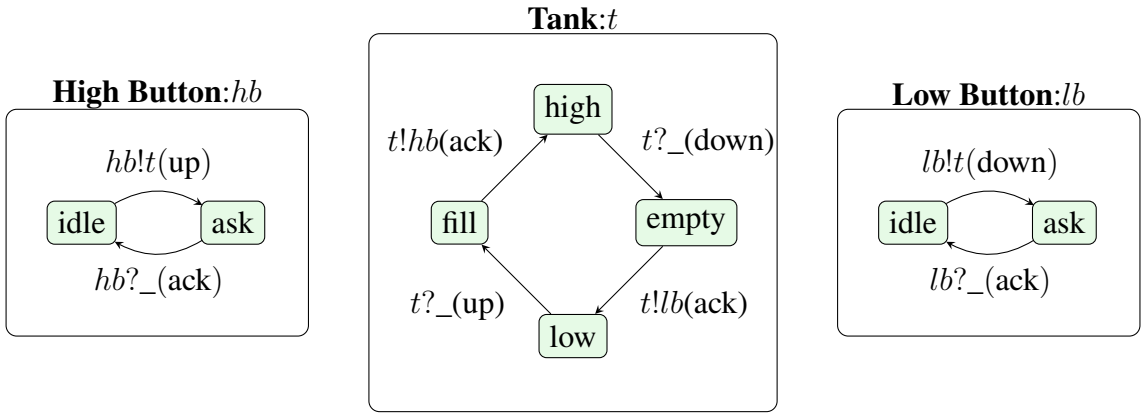


Figure 1.1: An example of a CFM representing a lock of a tank

Definition 2 (Global Transition System)
Let $\mathcal{A} = (\mathcal{A}_p)_{p\in\mathcal{P}}$ be a CFM. The *global transition system* of $\mathcal{A}$ is the tuple $T_\mathcal{A} = (C_\mathcal{A}, \to_\mathcal{A})$ where:

- $C_\mathcal{A} = \prod_{p\in\mathcal{P}}(L_p \times M^*)$ is the set of configurations of the global transition system, consisting of a local state and a mailbox for each process.
- $\to_\mathcal{A} \subseteq C_\mathcal{A} \times \Sigma \times C_\mathcal{A}$ is the set of transitions of the global transition system where:

$$((l_p, w_p)_{p\in\mathcal{P}}) \xrightarrow{a(m)} ((l'_p, w'_p)_{p\in\mathcal{P}})$$

  is in $\to_\mathcal{A}$ if

  - $l_p \xrightarrow{a(m)}_p l'_p$ and $l_q = l'_q$ for $q \neq p$.
  - Send actions: if $a = p!q$ then $w'_q = w_q \cdot m$ and $w'_p = w_p$ for $p \neq q$.
  - Receive actions: if $a = p?\_$ then $m \cdot w'_p = w_p$ and $w'_q = w_q$ for $q \neq p$.

A run of $T_\mathcal{A}$ is a sequence $\rho = c_0 \xrightarrow{a_1(m_1)} c_1 \ldots \xrightarrow{a_n(m_n)} c_n$ with $c_i \in C_\mathcal{A}$, and $a_i \in \Sigma$ such that $c_{i-1} \xrightarrow{a_i(m_i)} c_i$ is in $\to_\mathcal{A}$ for every i. It is initial if $c_0 = ((i_p, \varepsilon)_{p\in\mathcal{P}})$. We define $L(T_\mathcal{A}) \subseteq \Sigma^*$ as the set of projections of the initial runs on their actions.

Remark. We will work most of the time with sequences of actions instead of runs, so taking projections and forgetting the configurations. Moreover, we might have to work on parts of the actions of a sequence. To make reading easier, we introduce the notation $u_{|E}$ for the projection of the sequence u over the set E. For example $u_{|S_{\to p}}$ is the sequence composed of the send actions to process p in u.

1.2 Partial orders and MSCs

We will now introduce an extension of the partial order over send/receive events as presented in [2]. First we have a natural total order over send/receives of a CFM:

Definition 3 (Trace)
A *trace* of a CFM $\mathcal{A}$ is a sequence $u \in \Sigma^*$ such that there exists an initial run ρ over $T_{\mathcal{A}}$ with its projection over its actions equal to u. The set of all traces of $\mathcal{A}$ is denoted $\mathrm{Tr}(\mathcal{A})$.

As we will have to deal with parts of traces, we also need the following definition:

Definition 4 (Viable sequence, unmatched sends)
A sequence of actions $v \in \Sigma^*$ is called *viable* if for every process $p \in \mathcal{P}$:

- for every prefix u of v the number of receives on p in u is less or equal the number of sends to p in u: $|u_{|R_p}| \leq |u_{|S_{\to}p}|$.
- for every k, if the k-th receive on p is labeled by $p?\_(m)$ then the k-th send to p is labeled by $q!p(m)$ for some q.

If $w = uav$ is a viable sequence, $a \in S_{\to}p$ and $|w_{|R_p}| \leq |u_{|S_{\to}p}|$ then we refer to event a as *unmatched* send (i.e. the send is not associated with a receive action). It is said *matched* otherwise.

Remark. Clearly, every trace is a viable sequence.

Now, and in order to talk easily about equivalent sequences, we first adapt the notion of partial order diagrams called *message sequence charts* that are used in P2P communication (see e.g. [5]) to the mailbox semantics. We define a message sequence chart (MSC) as a Σ-labeled poset $(E, \leq, \lambda)$. The partial order $\leq$ is generated by three relations: the total order over events belonging to the same process, the message relation, and a causal relation related to the mailbox semantics. It is convenient to use below the function $P : E \to \mathcal{P}$ associating each event with the process on which it occurs (i.e., $P(e) = p$ if $\ell(e) \in \Sigma_p$).

Definition 5 (Message Sequence Chart)
A *message sequence chart* (MSC) is a Σ-labeled partially ordered set $M = (E, \leq, \lambda)$ of events with $\leq = (\leq_P \cup \mathrm{msg} \cup \leq_{\mathrm{mb}})^*$, where $\leq_P, \mathrm{msg}, \leq_{\mathrm{mb}}$ are the least relations satisfying the following conditions:

1. For every $p \in \mathcal{P}$, the set $\{e \mid P(e) = p\}$ of events on process p is totally ordered by $\leq_P$.
2. $(e, f) \in \mathrm{msg}$ implies $\lambda(e) = p!q(m)$, $\lambda(f) = q?\_(m)$ for some $p, q \in \mathcal{P}$, $p \neq q$, and $m \in M$; moreover, the partial function $\mathrm{msg} : \lambda^{-1}(S) \to \lambda^{-1}(R)$ is one-to-one, i.e. every event labeled by a receive action is paired with one and only one send action.

3. $e \leq_{\mathrm{mb}} f$ implies $\lambda(e) = p!q(m)$, $\lambda(f) = p'!q(m')$ for some $p, p', q \in \mathcal{P}$, $m, m' \in M$ and e is matched.

4. For every $q \in \mathcal{P}$, the set $\{e \mid \lambda(e) = p!q(m)$ for some $p \in \mathcal{P}, m \in M$ and e is matched$\}$ of matched sends to process q is totally ordered by $\leq_{\mathrm{mb}}$. Moreover, if e, e' are such that $\lambda(e) = p!q(m)$, $\lambda(e') = p'!q(m')$ for some $p, p', q \in \mathcal{P}$, $m, m' \in M$, and $(e, f) \in \mathrm{msg}, (e', f') \in \mathrm{msg}$, then $e \leq_{\mathrm{mb}} e'$ if and only if $f \leq_P f'$.

5. If e, e' are such that $\lambda(e) = p!q(m)$, $\lambda(e') = p'!q(m')$, e is matched and e' is unmatched, then $e \leq_{\mathrm{mb}} e'$.

Remark. Compared to the usual definition of MSC in P2P semantics [5] we added the relation $\leq_{\mathrm{mb}}$, which expresses the *causal order* between sends to the same process induced by the mailbox semantics. The relation $\leq_{\mathrm{mb}}$ captures precisely the *causal delivery* condition used in [2, 4]. {added} Note also that condition 4 together with the partial order implies that messages between any pair of processes are received in the same order as they were sent (FIFO property).

If $u = a_0 \dots a_n$ is a viable sequence of actions, then we can associate an MSC with u by letting $\mathrm{MSC}(u) = (E, \leq, \lambda)$ with $E = \{e_0, \dots, e_n\}$, $\lambda(e_i) = a_i$, and $\leq_P, \mathrm{msg}, \leq_{\mathrm{mb}}$ defined as expected:

- If $P(e_i) = P(e_j)$ then $e_i \leq_P e_j$ iff $i \leq j$.
- For the msg relation, we count the number of sends to a process and the number of receives on this process, and we pair a send to a receive if the number of previous sends corresponds to the number of previous receives: if $\lambda(e_i) = p!q(m)$, $\lambda(e_j) = q?\_(m)$ and

$$|\{e_k \mid \lambda(e_k) = p'!q(*), k < i, p' \in \mathcal{P}\}| = |\{e_\ell \mid \lambda(e_\ell) = q?\_(*), \ell < j\}|$$

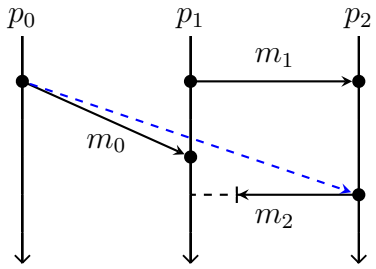
  then $(e_i, e_j) \in \mathrm{msg}$.
- For the relation $\leq_{\mathrm{mb}}$ let k be the number of receives on process p in u. If e_i is among the first k sends to p, then $e_i \leq_{\mathrm{mb}} e_j$ for every $i < j$ such that e_j is a send to p, too.

We define $\mathrm{MSC}(\mathcal{A}) = \{\mathrm{MSC}(u) \mid u$ is trace of $\mathcal{A}\}$. A *linearization* of an MSC $(E, \leq, \ell)$ is a linear extension of $\leq$. An example of a sequence of actions and its corresponding MSC is given in Figure 1.2.

