Decidability of deadlock-freedom of RSC systems*

Nowar Kazem
ENS de Rennes
Inria
Rennes, France
nowar.kazem@ens-rennes.fr

FIFO automata are finite state machines that communicate via FIFO queues and are often used to model distributed protocols. However, the unbounded nature of these queues makes many verification problems undecidable. To enable model checking of such systems, researchers have focused on identifying decidable subclasses of FIFO systems. One such subclass, known as Realizable with Synchronous Communications (RSC) systems, was introduced by Germerie-Guizouarn, Di Giusto, and Lozes in order to generalize half-duplex systems studied by Cece and Finkel. They demonstrated that several verification properties for RSC systems are decidable. While Germerie-Guizouarn also investigated the property of deadlock-freedom and proposed a sufficient condition for deadlock-freedom in RSC systems, this condition does not establish the decidability of the property itself. In this work, we prove that deadlock-freedom is indeed decidable for RSC systems.

1 Introduction

FIFO automata—also known as asynchronous communicating automata—are finite-state machines that exchange messages via FIFO (First-In-First-Out) queues. They provide a powerful and expressive framework for modeling the Communications of distributed protocols. However, in their general form, FIFO automata are Turing-powerful, making many verification problems undecidable. To enable model checking, it becomes necessary to identify restricted subclasses with more manageable properties.

One such subclass, known as *Realizable with Synchronous Communications* (RSC) systems, was introduced by Germerie-Guizouarn, Di Giusto, and Lozes [2] as a generalization of binary half-duplex systems, originally proposed by Cece and Finkel [1]. In their work, Germerie-Guizouarn et al. demonstrated that (1) it is decidable whether a system is RSC, and (2) all regular properties are decidable for RSC systems.

A particularly important property is deadlock-freedom, which ensures that from any configuration, the system can reach a stable configuration—i.e., one in which all channels are empty. In [3], Germerie-Guizouarn also identified a sufficient condition under which an RSC system is deadlock-free. However, this condition is not necessary: he provided an example of an RSC system that is deadlock-free but does not satisfy the condition. As a result, the question of whether deadlock-freedom is decidable for RSC systems remained open.

In this work, we answer this question affirmatively. We prove that deadlock-freedom is decidable for RSC systems and provide an algorithm to decide this property.

Outline. The paper is organised as follows: Section 2 introduces communicating automata, systems, and relevant preliminaries. Section 3 defines RSC systems and presents related definitions and results. Section 4 formalizes the deadlock-freedom property and proves its decidability.

*The author thanks Loïc Germerie Guizouarn and Thierry Jéron for their helpful comments and suggestions.

2 Preliminaries[1]

Given a finite set S (of letter), we denote by S^* the set of finite word over S. Given two words w_1, w_2, $w_1 \cdot w_2$ denotes the concatenation of w_1 and w_2, ε denotes the empty word, and $|w_1|$ denotes the length of w_1. If $\mathscr{A}$ is an automaton, then $\mathscr{L}(\mathscr{A})$ denotes the language accepted by $\mathscr{A}$. For two sets S and I, we write $\mathbf{b}$ (in bold) for an element of S^I, and b_i for the i-th component of $\mathbf{b}$, so that $\mathbf{b} = (b_i)_{i\in I}$. For a set A, we denote by $\mathscr{P}(A)$ the set of subsets of A (also denoted 2^A).