Two sequences $u, v \in \Sigma^*$ are *equivalent* if $\mathrm{MSC}(u) = \mathrm{MSC}(v)$, and we write $u \equiv v$ in this case.

The interest of our new definition of MSC is the following lemma:

$$p_0!p_1(m_0)\, p_1!p_2(m_1)\, p_2?\_(m_1)\, p_2!p_1(m_2)\, p_1?\_(m_0)$$



The dashed arrow represents the mailbox order between the two sends of m_0 and m_2. An unmatched send action is marked by a special arrowhead, as for m_2.

Figure 1.2: A sequence and its MSC

Lemma 1. *Every linearization of an* MSC *is a viable sequence.*

Proof. To prove the lemma we work by induction on the size of the MSC (i.e., the size of E).

The minimal MSC is $N = (\emptyset, \emptyset, \emptyset)$. The only linearization of N is the empty word ε and we are done.

Let N be an MSC. Then let $z = e_1 e_2 \dots e_n$ be a linearization of N. We construct a smaller one by removing e_n. We have that e_n is either an unmatched send or a receive. We define N' as the MSC where we remove e_n (it is also an MSC as e_n is maximal in N). Since N' is strictly smaller than N and $z' = e_1 e_2 \dots e_{n-1}$ is a linearization of N', z' it is a viable sequence.

We then know that all prefixes v of z up to z' respect the first rule of a viable sequence:

$$|v_{|R_p}| \leq |v_{|S_{\to p}}| \quad \text{for all } p \in \mathcal{P}.$$

Moreover, since N is an MSC, we know that each event labeled with a receive action on p is paired with an unique event labeled with a send action to p, so:

$$|z_{|R_p}| \leq |z_{|S_{\to p}}| \quad \text{for all } p \in \mathcal{P}.$$

We now need to check the second property of a viable sequence. We will differentiate two cases:

- $e_n = p!q(m)$: We know that e_n is an unmatched send, as it is last in z. We also know that the property holds in z', and e_n is added after all the other send actions to q. So the pairing of the receives on q is the same in z. The property holds in z.

- $e_n = p?\_(m)$: As z is a linearization of N, there exists a send action s paired to e_n by msg. Moreover, the number of send actions to p before s is equal to the number of receive actions on p before e_n in N. If we remove e_n in z, s becomes the first unmatched send to p.We know that the property holds in z', so there are k receives actions on p in z' that are well matched to the k first send actions to p, for a given k. In z, we then know that s is the $k+1$-th send action to p and e_n is the $k+1$-th receive action on p. The property holds. □

Chapter 2

On k-synchronizability

In this chapter, we define a restriction over the structure of the executions of a system, called k-synchronizability, and show how to decide the reachability problem under this restriction.

2.1 Definitions

In this section we introduce k-synchronizable sequences of actions and give a graphical characterization.

To do so, we first define how to concatenate two viable sequences. We fix in this part a CFM $\mathcal{A}$ over an alphabet Σ.

Definition 6 (Concatenation of sequences)
Let u and v be two viable sequences. The concatenation of u, v, written as $u * v$, is defined if for every process $p \in \mathcal{P}$, if p is the destination of an unmatched send in u, then there is no receive on p in v.

Remark. Note that the operator $*$ is not associative, because of the unmatched sends. So the parentheses in $u = ((u_0 * u_1) * u_2) * \dots u_n$ will be implicit in the following.

Moreover, note that if $u_0 * \dots u_i * \dots u_j * u_{j+1} \dots u_n$ is defined then $u_0 * \dots u_i * u_{j+1} \dots u_n$ is also defined, for every $i < j$.

With this operator, one can divide a viable sequence into smaller sequences. A u be a viable sequence. It is called *divisible* if there exist non-empty viable sequences v, w such that $u \equiv v * w$, otherwise it is called *indivisible*.

Remark. Note that we use the equivalence between the sequences and not the equality. This connects our operator to the notion of concatenation for MSCs.

For the rest of this section, we fix $k \in \mathbb{N}_+$ as a positive integer.

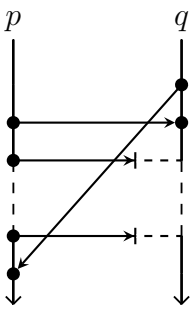


Figure 2.1: An MSC of an indivisible sequence with an unbounded number of sends.

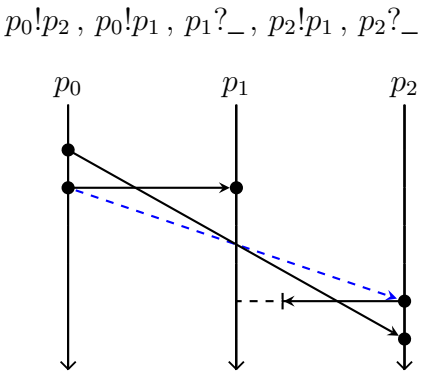


Figure 2.2: An MSC of an indivisible sequence due to mailbox order.

Definition 7 (k-exchange)
A *k-exchange* is a viable sequence of actions that contains at most k sends.

Remark. As every receive in a viable sequence is matched, any k-exchange has size at most $2k$.

As explained afterwards, our goal is to have a bound on the size of k-exchanges. Note that we cannot count just the receives, as we could have an unlimited number of unmatched sends in such a k-exchange, as shown by Figure 2.1.

Definition 8 (k-synchronizability)
Let u be a viable sequence of actions. It is called k-synchronous if there exist $u_0, u_1 \ldots, u_n$ such that $u = u_0 * u_1 * \ldots u_n$, and every u_i is a k-exchange.
A viable sequence is *k-synchronizable* if it is equivalent to some k-synchronous sequence.
A CFM $\mathcal{A}$ is *k-synchronizable* if every trace from $\mathrm{Tr}(\mathcal{A})$ is k-synchronizable.

In order to decide if a CFM is k-synchronous, one can remark that some sequences of actions are indivisible, such as:

$$p!q\,,\ q!p\,,\ q?\_\,,\ p?\_$$

By identifying such indivisible subsequences and counting the number of sends one can tell if a system is k-synchronous or not.

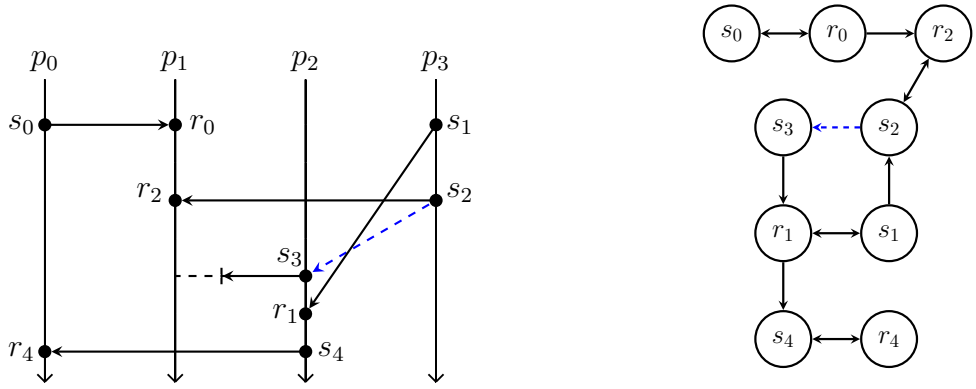


Figure 2.3: An MSC and its corresponding interlocking graph

Remark. The notion of indivisible sequences was already introduced in [6], where they are called prime. However, this notion goes back to MSCs with P2P semantics, where such MSCs are called *atomic* [7].

It is also worth noting that our additional order $\leq_{\mathrm{mb}}$ makes some non-prime sequences in [4] indivisible in our sense. An example of such a sequence is given in the right part of Figure 2.2. In [4], the sequence

$$p_2!p_1\,, p_0!p_2\,,\; p_2?\_\,,\; p_0!p_1\,,\; p_1?\_$$

which is a linearization of the MSC in the right part of Figure 2.2 *without the dashed arrow*, is 1-synchronous. However the above sequence cannot be written as $u_1 * \cdots * u_n$, where u_i are non-empty 1-exchanges, and there is no linearization of the MSC *with the dashed arrow* that is 1-synchronous. Our Lemma 1 says exactly which are the linearizations that should be considered for k-synchronizability.

To identify indivisible sequences we make use of the following graph associated with a viable sequence:

Definition 9 (Interlocking Graph)
Let u be a viable sequence, and $M = \mathrm{MSC}(u)$. The *interlocking graph* of u is the directed graph $G(u) = (V, E)$ where V is the set of all events of M, and the edges are defined by $(e, e') \in E$ if $e <_{\mathcal{P}} e'$ or $e \leq_{\mathrm{mb}} e'$ or $\{(e, e'), (e', e)\} \cap \mathrm{msg} \neq \emptyset$.

We give an example of an interlocking graph in Figure 2.3 One can show that indivisible sequences correspond to strongly connected components of the interlocking graph.

Lemma 2. *Let u be a viable sequence, and $G(u)$ its interlocking graph. The sequence u is k-synchronizable if and only if each strongly connected component of $G(u)$ has at most k sends. Moreover, if $C_1, \ldots, C_n$ are the SCC of $G(u)$ given in some topological sorting and u_i is a linearization of C_i, for every i, then $u \equiv u_1 * \cdots * u_n$ and each u_i is indivisible.*

Proof. We will show this lemma by double implication. Let u be a viable sequence, $M = (E_u, \leq, \lambda)$ its MSC, and $G(u)$ its interlocking graph.

For the right-to-left direction we assume that every strongly connected component of $G(u)$ has at most k send actions. We will show that we can reconstruct a viable sequence from $G(u)$ that is k-synchronous and equivalent to u, hence showing that u is k-synchronizable. First we show that the restriction $M_{|C}$ of M to any SCC C of $G(u)$ is an MSC. Let C be a SCC of $G(u)$. By definition of $G(u)$, we have that for any $(e, f) \in \mathrm{msg}(M)$, e is in C if and only if f is in C. Since $<_{\mathcal{P}}$ and $\leq_{\mathrm{mb}}$ are inherited from u, this shows that $M_{|C}$ is an MSC.