A *FIFO automaton* is basically a finite state machine equipped with FIFO queues where transitions are labelled with either queuing or dequeuing actions. More precisely, considering a set of process $\mathbb{P}$:

Definition 1 (FIFO automaton)**.** *A* FIFO automaton *associated with the participant $p \in \mathbb{P}$ is a tuple $\mathscr{A} = (L_p, \mathbb{V}_p, I_p, \mathsf{Act}_p, \delta_p, l_{0,p})$ where*

- L_p *is a finite set of* control states*,*
- $\mathbb{V}_p$ *is a finite set of* messages*,*
- I_p *is a finite set of* buffer identifiers*,*
- $\mathsf{Act}_p \subseteq I_p \times \{!^p, ?^p\} \times \mathbb{V}_p$ *is a finite set of* actions*,*
- $\delta_p \subseteq L_p \times \mathsf{Act}_p \times L_p$ *is the* transition relation*, and*
- $l_{0,p} \in L_p$ *is the* initial control state*.*

The size $|\mathscr{A}_p|$ *of $\mathscr{A}_p$ is* $|L_p| + |\delta_p|$.

Actions $a = (i, \dagger, v)$, for $\dagger \in \{!^p, ?^p\}$ are also denoted by $i\dagger v$, p is omitted if there is no ambiguity. An action $i!^p v$ means that participant p sends mesage v on buffer i, while $i?^p v$ stands for the reception of v from i by participant p. Given a FIFO automaton $\mathscr{A} = (L_p, \mathbb{V}_p, I_p, \delta_p, l_{0,p})$, a *configuration* of $\mathscr{A}$ is a tuple $\gamma = (l, \mathbf{b}) \in L_p \times (\mathbb{V}_p^*)^{I_p}$. The initial configuration is $\gamma_0 = (l_{0,p}, \mathbf{b}^{\emptyset})$ where for all $i \in I$, $b_i^{\emptyset} = \varepsilon$. A *step* is a tuple (γ, a, γ') (often written $\gamma \xrightarrow[\mathscr{A}]{a} \gamma'$) where $\gamma = (l, \mathbf{b})$ and $\gamma' = (l', \mathbf{b}')$ are configurations and a is an action, such that the following holds:

- $(l, a, l') \in \delta_p$,
- if $a = i!^p v$, then $b'_i = b_i \cdot v$ and $b'_j = b_j$ for all $j \in I_p \setminus \{i\}$.
- if $a = i?^p v$, then $b_i = v \cdot b'_i$ and $b'_j = b_j$ for all $j \in I_p \setminus \{i\}$.

A FIFO system, later called simply a *system*, is a family $\mathfrak{S} = (\mathscr{A}_p)_{p\in\mathbb{P}}$ of FIFO automata. Since actions are labeled by its participant, all FIFO automata have disjoint sets of actions: for all $p \neq q \in \mathbb{P}$, $\mathsf{Act}_p \cap \mathsf{Act}_q = \emptyset$. Each $p \in \mathbb{P}$ is referred to as a *process*. We write $\mathsf{process}(a)$ to denote the unique p (when it exists) such that $a \in \mathsf{Act}_p$, i.e., such that there exist i, v and $\dagger \in \{!^p, ?^p\}$ such that $a = i\dagger v$.

Example 1 (FIFO Automata). Figure 1 shows a graphical representation of a FIFO system, borrowed from [2].This system represents a protocol between a client, a server and a database logging requests from the client and the server. In this protocol, a client can log something on the database or send requests to the server, when those requests are satisfied the server logs them in a database. Each automaton is equipped with a buffer in which it receives messages from all other participants: this system is an example of a mailbox system. To improve readability of the graphical representation, we refer to the buffers with the initial of the automaton to which they are associated. For example, s is the buffer into which the server can receive messages. This simple system will be used as a running example throughout the paper. □

[1] Some definitions and concepts in this section are taken from the work of Loïc Germerie Guizouarn et al. [2].