By Lemma 1 every linearization of $M_{|C}$ is a viable sequence of actions, with at most k send actions, so it is a k-exchange. Now that we can linearize every SCC of $G(u)$, we want to organize them in a sequence. For this we chose some topological order $C_1, \ldots, C_n$ of the SCC of $G(u)$. Let $u' = u'_1 \ldots u'_n$, with each u'_i being a linearization of the MSC $M_{|C_i}$. We need to show that, for every i, we can apply the operator $*$ between the sequence composed of the concatenation of the k-exchanges from u'_1 to u'_{i-1} with u'_i. Let us suppose the property true for i. We denote by $u'_{0\to i}$ the concatenation of the k-exchanges from u'_1 to u'_i. We know that $u'_{0\to i}$ and u'_{i+1} are two viable sequences. The other property needed to concatenate the two sequences is *if p is destination of an unmatched send in $u'_{0\to i}$, then there is no receive action on p in u'_{i+1}*.

Let us suppose this property is false. Then there exists a $j \leq i$ such that there is an unmatched send to p in u'_j, and we will call this unmatched send e_j. There is also a receive action on p in u'_{i+1}. As u'_{i+1} is a viable sequence, this receive action is paired to a send action. We will call this send action e_{i+1}. As e_j is unmatched and e_{i+1} is matched, and they both have p as destination, we have that $e_{i+1} \leq_{\mathrm{mb}} e_j$. So there is an edge from C_{i+1} to C_j in $G(u)$. This is a contradiction to the topological sorting of SCC. So we can apply the $*$ operator between $u'_{0\to i}$ and u'_{i+1}. Thus $u' = u'_1 * \ldots u'_n$ is a k-synchronous sequence of actions. It is equivalent to u, therefore u is k-synchronizable.

Now we show the left-to-right implication. Let us suppose that u is k-synchronizable. We then have a sequence $u' = u'_1 * u'_2 * \ldots u'_n$, equivalent to u such that every u'_i is a k-exchange. We will show that every strongly connected component of $G(u)$ is included in a k-exchange. To do so we will assign a rank to every event $\pi : E_u \to \mathbb{N}$ in u, corresponding to the index of the k-exchange containing it. Note that for any two events e and f we have

1. $(e, f) \in \mathrm{msg}(M)$ or $(f, e) \in \ msg(M) \implies \pi(e) = \pi(f)$.
2. $e \leq_{\mathcal{P}} f$ or $e \leq_{\mathrm{mb}} f \implies \pi(e) \leq \pi(f)$.

Now we want to show that for every SCC C of $G(u)$, the rank of every event in C must be the same. First let take a cycle in $G(u)$, composed of the events $e_0 \to e_1 \to \dots e_n \to e_0$. For each pair of events e_i and e_{i+1} next to each other in the cycle, there is an edge, so either they are in the same message, or there is an order from e_i to e_{i+1} in M. Either way, $\pi(e_i) \leq \pi(e_{i+1})$. By applying this to every edge of the cycle, we get $\pi(e_0) \leq \pi(e_1) \leq \dots \pi(e_n) \leq \pi(e_0)$. So the rank of every event of the cycle is the same. So the events of a SCC of $G(u)$ share the same rank. Therefore every SCC of $G(u)$ is included in some k-exchange, and has thus at most k sends actions.

The last claim of the lemma follows easily: $u \equiv u_1 * \dots * u_n$ was shown in the first half of the proof, whereas u_i being indivisible was shown in the second part. □

2.2 Detecting k-synchronizability

Now that we have defined and characterized k-synchronizable traces, we need to show how we can detect them in a CFM. To do so, we will first show that if the CFM is non-k-synchronizable, there is some trace of bounded length that is not k-synchronizable, and then we will give an algorithm to detect such traces using polynomial memory.

First we know that our CFM has a finite number of processes, and each process has a finite number of states. This means the number of possible global states of our CFM is bounded. We can see that if we take a trace which is long enough, there will be two identical global states. By removing all the events between those global states, and taking care of the contents of the mailboxes, we can still obtain a viable trace. From this idea we get the following lemma:

Lemma 3. *Let $\mathcal{A}$ be a CFM. If $\mathcal{A}$ is not k-synchronous, then it has some trace of size at most $2k \cdot |\mathcal{P}| \cdot L^{|\mathcal{P}|} \cdot (k+2) + 2k \cdot L^{|\mathcal{P}|}$ that is not k-synchronizable, with $L = \max(|L_p| \mid p \in \mathcal{P})$.*

We call this trace a witness of the non-k-synchronizability of our CFM.

Proof. Let $\mathcal{A}$ be not k-synchronous, u a minimal length witness for the non-k-synchronizability of $\mathcal{A}$, and $G(u)$ the interlocking graph of u. As u has minimal length, we can assume it is a *-product of k-exchanges, followed by a sequence that is indivisible and has more than k send actions. So we can write $u = u_0 * u_1 * \dots u_m * v$ with each u_i a k-exchange, $0 \leq i \leq m$, and v the sequence with more than k sends actions.

First we consider the size of $u_0 * u_1 * \dots u_m$. If we have more than $2k \cdot L^{|\mathcal{P}|}$ events before v, then $m > L^{|\mathcal{P}|}$. So we must have at least two global states that are identical, say after $u_0 * \dots * u_i$ and after $u_0 * \dots * u_j$, $i < j \leq m$. By removing all the events of $u_{i+1} * \dots * u_j$ we still have a trace ($u_0 * \dots u_i * u_{j+1} * \dots * u_m$ is well-defined, see remark after Definition 6). The new trace is strictly smaller than u, and is not k-synchronizable, so u was not a minimal witness. Thus there are at most $2k \cdot L^{|\mathcal{P}|}$ events before v.

Now we consider the size of v. First, v must end by a receive action f, since an unmatched send as last event contradicts the minimality of u. Moreover, if we remove the receive action f, u must be k-synchronizable (again by minimality). So we know that the sequence v without its last receive f is a *-product of k-exchanges. We write $v' = v_0 * v_1 * \dots v_n$ for the division in k-exchanges of v without f. We can assume that v is smallest possible, i.e. every v_i is necessarily in v; this means that v_0 contains the send event e that will be matched with f, and for every v_i, some event of v_i is reachable from v_0 and some event of v_i is co-reachable from f in $G(u)$. For brevity we say that v_i is reachable from v_0 and co-reachable from f. Let us suppose that there are more than $L^{|\mathcal{P}|} \cdot |\mathcal{P}| \cdot (k+2) \cdot (2k)$ events in v. This means that there are at least $L^{|\mathcal{P}|} \cdot |\mathcal{P}| \cdot (k+2)$ k-exchanges in v'. Thus we have a repetition of global states, and also a repetition of the processes used in the k-exchanges: there exists a process p and a global state ℓ such that there are at least $(k+2)$ k-exchanges containing an action on p and ending by the global state ℓ. We can then rewrite v' as:

$$v' = v_0 * \dots v_{p(0)} * \dots v_{p(1)} * \dots v_{p(k+1)} * \dots v_n$$

with every $v_{p(i)}$ being a k-exchange containing an action on p and ending by the global state ℓ. As they all end by the same global state, we can remove the events from the end of one of the $v_{p(i)}$ to the end of the next one and still have a viable trace. Let i be an integer. We want to remove the events from the end of $v_{p(i)}$ to the end of $v_{p(i+1)}$ (the ones highlighted below), we call the sequence without these events v'':

$$v' = v_0 * \dots v_{p(0)} * \dots v_{p(i)} * \boldsymbol{\dots v_{p(i+1)}} * \dots v_{p(i+2)} * \dots v_{p(k+1)} * \dots v_n$$

$$v'' = v_0 * \dots v_{p(0)} * \dots v_{p(i)} * \dots v_{p(i+2)} * \dots v_{p(k+1)} * \dots v_n$$

We know that we will still have a viable trace. We now show that this new trace is still not k-synchronizable. To do so, we demonstrate that we have still enough events trapped between v_0 and f. By removing some events, we may have removed paths in the graph $G(u)$. However, every k-exchange until $v_{p(i)}$ is still reachable from v_0, and every k-exchange from $v_{p(i+2)}$ to v_n is still co-reachable from f. Since $v_{p(i)}$ and $v_{p(i+2)}$ both contain some action on p, there is an edge from $v_{p(i)}$ to $v_{p(i+2)}$ in $G(u)$. So there is a path from v_0 to f going through every $v_{p(j)}$. Because of the edge from f back to e these k-exchanges< are trapped in $v''f$. This path contains at least $(k+1)$ k-exchanges, so there are at least $k+1$ sends in $v''f$ that belong to the same SCC of $u' = u_0 * \dots * u_h * v''f$. The trace u' is therefore not k-synchronizable, so u was not a minimal witness, contradiction. To conclude there are at most $L^{|\mathcal{P}|} \cdot |\mathcal{P}| \cdot (k+2) \cdot (2k)$ events in v. □

By Lemma 3 we know now that if the CFM is not k-synchronizable, then there is a bounded witness of non-k-synchronizability. We now will show how we can generate such a witness with a PSPACE algorithm. To do so, we will describe a new representation of our sequences.