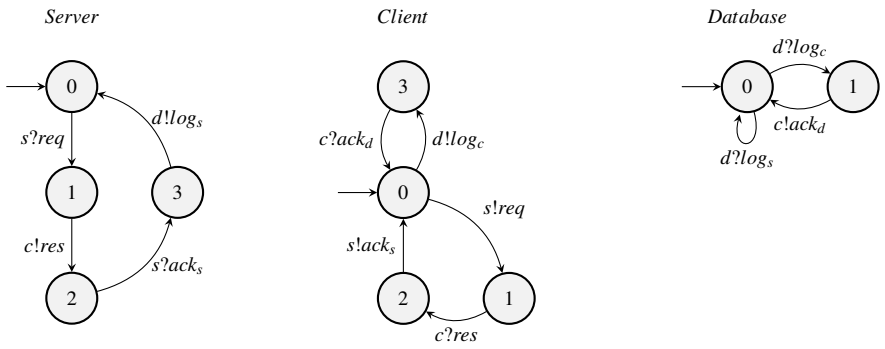


Figure 1: Client/Server/Database protocol

The size $|\mathfrak{S}|$ of $\mathfrak{S}$ is the sum of the $|\mathscr{A}_p|$. Note that $|\mathbb{P}|$ and $|I|$ are independent from the size of $\mathfrak{S}$. In particular, when we say that an algorithm is in polynomial time, we mean in time $\mathscr{O}(|\mathfrak{S}|^k)$ for some k that may depend on $|\mathbb{P}|$ and $|I|$.

The FIFO automaton $\mathsf{product}(\mathfrak{S})$ associated with $\mathfrak{S}$ is the standard asynchronous product automaton: $\mathsf{product}(\mathfrak{S}) = (\prod_{p\in\mathbb{P}} L_p, \bigcup_{p\in\mathbb{P}} \mathbb{V}_p, \bigcup_{p\in\mathbb{P}} I_p, \delta, \mathbf{l}_0)$ where $\mathbf{l}_0 = (l_{0,p})_{p\in\mathbb{P}}$ and δ is the set of triples $(\mathbf{l}, a, \mathbf{l}')$ for which there exists $p \in \mathbb{P}$ such that $(l_p, a, l'_p) \in \delta_p$ and $l_q = l'_q$ for all $q \in \mathbb{P} \setminus \{p\}$. We often identify $\mathfrak{S}$ and $\mathsf{product}(\mathfrak{S})$ and write for instance $\xrightarrow[\mathfrak{S}]{a}$ instead of $\xrightarrow[\mathsf{product}(\mathfrak{S})]{a}$. Similarly, we say that $\gamma = (\mathbf{l}, \mathbf{b})$ is a configuration of $\mathfrak{S}$ while we mean that γ is a configuration of $\mathsf{product}(\mathfrak{S})$. An execution $e = a_1 \cdots a_n \in \mathsf{Act}^*$ is a sequence of actions. As usual $\overset{e}{\Rightarrow}$ stands for $\xrightarrow{a_1} \cdots \xrightarrow{a_n}$. We write $\mathsf{executions}(\mathfrak{S})$ for $\{e \in \mathsf{Act}^* \mid \gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma \text{ for some } \gamma\}$.

Next, we introduce the definition of reachable configuration:

Definition 2 (Reachable configuration [2])**.** *Let $\mathfrak{S}$ be a system. A configuration γ is reachable if there exists $e \in \mathsf{Act}^*$ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$. The set of all reachable configurations of $\mathfrak{S}$ is denoted $RS(\mathfrak{S})$. For a configuration γ, we denote also $RS(\gamma)$ the set of all configurations reachable from γ in $\mathfrak{S}$, i.e., $\{\gamma' \mid \exists e, \gamma \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma'\}$*

Definition 3 (Buffer size)**.** *Let $\mathfrak{S}$ be a system and let $\gamma = (\mathbf{l}, \mathbf{b})$ be a configuration. The buffer size of γ is $\sum_{i\in I} |b_i|$. Given an execution e, the buffer size of e is the buffer size of the configuration γ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$.*

Definition 4 (Stable configuration [1])**.** *Let $\mathfrak{S}$ be a system and $\gamma = (\mathbf{l}, \mathbf{b})$ be one of its configuration. γ is a* stable *configuration if all its channels are empty, i.e., $\mathbf{b} = \mathbf{b}^{\emptyset}$.*