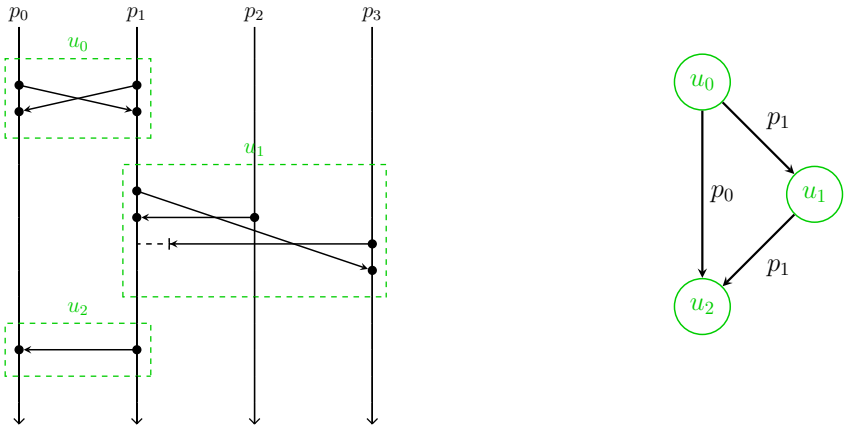


Figure 2.4: An MSC and its corresponding DAG

We need a way to display the order between the k-exchanges in a k-synchronous sequence. Let $u = u_0 * u_1 \cdots * u_n$ be a k-synchronous sequence, with every u_i being a k-exchange. We first construct a graph $H(u)$ where the vertices are the k-exchanges, and for two vertices u_i and u_j, there is an edge between u_i and u_j if and only if there is an edge from one event of u_i to an event of u_j in the interlocking graph $G(u)$. Each edge in $H(u)$ is labelled by some process: either it is a process shared by u_i and u_j, or a (receiver) process corresponding to an $\leq_{\text{mb}}$-edge. One can see that this graph is a DAG, since the vertices correspond to the SCC of the interlocking graph. We give an example of this DAG in Figure 2.4.

We can see that the maximal width of the DAG (i.e., the number of vertices that are not accessible from each other) is $|\mathcal{P}|$. Indeed, as all events on a given process are ordered, there will be a path between two k-exchanges containing actions on the same process. Moreover, if we take a minimal witness of non-k-synchronizability, we know that the last indivisible sequence ends with a receive action, and that the sequence becomes k-synchronizable if we remove this action. Another point that can be shown in this DAG is that there is only one k-exchange that has an edge going to this last receive action. Indeed this edge can only be one corresponding to a process order. We have an example of those phenomena in Figure 2.5.

So, to generate a witness of non-k-synchronizability, we need to construct a sequence of k-exchanges $v = v_0 * v_1 \cdots * v_n$ starting with a k-exchange v_0 that has an unmatched send $p!q$, and ending with a k-exchange v_n that has an action on q, such that every v_i is accessible from v_0 and coaccessible from v_n in $H(v)$, and such that $|v_{|S}| > k$. We will guess the k-exchanges in this sequence, and to reduce the amount of information we need to store to construct it, we will only remember the maximal k-exhanges (i.e., the farthest ones from v_0 in $H(v)$) at each step of the construction. As the DAG is of maximal width $|\mathcal{P}|$, we know that there will be at

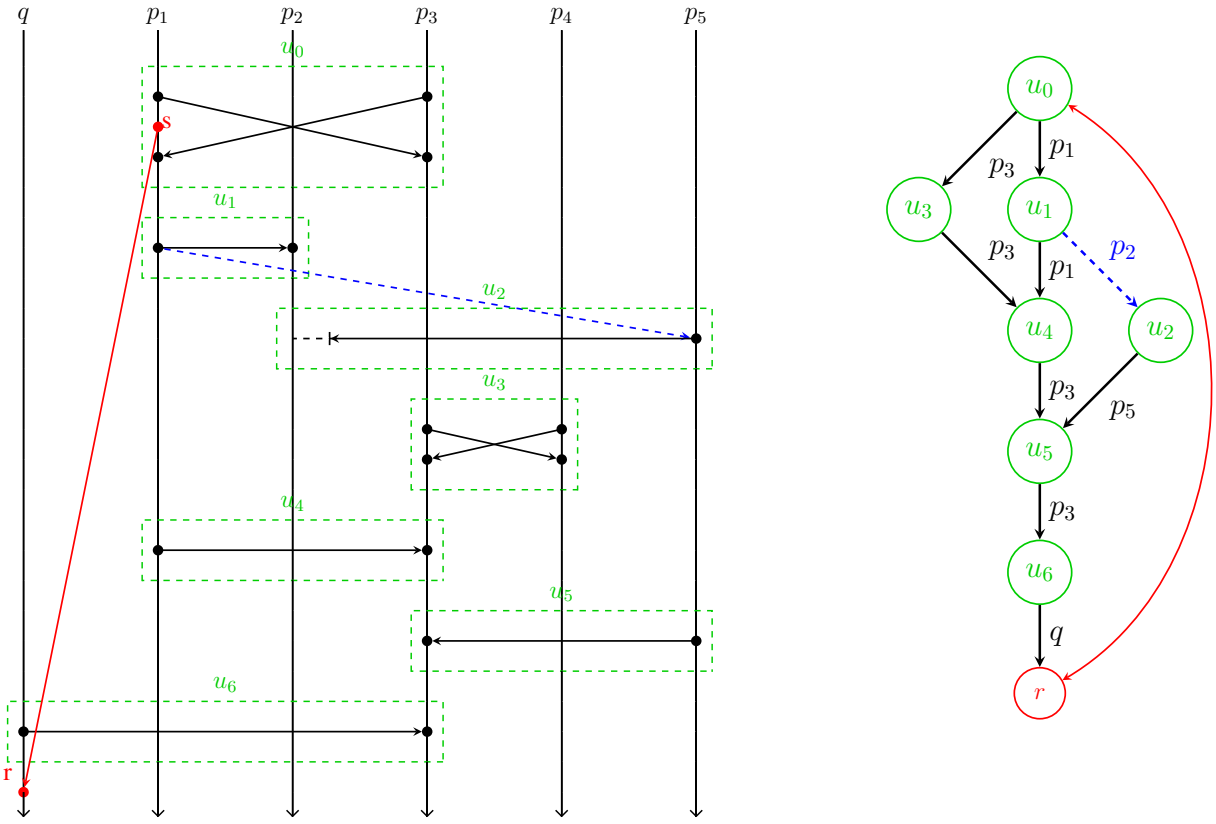


Figure 2.5: The MSC of an indivisible sequence, and the DAG constructed by removing the last receive action

most $|\mathcal{P}|$ k-exchanges stored at the same time.

We can then construct an algorithm (presented below) that takes a global state of a system and the set of blocked processes (that cannot do any receive action anymore). It will then guess a sequence of k-exchanges that have more than k send actions, and a process q such that if we add a receive action after the sequence it will make the entire sequence indivisible.

To do so, we define a function `validity_check`($\mathcal{A}$,l,B,u) that checks for the CFM $\mathcal{A}$ at global state l, if the sequence u is indivisible, and can be done with no reception on any process in $B \subseteq \mathcal{P}$ (these verifications can be done in a polynomial time). We also define a function `update`$(\mathcal{A}, l, u)$ that returns the new global state of the CFM $\mathcal{A}$ after applying the actions of u from the global state l.

We give an example of a run of this algorithm in Figure 2.6 on the sequence of the MSC from Figure 2.5, showing that this sequence is non-k-synchronizable for every $k < 10$. As

we can see, Algorithm 1 requires polynomial space. However, as we want to construct a minimal witness of non-k-synchronizability, we also need to check if the global state L is accessible through k-exchanges, and with the processes in B blocked. We can construct a PSPACE algorithm to do so, by guessing the k-exchanges needed, and by storing at each step only the global state reached and the set of processes blocked.

Theorem 1. *Let $\mathcal{A}$ be a CFM and k be given, then checking if $\mathcal{A}$ is k-synchronizable is doable in PSPACE.*

Algorithm 1 Detecting non k-synchronizability witnesses

INPUT : $\mathcal{A}$ *a CFM*,
l *a global state*,
B *a set of blocked processes without* q,
q *the process that will cause the non* k*-synchronizability*,
OUTPUT : true if there is a witness of non k-synchronizability in $\mathcal{A}$ from l, caused by the process q
guess v_0 with $|v_{0|S}| \leq k$ **s.t.** $(\exists a \in v_0, a = p!q(m), a$ *unmatched*$)$
if ! `validity_check`$(\mathcal{A},l,B,v_0)$ **then**
  STOP
end if
$Q = \{v_0\}$
$cpt = |v_{0|S}|$
$Act =$ set of all processes doing an action in v_0
$Rec =$ set of all processes doing a receive action in v_0
$B = B \cup$ set of all processes destination of an unmatched send action in v_0
$l =$ `update`$(\mathcal{A}, l, v_0)$
repeat
  guess w with $|w_{|S}| \leq k$ **s.t.** there is $a \in w$ with $P(a) \in Act$ **or** $a = p!q$, is unmatched and $q \in Rec$
  if ! `validity_check`$(\mathcal{A},l,B,w)$ **then**
    STOP
  end if
  $cpt+ = |w_{|S}|$
  $Act = Act \cup$ set of all processes doing an action in w
  $Rec = Rec \cup$ set of all processes doing a receive action in w
  $B = B \cup$ set of all processes destination of an unmatched send action in w
  for all $v_i \in Q$ **do** // Removing the non-maximal elements in Q
    if $\exists e_i \in v_i, e_w \in w$ **s.t.** $e_i \leq_{\mathcal{P}} e_w$ or $e_i \leq_{\text{mb}} e_w$ **then**
      $Q = Q \backslash \{v_i\}$
    end if
  end for
  $Q = Q \cup \{w\}$
  $l =$ `update`$(\mathcal{A}, l, v_0)$
until Q contains only one sequence that has an action on process q **and** $cpt > k$ **and** q can do the action $q?\_(m)$ from l
return true

	init	u_1	u_2	u_3	u_4	u_5	u_6
$Q =$	$\{u_0\}$	$\{u_1\}$	$\{u_2\}$	$\{u_2, u_3\}$	$\{u_2, u_4\}$	$\{u_5\}$	$\{u_6\}$
$cpt =$	3	4	5	7	8	9	10
$Act =$	$\{p_1, p_3\}$	$\{p_1, p_2, p_3\}$	$\mathcal{P}\backslash\{p_4\}$	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$	$\mathcal{P}$
$Rec =$	$\{p_1, p_3\}$	$\{p_1, p_2, p_3\}$	$\{p_1, p_2, p_3\}$	$\mathcal{P}\backslash\{p_5\}$	$\mathcal{P}\backslash\{p_5\}$	$\mathcal{P}\backslash\{p_5\}$	$\mathcal{P}\backslash\{p_5\}$
$B =$	$\{q\}$	$\{q\}$	$\{q, p_2\}$	$\{q, p_2\}$	$\{q, p_2\}$	$\{q, p_2\}$	$\{q, p_2\}$

Figure 2.6: An run example of Algorithm 1 on one linearization of the MSC in Figure 2.5

2.3 Undecidability of $*$-synchronizability

In the previous section, we gave a PSPACE algorithm to check if, for a given k, the CFM $\mathcal{A}$ is k-synchronizable. The next question is *"does there exist k such that $\mathcal{A}$ is k-synchronizable?"* This problem will be called *$*$-synchronizability*.

We will now show that this question is undecidable. To do so, we will reduce the halting problem on a Minsky machine with 2 counters to the problem of $*$-synchronizability.

Definition 10 (Minsky machine)
A *Minsky machine* [8] with counters $c_1, c_2, \ldots, c_m$ is a sequence of labeled instructions

0: $\texttt{instr}_0$;

1: $\texttt{instr}_1$;

...

n: $\texttt{instr}_n$;

Where $\texttt{instr}_n$ = HALT, and $\texttt{instr}_i$ for $0 \leq i < n$ is of the one of the forms

- $c := c + 1$; goto l, for a counter c and a label l
- if $c = 0$ then goto l_t else ($c := c - 1$; goto l_f), for a counter c, and labels l_t and l_f.

We start a computation of a Minsky machine at label 0 with all counters 0. For a given Minsky machine, there is one maximal trace. This maximal trace is finite if the machine reaches HALT, it is infinite otherwise. We have the following theorem:

Theorem ([8]). *The problem of determining if a Minsky machine with two counters initialized at 0 reaches the HALT instruction is undecidable.*

By reducing this halting problem to our problem, we will show that the later one is undecidable.

Lemma 4. *The $*$-synchronizability problem is undecidable.*

Proof. Let B be a Minsky machine with two counters c_0 and c_1. We will simulate this machine by a CFM with two processes p, q, and we will use sequences of messages to describe the values of the registers. Let $M = \{\%, a, \$, b, \#\}$ be the message contents. The word $\%a^n\$b^m\#$ represents the value of the counters of the Minsky machine B with $c_0 = n$ and $c_1 = m$. The process p will have states bearing the labels of the instructions of B, and intermediate states that we will use for our simulation. It will initialize the messages sequence (by sending the messages $\%, \$, \#$) before going to state 0, and then it will simulate the operations of the

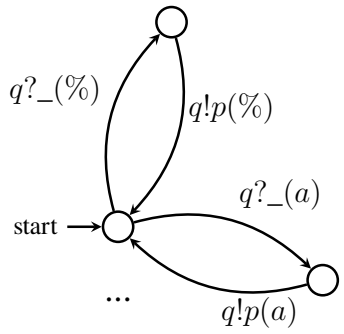


Figure 2.7: The automata representing the process q of the reduction

Minsky machine. The process q will just send back (we will say relay) every messages that it receives (we give the automata representing q in Figure 2.7).

To simulate the instruction $c := c + 1$; `goto` l, process p relays the first message $\%$ and then sends an extra a message to q, before relaying the rest of the messages received from q. When it sends the $\#$ message it goes to the state l.

To simulate the instruction `if` $c = 0$ `then goto` l_t `else (`$c := c - 1$`; goto` l_f`)`, process p relays the first message $\%$. Then if p receives $\$$ from q, it relays the rest of the messages up to $\#$, and goes to the state l_t. Otherwise, if the next message is an a, then p doesn't relay this message and it relays the rest up to $\#$. Then it goes to the state l_f.

For operations on c_1, process p does the same but it first relays all messages up to $\$$.

We know that our Minsky machine has one maximal run, and that every smaller run is a prefix of the maximal one. We have constructed our system to simulates the same set of instructions and to start with the same values as B, so for every run of B there exists an trace of our CFM where p crosses the labeled states in the same order as the run of B crosses the label of the instructions. In the other way, our system is deterministic, and it does the same operations as B, so every projection of a run of our CFM on the labeled states of p correspond to a run of B.

We will now show by double implication that our Minsky machine terminates iff there is some k such that our system is k-synchronizable.

We will start by the left-to-right direction. Let suppose that B terminates. Then the maximal run of B is finite. There is then a finite number of runs of B (as each run is a prefix of the maximal one). Let suppose we have an infinite trace in our CFM. We have constructed it to have a finite number of actions between two labeled states (as this number of action is bounded by the size of the message sequence representing the counters). So we must go through labeled states infinitely often. Or every projection of an trace of our CFM on the labeled states of p corresponds to a run in B, which contradicts the fact that all runs of B are finite. So every trace of our CFM is finite. By using König's lemma, we know that every trace of our system is bounded. Hence there exists some k such that our system is k-synchronizable,

with k being e.g. the bound on our traces.

Now for the other direction, let suppose that B does not terminate. We will show that for every k, we can construct an trace that has an indivisible sequence of actions with more than k sends. Let k be an integer. We know that for every run of our Minsky machine B, there is at least one trace in our CFM that does the same operations as this run. In this trace, after we sent the initial message sequence, we have a viable sequence corresponding to a projection of a run.

The emptiness of all the mailboxes is a necessary condition to divide a sequence composed of matched send. As we have always at least 3 messages transiting between p and q, the mailboxes are never all empty at the same time. So if we take a run of B with more than k steps, we know that we will have an trace of our CFM with an indivisible sequence with more than k messages. So this trace is not k-synchronizable, and our system is non-k-synchronizable, for any k. □

Chapter 3

Decidability of $*$-synchronizability for SR-exchanges

In this chapter, we consider k-synchronizability with SR-exchanges, as done in [2], and show how this restriction guarantees decidability for the question of $*$-synchronizability.

3.1 Definitions

We have shown previously that for our definition of a k-exchange, we can decide with a PSPACE algorithm if the system is k-synchronizable for a given k, but it is undecidable to find such a k.

However, Di Giusto et. al. have shown in [6] that the $*$-synchronizability problem is decidable *restricted to SR-exchanges*. As the definition of an MSC in their paper differs from our own, we will now re-prove this result in our setting.

For the rest of this chapter, we fix a CFM $\mathcal{A}$ over a set of message M.

Definition 11 (SR-exchange)
An *SR-exchange* is a viable sequence of actions where all the send actions are before all the receives actions. An *SR-k-exchange* is an SR-exchange with at most k send actions.

Remark. One can see how we can formulate our different types of exchanges using regular languages. A k-exchange is a viable sequence of actions in the language $(SR^*)^{\leq k}$, whereas a SR-k-exchange is a viable sequence of actions in the language $S^{\leq k}R^*$.

We say that a sequence is *SR-synchronous* if it is a $*$-product of SR-exchanges, and a sequence is *SR-synchronizable* if it is equivalent to an SR-synchronous sequence. A system is SR-synchronizable if all its traces are SR-synchronizable. The definitions of *SR-k-synchronous* and *SR-k-synchronizable* are analoguous to those of k-synchronous and k-synchronizable, up to the fact that we use SR-exchanges.

In this chapter we will show that verifying if a system is SR-synchronizable and the existence of a k such that a system is SR-k-synchronizable (or *SR-$*$-synchronizability*) are two decidable questions in PSPACE.

To show that our problems are solvable in PSPACE, we will devise a method to recognize indivisible SR-exchanges. We divide this process by creating two kinds of automata, each one checking a different property in the following list:

- The sequence is an SR-exchange and is executable in the system.
- The sequence is indivisible.

For the problem of SR-synchronizability, we will check if we can construct a sequence that is not equivalent to an SR-exchange by constructing a sequence of SR-exchanges that will be trapped after a receive action (with the same idea as what we did for Theorem 1). For the problem of SR-$*$-synchronizability, we will check if there is a loop in the product of these automata, hence permitting to inflate an indivisible exchange and making the system non SR-k-synchronizable for any k.

As the sequences we work on are meant to be SR-exchanges we show that it suffices to work with regular languages, by focusing on the send actions of our sequences. However, we still need to know which send is matched and which one is not. To this purpose we introduce marked send sequences, where symbols in S denote sends as usual, whereas symbols in $\overline{S}$ denote unmatched sends:

Definition 12 (marked send sequence)
A *marked send* sequence (*ms-sequence* for short) is a sequence of actions $v = a_0 \dots a_n$ in $(S \cup \overline{S})^*$ such that for all $i < j$ and $p \in \mathcal{P}$, $v_i \in \overline{S}_{\to p}$ implies $v_j \notin S_{\to p}$.

The mapping $\mathtt{ms} : S^*R^* \to (S \cup \overline{S})^*$ takes an SR-exchange $u = s_0 \dots s_n\ r_0 \dots r_m$ and returns its corresponding ms-sequence $\mathtt{ms}(u) = s'_0 \dots s'_n$, where $s'_i = s_i$ if s_i is matched in u, and $s'_i = \overline{s_i}$ otherwise.

We say that action a *bears* the process p if $a \in S_p \cup \overline{S_p} \cup S_{\to p} \cup \overline{S}_{\to p}$, and it *actively bears* the process p if it bears p and is not in $\overline{S}_{\to p}$. We say also that an ms-sequence v bears p if there some action of v that bears p.

Remark. It is possible to construct a function that takes a marked send sequence and returns a corresponding SR-exchange, see lemma below.

The interest behind ms-sequences is the following property:

Lemma 5. *Let u and u' be two SR-exchanges, if $\mathtt{ms}(u) = \mathtt{ms}(u')$ then $u \equiv u'$.*

Proof. Indeed, as $\mathtt{ms}(u) = \mathtt{ms}(u')$, the send actions in u and v have the same order. Thus $\mathrm{MSC}(u)$ and $\mathrm{MSC}(v)$ have the same partial order on their receives, since we work in the mailbox semantics. So $\mathrm{MSC}(u) = \mathrm{MSC}(v)$. □

3.2 Automata constructions

3.2.1 Automata for SR-exchanges

As expressed before, we first want a way to generate sequences that are SR-exchanges and can be executed in our CFM. To do so, we will construct a finite state machine (FSM) that reads ms-sequences following the transitions of $T_{\mathcal{A}}$. Each state of the FSM will be the product of two global states of $T_{\mathcal{A}}$ and a set of processes. Our construction is similar to the one in [6].