Definition 5 (Matching pair)**.** *Given an execution $e = a_1 \cdot \ldots \cdot a_n$, we say that $\{j, j'\} \subseteq \{1, \ldots, n\}^2$ is a* matching pair *if there exists a buffer identifier i, a message v and natural number k such that*

1. *$a_j = i!v$,*
2. *$a_{j'} = i?v$,*
3. *a_j is the k-th send action on i in e, and*
4. *$a_{j'}$ is the k-th receive action on i in e.*

We also said that two actions $a_j, a_{j'}$ form a matching pair if $\{j, j'\}$ is a matching pair.

Intuitively, a matching pair consists of a message send action and its corresponding receive action. An execution imposes a total order on the actions. Sometimes, however, it is useful to visualise only the causal dependencies between actions. Here we define *action graphs* that depict all causal dependencies. We say that two actions a_1, a_2 are independent if they do not form a matching pair, $\mathsf{process}(a_1) \neq \mathsf{process}(a_2)$ and it is not the case that a_1 and a_2 are two actions of the same type on a same buffer: there is no $\dagger \in \{!,?\}$, $i \in I$ and $v_1, v_2 \in \mathbb{V}$ such that $a_1 = i\dagger v_1$ and $a_2 = i\dagger v_2$. If two actions a_1 and a_2 are independent and adjacent, then they commute. For this reason we also say that two actions commute if they are independent.

Definition 6 (Communication)**.** *Given an execution $e = a_1 \cdot \ldots \cdot a_n$, a* communication *of e is either a matching pair $\{j, j'\}$, or a singleton $\{j\}$ such that j does not belong to any matching pair (such a communication is called unmatched). We also said that $a_j, a_{j'}$ form a communication of e if $\{j, j'\}$ is a communication of e, and a_j forms a communication of e if $\{j\}$ is an unmatched communication of e.*

Given a matched communication $i!v, i?v$ (of an execution e), we denote by $i!?v$ for $i!v \cdot i?v$.

Definition 7 (Action graph [2])**.** *Given an execution $e = a_1 \cdots a_n$, the* action graph $\mathsf{agraph}(e)$ *is the vertex-labeled directed graph $(\{1,\ldots,n\}, \prec_e, \lambda_e)$ where $\lambda_e(j) = a_j$ and $j \prec_e j'$ if $j < j'$ and $a_j, a_{j'}$ do not commute.*

When $j \prec_e j'$, we say that a_j *happens before* $a_{j'}$. Note that for an execution e with $e \in \mathsf{executions}(\mathfrak{S})$, $\prec_e^*$ is a partial order. Two executions e, e' are causally equivalent, denoted by $e \overset{\mathfrak{S}}{\sim} e'$, if their action graphs are isomorphic. Said differently, e, e' are causally equivalent if they are two linearisations of a same *happens before* partial order. Note that if $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Longrightarrow}} \gamma$ and $e \overset{\mathfrak{S}}{\sim} e'$, then $\gamma_0 \overset{e'}{\underset{\mathfrak{S}}{\Longrightarrow}} \gamma$.

Definition 8 (Causal equivalence [2])**.** *Let e, e' be two executions, with $e = a_1 \cdot \ldots \cdot a_n$ and $e' = a'_1 \cdot \ldots \cdot a'_n$ and $\mathsf{agraph}(e)$ respectively $\mathsf{agraph}(e')$, their action graphs. Executions e and e' are causally equivalent (denoted $e \sim e'$) if there is a permutation σ of $\{1,\ldots,n\}$ such that*

1. *for all $i \in \{1,\ldots,n\}$, $a'_{\sigma(i)} = a_i$, and*
2. *for all $j, j' \in \{1,\ldots,n\}$, $j \prec_e j'$ if and only if $\sigma(j) \prec_{e'} \sigma(j')$.*

Informally, two executions are causally equivalent if one can be obtained from the other by swapping independent actions (by commuting actions that commute). Now we define *partial executions*.