The first global state represents the local states of each process after reading send actions in the sequence, the second state corresponds to the local states of the processes after reading corresponding receive actions (of matched sends); the set of processes will represent the processes blocked due to unmatched sends. The idea is that we do not need to store messages, since due to the SR condition we can do the transitions receives in parallel to the ones on sends.

We will call $A_{SR} = (\Sigma_{SR}, H_{SR}, \delta_{SR})$ the FSM where:

- $\Sigma_{SR} = S \cup \overline{S}$ is the alphabet
- $H = L \times L \times 2^P$ is the set of states
- δ_{SR} is built as follows. Let (l, h, B) and (l', h', B') be states of H_{SR}, let $p, q \in \mathcal{P}$ and $m \in M$, then:

$$(l, h, B) \xrightarrow{p!q(m)} (l', h', B')$$

iff

$$\begin{aligned} &l_p \xrightarrow{p!q(m)}_p l'_p \text{ and } l_r = l'_r \text{ for all } r \neq p \ , \\ &h_q \xrightarrow{q?\_(m)}_q h'_q \text{ and } h_r = h'_r \text{ for all } r \neq q \ , \\ &B = B' \text{ and } q \notin B \end{aligned}$$

and

$$(l, h, B) \xrightarrow{\overline{p!q(m)}} (l', h', B')$$

iff

$$l_p \xrightarrow{p!q(m)}_p l'_p \text{ and } l_r = l'_r \text{ for all } r \neq p \ , \ h = h' \text{ and } B' = B \cup \{q\}$$

From this FSM we can construct any automaton that recognizes all ms-sequences between two global states of $T_{\mathcal{A}}$. Let l and l' be two global states of our system and B and B' two sets of processes. We can construct the automaton accepting all ms-sequences that represent SR-exchanges going from l with blocked processes B to l' with blocked processes B' in $T_{\mathcal{A}}$ by taking the union of all automata constructed from A_{SR} where the initial state is set

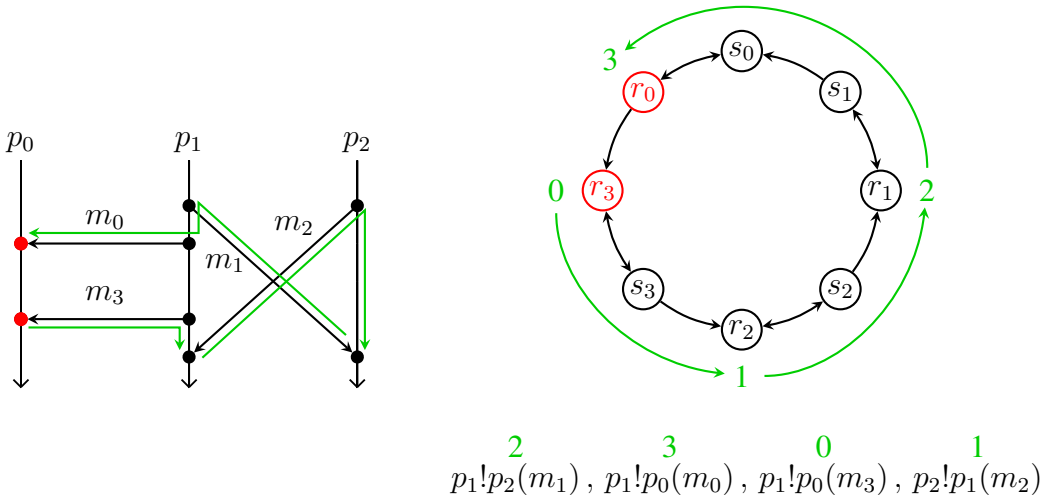


Figure 3.1: An MSC, its interlocking graph and the labeling of its ms-sequence witnessing a path from the last to the first event of process p_0

to (l, l_{int}, B) and the final state is set to (l_{int}, l', B') *for all* $l_{int} \in L$. This automaton has $|L|^3 \times 2^{|\mathcal{P}|}$ states. If we want an automaton that accepts all SR-exchanges between two global states no matter the blocked processes, we must construct the union of all the automata for any B and B'. We must multiply the number of states by $2^{2|\mathcal{P}|}$.

Lemma 6. *For a CFM $\mathcal{A}$ and two global states l and l' of $\mathcal{A}$, we can construct an automaton with $|L|^3 \times 2^{O(|\mathcal{P}|)}$ states that accepts the ms-sequences corresponding to SR-exchanges that can be executed from the global state l to the global state l' of $\mathcal{A}$.*

3.2.2 Automata for indivisible sequences

Let u be an SR-exchange and $v = \texttt{ms}(u)$. To show that u is indivisible, we will proceed in two steps: First we will show that for every process p doing an action, there is a path in the interlocking graph $G(u)$ that goes from the last action of p to the first action of p, hence showing a cycle containing all its actions, and making them indivisible. Then we will show that there is a cycle going through every process doing an action. As each process has its actions indivisible, if we have a cycle going through every process, the entire sequence must be indivisible, according to Lemma 2. Let first show how we verify the existence of a path in $G(u)$.

To show that there is a path in $G(u)$ between two events, we will label a list of actions in the ms-sequence representing the steps of the path. Indeed, we do not need the entire ms-sequence to show a path between two events. An example of such a labeling is provided in Figure 3.1. We will define a labeling of an ms-sequence v as a total function $\pi : \{0, \ldots, n\} \to \{0, \ldots, |v|\}$, where $\pi(i) = j$ means that the elements v_j of v is labeled i. To show how the labeling works,

we will distinguish two orders between the elements of our ms-sequence :

- The *direct order* corresponds to the order between two elements of the same type – two sends or two receives – due to a process order $\leq_{\mathcal{P}}$ or the mailbox order $\leq_{\mathrm{mb}}$. In the direct order, the two elements we compare are ordered in the ms-sequence, and they bear the same process as sender or receiver of the message. One such example of direct order is between element 2 and element 3 of the labeling for the sequence in Figure 3.1: $\boldsymbol{p_1}!p_2(m_1)$, $\boldsymbol{p_1}!p_0(m_0)$. Formally, for $i < j \leq |v|$ and for some p:

  $v_i \in S_p \cup \overline{S_p}$ and $v_j \in S_p \cup \overline{S_p}$ (process order), or

  $v_i \in S_{\to p}$ and $v_j \in S_{\to p} \cup \overline{S}_{\to p}$ (mailbox order)

- The *indirect order* involves a message arc. As we know that each sequence is an SR-exchange, if we have a send action and a receive action on the same process, the send action is always first. Unlike the direct order, the element we compare do not need to be ordered in the ms-sequence. Two examples of this order can be found in Figure 3.1, one is from element 0 to element 1: $\boldsymbol{p_1}!p_0(m_3)$, $p_2!\boldsymbol{p_1}(m_2)$. There is also an indirect order from element 1 to element 2 of the labeling. Note they are not ordered this way in the ms-sequence, but we use here the reverse msg edges. Formally, for i and j smaller than $|v|$, and for some p:

  $v_i \in S_p \cup \overline{S_p}$ and $v_j \in S_{\to p}$.

We see that if two elements of v are in direct or indirect order, there is a path between their events in $G(u)$. Let v be a ms-sequence, and for some n, $\pi : \{0, \ldots, n\} \to \{0, \ldots, |v|\}$ the labeling of v. Note that an element of v can be used multiple times in our labeling, but each label is used only once.

We say that v is *well-labeled* by π if for every $0 \leq i < n$:

- if $\pi(i) > \pi(i+1)$ (i.e., the element labeled i is after the element labeled $i+1$ in v) then $v_{\pi(i)} \in S_p \cup \overline{S_p}$ and $v_{\pi(i+1)} \in S_{\to p}$ (indirect order) for some p.

- if $\pi(i) < \pi(i+1)$ (i.e., the element labeled i is before the element labeled $i+1$ in v) then, for some p

  $v_{\pi(i)} \in S_p \cup \overline{S_p}$ and $v_{\pi(i+1)} \in S_p \cup \overline{S_p}$, or

  $v_{\pi(i)} \in S_{\to p}$ and $v_{\pi(i+1)} \in S_{\to p} \cup \overline{S}_{\to p}$, or

  $v_{\pi(i)} \in S_p \cup \overline{S_p}$ and $v_{\pi(i+1)} \in S_{\to p}$

We now will show the following lemma:

Lemma 7. *Let u be an SR-exchange. There is a path in the interlocking graph $G(u)$ between two elements u_i and u_j if and only if there is a well-labeling of $\texttt{ms}(u)$ starting in u_i and ending in u_j.*

Proof. Let u be an SR-exchange. We start by the right-to-left direction: if we have a well-labeling π of $\texttt{ms}(u)$ such that $\pi(0) = i$ and $\pi(n) = j$, we know that two consecutive elements of the labeling are either in direct or indirect order, showing a path in $G(u)$ between their events, hence there is a path from the first element of the labeling u_i to the last element of the labeling u_j.

Now we show the left-to-right direction. Let us suppose we have a path from u_i to u_j in $G(u)$. We construct a labeling π of $v = \texttt{ms}(u)$ that starts in the send action of u_i and ends in the send action of u_j, and we label each marked send in v in the order they are seen on the path from u_i to u_j. Let us show for any element k of our labeling, that there is a direct or indirect order between $v_{\pi(k)}$ and $v_{\pi(k+1)}$ in v. We will call s_k the send action in u corresponding to $v_{\pi(k)}$. We know that there is no send between s_k and s_{k+1} in the path of $G(u)$, so either they are consecutive, or there are receive actions between them.

- If we have no receive action between s_k and s_{k+1}, then they are either in process order, or in mailbox order in $\mathrm{MSC}(u)$. So s_k is before s_{k+1} in u, and $v_{\pi(k)}$ is before $v_{\pi(k+1)}$ in v, and they bear the same process as sender or the process actively bear as receiver by $v_{\pi(k)}$ is bear as receiver by $v_{\pi(k+1)}$. So $v_{\pi(k)}$ and $v_{\pi(k+1)}$ are in direct order in v.

- Let suppose they are receive actions between s_k and s_{k+1} in the path. We will denote r_k for the receive action linked to s_k (if there is one), and r_{k+1} for s_{k+1} (if there is one). We know that, if they are receive actions between s_k and s_{k+1} in the path, the last one must be r_{k+1}. Indeed, the only edge possible from a receive action to a send action that are not from the same message must be created by process order, and as we are in an SR-exchange, it is impossible to have a receive action before a send action. If r_{k+1} is on the same process p as s_k, then we have an indirect order between $v_{\pi(k)}$ and $v_{\pi(k+1)}$, as $v_{\pi(k)}$ bears p as an sender, and $v_{\pi(k+1)}$ actively bears p as a receiver. Otherwise, if r_{k+1} is on another process than s_k, it means that we change of process to access r_{k+1} from s_k, and as there is no send action between them, we must go through the edge between s_k and r_k. As we cannot change of process after that, it means that r_k if before r_{k+1} on the same process p. There is then a mailbox order between s_k and s_{k+1}, and as said before, there is a direct order between $v_{\pi(k)}$ and $v_{\pi(k+1)}$.

So there is a direct or indirect order between $v_{\pi(k)}$ and $v_{\pi(k+1)}$ in v, and π is a well-labeling of v. □

Automaton for indivisible actions on a process

Let u be a viable sequence, and $v = \texttt{ms}(u)$. For a process p, all events of p are indivisible if there is a well-labeling of v such that the last element actively bearing p in v is labeled 0, and the first element actively bearing p in v is also labeled (to minimize the labeling, we might want it to have the biggest label).

To construct an automaton that will check if each process respects this characterization we need a bound on the number of elements of a well-labeling:

Lemma 8. *Let u be a viable sequence of action, and $\texttt{ms}(u) = v = v_0 \dots v_n$ be its ms-sequence. For every i and j, if there is a path between an action of v_i and an action of v_j in the interlocking graph $G(u)$, then there exists a well-labeling of $\texttt{ms}(u)$ starting in v_i and ending in v_j with at most $|\mathcal{P}|^2$ elements.*

Proof. To prove this lemma, we will show that if there is a path, there exists a well-labeling with at most $|\mathcal{P}|$ indirect order edges between two labeled elements, and then we will show that the number of element between two consecutive indirect orders is bounded by $|\mathcal{P}|$. Let u be an SR-exchange and $v = \texttt{ms}(u)$.

First, let suppose we have a minimal well-labeling going from an element v_a to an element v_b of our sequence with more than $|\mathcal{P}|$ indirect orders. It means that there is a process used in at least two indirect order. Let $i < j$ be two labels preceding an indirect order on the same process p. We know that i and j represent send actions on p, and $i + 1$ and $j + 1$ receive actions on p. So we also have an indirect order between i and $j + 1$, and we can construct a well-labeling that goes from v_a to v_b without the elements from $i + 1$ to j, contradiction.

Now let us suppose that we have a minimal well-labeling going from v_a to v_b with less than $|\mathcal{P}|$ indirect orders. Let i and j be two labeled elements of v preceding an indirect order, such that $j - i > |\mathcal{P}|$, and such that there is no other indirect order used between i and j. As there are more than $|\mathcal{P}|$ elements, we must have two elements m and m' preceding a direct order on the same process p. As all the elements between i and j are in direct order, we respect the order of the ms-sequence v, so we also have an order between m and $m' + 1$ (it can be direct or indirect). So we can create a new well-labeling where we go directly from m to $m' + 1$, which is smaller than our former labeling, contradiction. □

We can then construct an automaton of exponential size that checks if, for a process p, all the events done on p in the SR-exchange u are indivisible by reading the corresponding $v = \texttt{ms}(u)$. To do so, we guess the size n of the labeling, then we construct a list that will represent our labeling. While reading the sequence v we guess the elements of the labeling. We also store the time we added them, i.e., the number of elements that were added before it in the list; this will be used to remember the order of the elements in v.

We make sure as we construct the list that the first element of the list corresponds to the last element actively bearing p in v, and that the last element of the list corresponds to the first

element actively bearing p in v. When we have read the whole ms-sequence, we check that the list corresponds to a well-labeling.

One can see how we can construct an automaton that will accept the ms-sequences that if they correspond to SR-exchanges, then for the process p, all the actions on p are indivisible. The states will be possible contents of the list. When the automaton reads an element of v, it chooses to add it in the list or not, and the final states are the ones containing a well-labeling. If the process does not do any action, the list must stay empty. The number of states is $(2 \times |\mathcal{P}| \times |\mathcal{P}| \times |M|)^{|\mathcal{P}|^2} \times (|\mathcal{P}|^2)^{|\mathcal{P}|^2}$, as we have a list of size $|\mathcal{P}|^2$ containing marked sends, and a list of size $|\mathcal{P}|^2$ containing the adding instant of each element of the list (going up to $|\mathcal{P}|^2$). As we have to do this for each process, we can do the product of all those automata, and we get an automaton that has $(|\mathcal{P}|^2 \times |M|)^{O(|\mathcal{P}|^2)}$ states.

Automaton for cycle between processes

We have now a way to check if all actions of a process are indivisible, but we also need to check if all the strongly connected components of each process are linked. To do so we will use a well-labeling between every pair of processes, checking that, if they act, there is a path from the first one to the second.

For an SR-exchange u and $v = \mathtt{ms}(u)$, where every process acting has its actions indivisible, u is an indivisible sequence if for every processes p and q doing an action in u, there is a well-labeling of v where there is some element of v actively bearing p that is labeled, and another element actively bearing q that is labeled.

For p and q two processes, as we have done for checking the indivisibility over one process, we can construct an automaton that has as states possible lists representing guessed labelings, and as final states well-labelings starting in an action of p and ending in an action of q if they both act, or an empty labeling if one of them does not. This automaton has $(2 \times |\mathcal{P}| \times |\mathcal{P}| \times |M|)^{|\mathcal{P}|^2} \times (|\mathcal{P}|^2)^{|\mathcal{P}|^2}$ states. We construct this automaton for each pair of processes, and then do the product of these automata. We obtain an automaton of size $(|\mathcal{P}|^2 \times |M|)^{O(|\mathcal{P}|^2)}$.

If we do the product of the two automata of this section, we get an automaton that accepts all ms-sequences such that, if they correspond to the projection of an SR-exchange, then this SR-exchange is indivisible:

Lemma 9. *Let $\mathcal{A}$ be a CFM. There is an automaton of size $(|\mathcal{P}|^2 \times |M|)^{O(|\mathcal{P}|^2)}$ that accepts all sequences $v \in S \cup \overline{S}$ such that every SR-exchange u with $v = \mathtt{ms}(u)$ is indivisible.*

The above automaton accepts also sequences of marked sends that do not correspond to any SR-exchange (one can even construct sequences of marked sends that does not correspond to any viable sequence). If we take the product with the first automaton of Section 3.2.1 then we accept only sequences of marked sends corrsponding to SR-exchanges. Since two

SR-exchanges having the same ms-sequence are equivalent (see Lemma 5), we can build an automaton that accepts for every indivisible SR-exchange some linearization (more precisely, the one where receives are in the same order as their sends).

Theorem 2. *Let $\mathcal{A}$ be a CFM, and l, l' two global states. We can construct an automaton of size $|L|^3 \times 2^{O(|\mathcal{P}|)} + (|\mathcal{P}|^2 \times |M|)^{O(|\mathcal{P}|^2)}$ that accepts precisely the ms-sequences of indivisible SR-exchanges that go from state l to state l' in $\mathcal{A}$.*

3.3 Results

We have shown that we can construct an automaton that accepts the ms-sequences representing indivisible SR-exchanges between two global states of $\mathcal{A}$. We also know that the accessibility in an automaton is a LOGSPACE^2 problem [1].

We will use these automata to get two results:

- Checking if a CFM is SR-synchronizable is in PSPACE.
- Checking if a CFM is SR-$*$-synchronizable is in PSPACE.

3.3.1 SR-synchronizability

To check for SR-synchronizability, we will use a similar method as the one used in Section 2.2. We will try to construct a trace that is not equivalent to a sequence of SR-exchanges. This means that this trace contains an indivisible sequence that is not equivalent to a SR-exchange.

We will call this trace a witness. This witness will be minimal when removing the last action of the sequence makes it SR-synchronizable. First, let us give a more precise definition of a minimal witness.

Definition 13
Minimal Witness Let $\mathcal{A}$ be a system. A minimal witness of non-SR-synchronizability is a trace $t = u * v$ where u is a SR-synchronizable sequence, and v is an indivisible sequence of action containing a receive action ordered before a send action, ending by a receive action such that by removing this receive action, u becomes SR-synchronizable.

Remark. One can see why the indivisible sequence ends by a receive action, and that by removing this action we get a new division of u into SR-exchanges.

We now prove the following lemma:

Lemma 10. *Let $\mathcal{A}$ be a CFM. If $\mathcal{A}$ is not-SR-synchronizable, then it has a minimal witness $t = u * v$, where v contains $O(L^{|\mathcal{P}|})$ SR-exchanges and v can be divided into at most $O(L^{|\mathcal{P}|} \cdot |\mathcal{P}|)$ SR-exchanges when its last aciton is removed.*

Proof. Let $t = u * v$ be a minimal witness. First we show that if u has more than $L^{|\mathcal{P}|}$ SR-exchanges, t is not minimal. Let suppose that u has more than $L^{|\mathcal{P}|}$ SR-exchanges, it means that there is two SR-exchanges ending in the same global state. We can then remove all the SR-exchanges between those two global states to produce u', and $u' * v$ is still a viable trace, smaller than t, ending by an indivisible non-SR-synchronizable sequence, so it is a smaller minimal witness, contradiction. So u must have less than $L^{|\mathcal{P}|}$ SR-exchanges.