Definition 9 (Partial execution [3])**.** *Let $e = a_1 \cdots a_n$ and $e' = a'_1 \cdots a'_m$ be two executions. We say that e is a* partial executions *of e', and we donote $e \trianglelefteq e'$, if there exists a function $\sigma : \{1,\ldots,n\} \to \{1,\ldots,m\}, n \leq m$ such that for all $i \in \{1,\ldots,n\}$:*

1. $a_i = a'_{\sigma(i)}$;
2. $\forall j \in \{1,\ldots,n\}$,
   - $i < j \iff \sigma(i) < \sigma(j)$, *and*
   - $i = j \iff \sigma(i) = \sigma(j)$;
3. $\forall k \in \{1,\ldots,m\}, a'_k \prec_{e'} a'_{\sigma(i)} \implies \exists j, \sigma(j) = k$

We can relate this definition to the action graph: we write $e \trianglelefteq e'$ if $\mathsf{agraph}(e)$ is a subgraph of $\mathsf{agraph}(e')$ preserving the order of nodes (the nodes are placed hirozontally, meaning the graph is drawn as a linear sequence). By subgraph, we mean that $\mathsf{agraph}(e)$ is obtained by removing some nodes from $\mathsf{agraph}(e')$, along with all nodes that depend on the removed ones. Said differently, if $a \prec a'$ and $a \in e$ then $a' \in e$. We denote $\mathsf{partial}(e) = \{e' \mid e' \trianglelefteq e\}$ for an execution e, and $\mathsf{partials}(\mathscr{L}) = \{e \mid \exists e' \in \mathscr{L}, e \trianglelefteq e'\}$ for a set of executions $\mathscr{L}$.

Example 2. Figure 4 illustrates an example of an action graph along with a partial execution. Note that when an action is removed to obtain a partial execution, all actions that depend on it are also removed. □

3 RSC systems

In this section we introduce realisable with synchronous communications (RSC) systems. Those systems aim at mimicking rendez-vous or synchronous communications by checking whether each execution can be rescheduled to an equivalent one where all receptions immediately follow their corresponding send.

Definition 10 (RSC system, [2])**.** *An execution e is* RSC *if all matching pairs are of the form $\{j, j+1\}$. A system $\mathfrak{S}$ is* RSC *if for all execution $e \in \mathsf{executions}(\mathfrak{S})$, there exists an RSC execution e' such that $e \overset{\mathfrak{S}}{\sim} e'$.*

Given a system $\mathfrak{S}$, we note $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) := \{e \in \mathsf{executions}(\mathfrak{S}) \mid e \text{ is RSC}\}$ the set of all RSC executions of $\mathfrak{S}$.

Example 3. The execution $s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot d!log_c \cdot d?log_c$ is RSC, but the execution $s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot d!log_c \cdot s?ack_s$ is not RSC, although causally equivalent to a RSC execution. □

Example 4 (RSC system). The system in Figure 1 is RSC. Indeed, it is always possible to reorder the actions so that each reception immediately follows its corresponding send action, while preserving causal dependencies, by commuting actions that commute. □

Example 5. Consider the system Figure 2, the execution $e = 0!^p v_1 \cdot 1!^q v_2 \cdot 1?^p v_2 \cdot 0?^q v_1$ is not causally equivalent to an RSC execution, therefore the whole system is not RSC. □

Figure 2: A system that is not RSC

One may wonder why the RSC subclass is of particular interest. In fact, beyond its appealing properties—such as the decidability of many verification problems—and its simplicity despite being a relatively broad subclass, checking whether a system is RSC can be done in polynomial time, as shown by the following theorem.

Theorem 11. *Whether a system $\mathfrak{S} = (\mathscr{A}_p)_{p\in\mathbb{P}}$ of size n is RSC is decidable in time $\mathscr{O}(|I|^5|\mathbb{V}|^4 2^{|I|} n^{|\mathbb{P}|+2})$.*

Proof. See [2] section 3. □

In the remainder of this section, we define several automata that will be used throughout the rest of the paper.

Definition 12 (automaton $\mathscr{A}_{\mathsf{synch}}$, [3])**.** *Let $\mathfrak{S} = (\mathscr{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$, $\mathscr{A}_{\mathsf{synch}}(\mathfrak{S}) = (Q_{\mathsf{synch}}, \delta_{\mathsf{synch}}, q^0_{\mathsf{synch}}, F_{\mathsf{synch}})$ is a finite state automaton such that :*

- $Q_{\mathsf{synch}} = L_{\mathfrak{S}}$,
- $q^0_{\mathsf{synch}} = \mathbf{l}_0$,
- $F_{\mathsf{synch}} = Q_{\mathsf{synch}}$, *and*
- $(\mathbf{l}, c, \mathbf{l}') \in \delta_{\mathsf{synch}}$ *if* $(\mathbf{l}, \mathbf{b}^{\emptyset}) \overset{c}{\underset{\mathfrak{S}}{\Longrightarrow}} (\mathbf{l}', \mathbf{b}^{\emptyset})$[2]

[2]Note that c is not a single action; since the buffers remain empty after its execution, c consists of a matching pair-i.e., it is of the form $i!a \cdot i?a$ (also noted $i!?a$).

We denote $\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}) := \mathscr{L}(\mathscr{A}_{\mathsf{synch}})$. *Intuitively,* $\mathscr{L}(\mathscr{A}_{\mathsf{synch}})$ *contains all executions that match all the send actions and the receptions (that we call a* synchronous *execution), in other word, a synchronous execution is an RSC execution such that all messages are received.*

Definition 13 (automaton $\mathscr{A}_{\mathsf{part}}$, [3])**.** *Let* $\mathfrak{S}$ *be an RSC system and* $\mathscr{A}_{\mathsf{synch}}$ *be the finite state automaton recognising synchronous executions of* $\mathfrak{S}$*;* $\mathscr{A}_{\mathsf{part}} = (\mathsf{Q}_{part}, \delta_{part}, \mathsf{q}^0_{part}, \mathsf{F}_{part})$ *is the non-deterministic finite state automaton where* $\mathsf{Q}_{part} = \mathsf{Q}_{synch} \times \mathscr{P}(\mathbb{P}) \times \mathscr{P}(\mathbb{I})$, $\mathsf{q}^0_{part} = (\mathsf{q}^0_{synch}, \varnothing, \varnothing)$, $\mathsf{F}_{part} = \mathsf{Q}_{part}$, *and*

1. $((\mathsf{q}, \mathfrak{p}, \mathfrak{f}), i!^p?^q v, (\mathsf{q}', \mathfrak{p}, \mathfrak{f})) \in \delta_{part}$ *if :*
   - $(\mathsf{q}, i!^p?^q v, \mathsf{q}') \in \delta_{synch}$, *and*
   - $\{p, q\} \cap \mathfrak{p} = \varnothing$ *and*
   - $\{i^r, i^s\} \cap \mathfrak{f} = \varnothing$*;*
2. $((\mathsf{q}, \mathfrak{p}, \mathfrak{f}), i!^p v, (\mathsf{q}', \mathfrak{p}', \mathfrak{f}')) \in \delta_{part}$ *if there exists* $q \in \mathbb{P}$ *such that :*
   - $(\mathsf{q}, i!^p?^q v, \mathsf{q}') \in \delta_{synch}$, *and*
   - $\mathfrak{p}' = \mathfrak{p} \cup \{q\}$ *and*
   - $i^s \notin \mathfrak{f}$*;*
3. $((\mathsf{q}, \mathfrak{p}, \mathfrak{f}), \varepsilon, (\mathsf{q}', \mathfrak{p}', \mathfrak{f}')) \in \delta_{part}$ *if there exist* $(p, q) \in \mathbb{P}^2, i \in \mathbb{I}$, *and* $v \in \mathbb{V}$ *such that :*
   - $(\mathsf{q}, i!^p?^q v, \mathsf{q}') \in \delta_{synch}$, *and*
   - $\mathfrak{p}' = \mathfrak{p} \cup \{p, q\}$ *and*
   - $\mathfrak{f}' = \mathfrak{f} \cup \{i^s\}$.