Now let talk about the bound of v. By minimality v must end by a receive action f, so $v = u'f$. Moreover, $v' \equiv u''$ for some v'' that is SR-synchronizable: $v'' = u_0 * u_1 * \ldots * u_n$ with each v_i being SR-exchange. Since v is not SR-synchronizable, there must be some receive action before a send action in the same indivisible part of v. This can only be achieved by having a process that does a receive action r in some SR-exchange v_i, a send action s in a later SR-exchange v_j (so $i < j$), and v_i and v_j are indivisible in v.

Now let suppose that v' contains more that $6 \cdot L^{|\mathcal{P}|} \cdot |\mathcal{P}|$ SR-exchanges. By minimality we know that each SR-exchanges cannot be rearranged out of v. As we have more than $6 \cdot L^{|\mathcal{P}|} \cdot |\mathcal{P}|$ SR-exchanges, we have at least 6 SR-exchanges actively bearing the same process p and ending in the same global state. As expressed in Section 2.2, we can remove the SR-exchanges between two indentical state and still have a sequence of action executable in the system. We still need to prove that we still have an indivisible block by removing these SR-exchanges. We will differentiate two cases:

- If we have two of these SR-exchanges between v_i and v_j, by removing the SR-exchanges, we still have v_0 ordered before v_i, v_i ordered before v_j (as they actively bear the same process), and v_j is ordered before v_n, so we can still construct an indivisible block containing a receive action ordered before a send action

- If we do not have two of these SR-exchanges, there is at least 5 of them between v_0 and v_i and between v_j and v_n. Let suppose that there is 3 of these SR-exchanges between v_0 and v_i (without loss of generality, there is at least 3 on one side). We write $v' = u_0 * \ldots u_{p(0)} * \boldsymbol{\ldots u_{p(1)}} * \ldots u_{p(2)} * \ldots u_i * \ldots u_n$, where $v_{p(i)}$ represent the i-th SR-exchange actively bearing p and ending in the same state. If we remove the SR-exchanges between $v_{p(0)}$ excluded to $v_{p(1)}$ included (in bold above), they still are two SR-exchanges actively bearing p. We have v_0 ordered before $v_{p(0)}$, ordered before $v_{p(2)}$, ordered before v_i, and v_i is ordered before v_n. So we still have an indivisible block with a receive action before a send action.

We can then construct a smaller indivisible sequence that is not equivalent to an SR-exchange. Contradiction. □

Now that we know the shape of the minimal witness, we can construct as in Section 2.2 a DAG where the nodes are SR-exchanges, and there is an edge between two nodes if there is an order between their corresponding SR-exchanges. We can see that the maximal width

(i.e. the number of nodes we can select that does not share an edge) is $|\mathcal{P}|$. We also see that there is an initial SR-exchange containing an unmatched send $p!q(m)$, and that the DAG ends in an SR-exchange followed by a global state from which we can do the receive action $q?\_(m)$. We get from this construction the following algorithm, where we guess the ms sequences corresponding to the SR-exchanges of the witness. As we cannot store all the SR-exchanges, we just store one occurence of each marked send in the SR-exchange, which is enough information to verify the order between them.

With Algorithm 2, we can check with a polynomial space if a CFM can have a viable indivisible sequence that is not equivalent to an SR-exchange.

Lemma 11. *Let $\mathcal{A}$ be a CFM. Checking if $\mathcal{A}$ is SR-synchronizable is* PSPACE.

Remark. With this method, checking if $\mathcal{A}$ is SR-k-synchronizable for a fixed k is PSPACE and can be done by checking if it is SR-synchronizable and then using Algorithm 1.

3.3.2 SR-$*$-synchronizability

With the automata we used in this chapter, we can construct an algorithm to verify that a CFM $\mathcal{A}$ is *not* SR-$*$-synchronizable.

The algorithm does the following:

1. It first checks that the system does not have any trace that is non-SR-synchronizable (using Algorithm 2). If this is not the case, it stops and answers *not* SR-$*$-synchronizable.

2. It guesses two global states l, l' of $\mathcal{A}$ and checks that l is reachable from the initial state through an SR-synchronous sequence.

3. It checks that the automaton accepting indivisible SR-exchanges (see Section 3.2.2) leading from l to l' accepts an infinite language.

The validity is justified as follows. If $\mathcal{A}$ is not SR-$*$-synchronizable, then (1) either there exists some trace that is non-SR-synchronizable , or (2) there exists some global states l, l' that are reachable through an SR-synchronous sequence, but there can be unboundedly many indivisible SR-exchanges from l to l'. If there were boundedly many indivisible SR-exchanges from l to l', for every l, l', then $\mathcal{A}$ would be SR-k-synchronizable for some k. The algorithm is in PSPACE because PSPACE is closed under complementation, so step (1) uses the algorithm from Section 3.3.1. . Step (2) uses the same principle as the algorithm from Section 3.3.1 (constructing a sequence of SR-exchanges on the fly that goes from the initial global state to l). Step (3) works in PSPACE because the automaton is of exponential size, so we can check for a loop in PSPACE.

Theorem 3. *Let $\mathcal{A}$ be a CFM. Checking if there exists a k such that $\mathcal{A}$ is SR-k-synchronizable is* PSPACE.

Algorithm 2 Detection of non SR-synchronizability witness

INPUT : $\mathcal{A}$ *a CFM*,
l *a global state*,
B *a set of blocked processes without* q,
q *the process that will cause the non SR-synchronizability*
OUTPUT : true if there is a witness of non SR-synchronizability in $\mathcal{A}$ from l, caused by the process q

bool *RS_act* **= false** *the boolean recording if a process did a send action after a receive action*
guess (l', B') **s.t.** $q \in B'$ *making sure we have an unmatched send to* q
$v_0, U =$ `check&set`(l, B, l', B') *U is the list of the first unmatched send to each process*
if $v_0 = \emptyset$ **then**
  STOP
end if
$Q = \{v_0\}$
$p!q(m) = U[q]$
$Act =$ set of all processes actively bear in v_0
$Rec =$ set of all processes actively bear as a receiver in v_0
$B = B'$
$l = l'$
repeat
  guess (l', B')
  $w, \_ =$ `check&set`(l, B, l', B')
  if $w = \emptyset$ **or** ! `ordered`(w, Act, Rec) **then**
    STOP
  end if
  $Act = Act \cup$ set of all processes actively bear in w
  $Rec = Rec \cup$ set of all processes actively bear as a receiver in w
  $B = B'$
  $l = l'$
  for all $v_i \in Q$ **do** // Removing the non-maximal elements in Q
    if $\exists p \in \mathcal{P}$ **s.t.** p is actively bear in v_i and in w **or** $v_i \cap S_{\to p} \neq \emptyset$ and $w \cap \overline{S_{\to p}} \neq \emptyset$ **then**
      $Q = Q \backslash \{v_i\}$
    end if
  end for
  if $\exists p \in Rec$ **s.t.** $w \cap (S_p \cup \overline{S_p}) \neq \emptyset$ **then**
    RS_act **= true**
  end if
  $Q = Q \cup \{w\}$
until Q contains only one list that actively bears q **and** *RS_act* **and** q can do the action $q?\_(m)$ from l
return true

Conclusion

During my internship, I have had the opportunity to work on an aspect of automata theory and of verification which I was unfamiliar with. I got new results on the verification of concurrent programs under the mailbox semantics. To enlighten the contribution of this internship, we can classify the different results of synchronizability under mailbox semantics, using the regular language form of the exchanges presented in Section 3.1 (our results are in red):
We know that we can check accessibility in a CFM with SR-exchanges and k-exchanges, and from these results we can prove that some global states are never reachable in our CFM, if it is k-synchronizable or SR-k-synchronizable.

As I will continue to work on this subject for my PhD, I will have the opportunity to search for better results with other restrictions on asynchronous message passing systems (for example getting faster algorithms on systems that have a certain topology). Moreover, I intend to apply what I've theorized on real systems by constructing a tool that will parse a program in `Rust` and construct its corresponding CFM. I will also implement the algorithms we described above over CFM, so they can be applied on the CFMs generated from programs.

	k fixed	k unknown
$S^{\leq k}R^*$	PSPACE [4]	Decidable [6] PSPACE
$(SR^*)^{\leq k}$	PSPACE	Undecidable

Classification of results over mailbox semantics

Bibliography

[1] S. Arora and B. Barak, *Computational complexity: a modern approach*, Cambridge University Press, 2009.

[2] A. Bouajjani, C. Enea, K. Ji, and S. Qadeer, *On the completeness of verifying message passing programs under bounded asynchrony*, in Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30, Springer, 2018, pp. 372–391.

[3] D. Brand and P. Zafiropulo, *On communicating finite-state machines*, Journal of the ACM (JACM), 30 (1983), pp. 323–342.

[4] C. Di Giusto, L. Laversa, and E. Lozes, *On the k-synchronizability of systems*, in 23rd International Conference on Foundations of Software Science and Computer Systems (FOSSACS 2020), vol. 12077, Springer, 2020, pp. 157–176.

[5] B. Genest, D. Kuske, and A. Muscholl, *On communicating automata with bounded channels*, Fundamenta Informaticae, 80 (2007), pp. 147–167.

[6] C. D. Giusto, L. Laversa, and É. Lozes, *Guessing the buffer bound for k-synchronizability*, in Implementation and Application of Automata - 25th International Conference, CIAA 2021, Virtual Event, July 19-22, 2021, Proceedings, S. Maneth, ed., vol. 12803 of Lecture Notes in Computer Science, Springer, 2021, pp. 102–114.

[7] L. Hélouët and P. L. Maigat, *Decomposition of Message Sequence Charts*, 2000.

[8] M. L. Minsky, *Computation*, Prentice-Hall Englewood Cliffs, 1967.