In a state $(\mathsf{q}, \mathfrak{p}, \mathfrak{f})$, q represents the control state of $\mathscr{A}_{synch}$, $\mathfrak{p}$ represents the set of blocked processes (processes that are not allowed to act), and $\mathfrak{f}$ represents the set of blocked FIFO buffers. The notation $i^s \in \mathfrak{f}$ means that i is blocked for sending actions, and so for all actions, and $i^r \in \mathfrak{f}$ that i is blocked for receptions.

Lemma 14 (See [3])**.** *Let* $\mathfrak{S}$ *be an RSC system, and* $\mathscr{A}_{part}$ *be the finite state automaton defined above. One has* $\mathscr{L}(\mathscr{A}_{part}) = \mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}))$ [3]

Proof. See [3], p.80, lemma 6.2.1. □

Note that partial executions of synchronous executions are RSC. In fact, let e be a partial execution of a synchronous execution e'. There exists a permutation σ that satifies the condition in Definition 9. Let $\{j, j'\}$ be a matching pair in e with $j < j'$. One has $\sigma(j) < \sigma(j')$ and $\{\sigma(j), \sigma(j')\}$ is a matching pair in e'. So $\sigma(j') = \sigma(j) + 1$. We proceed by contradiction : let us assume that $j' \neq j + 1$. Then there exists $j < i < j'$. So $\sigma(j) < \sigma(i) < \sigma(j')$ contradiction because $\sigma(j) = \sigma(j')$. Hence $j' = j + 1$ and e is RSC.

Definition 15 (automaton $\mathscr{A}_{\mathsf{rsc}}$)**.** *Let* $\mathfrak{S}$ *be an RSC system and* $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}, I, \mathsf{Act}, \delta_{\mathfrak{S}}, \mathbf{l}_0)$*;* $\mathscr{A}_{\mathsf{rsc}} = (L_{rsc}, \delta_{rsc}, l_{rsc,0}, F_{rsc})$ *is the non-deterministic finite state automaton where* $L_{rsc} = L_{\mathfrak{S}} \times \mathscr{P}(I)$, $l_{rsc,0} = (\mathbf{l}_0, \varnothing)$, $F_{rsc} = L_{rsc}$, *and*

1. $((\mathbf{l}, \mathfrak{f}), i!?v, (\mathbf{l}', \mathfrak{f}')) \in \delta_{rsc}$ *if :*
   - $i \notin \mathfrak{f}$, *and*
   - $\mathfrak{f} = \mathfrak{f}'$ *and*

[3] The two alphabets are not identical, but if we treat $i!?a$ (a single letter) as equivalent to the sequence $i!a \cdot i?a$ (a word of two letters), then the alphabets can be unified. In practice, when discussing executions, the specific representation is not important.

- *there exists* $\mathbf{l}'' \in L_{\mathfrak{S}}$ *such that* $(\mathbf{l}, i!v, \mathbf{l}''), (\mathbf{l}'', i?a, \mathbf{l}') \in \delta_{\mathfrak{S}}$

2. $((\mathbf{l}, \mathfrak{f}), i!v, (\mathbf{l}', \mathfrak{f}')) \in \delta_{rsc}$ *if :*
   - $\mathfrak{f}' = \mathfrak{f} \cup \{i\}$ *and*
   - $(\mathbf{l}, i!v, \mathbf{l}') \in \delta_{\mathfrak{S}}$

In a state $(\mathbf{l}, \mathfrak{f})$, $\mathbf{l}$ represents the control state of $\mathsf{product}(\mathfrak{S})$ and $\mathfrak{f}$ represents the set of blocked FIFO buffers (from which we cannot receive any message).

Lemma 16 (see [2])**.** *Let* $\mathfrak{S}$ *be a system. One has* $\mathscr{L}(\mathscr{A}_{\mathsf{rsc}}) = \mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S})$.

Proof. See [2]. □

4 Deadlock-freedom property

In this section we define what is a deadlock execution then we define what is a deadlock-free system, and finally we show that, knowing whether an RSC system is deadlock-free is decidable.

Definition 17 (Deadlock, [3])**.** *Let* $\mathfrak{S}$ *be a system, a configuration* γ *is a* deadlock *if for all* $\gamma' \in RS(\gamma)$*,* γ' *is not stable. An execution* e *is* deadlock *if the configuration* γ *such that* $\gamma_0 \xRightarrow[\mathfrak{S}]{e} \gamma$ *is deadlock.*

Definition 18 (Deadlock-Freedom, [3])**.** *A system* $\mathfrak{S}$ *is* deadlock-free *if* $RS(\mathfrak{S})$ *contains no deadlock configuration, i.e., for all reachable configuration* $\gamma \in RS(\mathfrak{S})$ *there exists* $\gamma \in RS(\gamma)$ *such that* γ_s *is a stable configuration.*

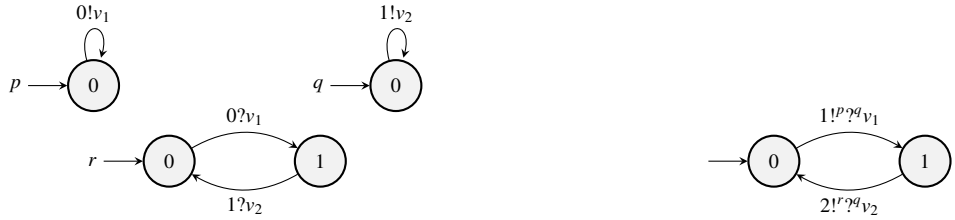
Notice that, if e is not deadlock, then there exists e' such that $\gamma_0 \xRightarrow[\mathfrak{S}]{e \cdot e'} \gamma$ where γ is a stable configuration. Since we work with RSC systems, $e \cdot e'$ is causally equivalent to an RSC execution e_s. However, all sent messages have been received, then $e_s \in \mathsf{executions}_{synch}(\mathfrak{S})$.

In [3], Germerie-Guizouarn establishes the folowing lemma :

Lemma 19 (Sufficient condition for deadlock-freedom)**.** *Let* $\mathfrak{S}$ *be an RSC system. If* $\mathsf{executions}_{rsc}(\mathfrak{S}) = \mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}\mathfrak{S})$*, then* $\mathfrak{S}$ *is deadlock-free.*

Proof. See [3], chapter 6, lemma 6.3.2. □

Germerie-Guizouarn also showed that this condition is not necessary for an RSC system to be deadlock-free. Figure 3 illustrates a system that is RSC and deadlock-free but does not satisfy the condition.



System $\mathfrak{S}_{rsc}$ $\mathscr{A}_{\mathsf{synch}}$

Figure 3: A deadlock-free RSC system which does not satisfy the condition borrowed from [3]

The main issue is that $\mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}))$ is not closed under RSC equivalence; that is, if $e \in \mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}))$ and $e \sim e'$ with e' an RSC execution, it is not necessarily the case that $e' \in \mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}))$. One can try to compute the closure under RSC equivalence of $\mathsf{partials}(\mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}))$. However, this solution is not straightforward, and it is unclear whether it is even computable. Regardless of its complexity, to prove decidability, we take a slightly simpler approach.

Lemma 20. *Let $\mathfrak{S}$ be an RSC system, and e an execution of $\mathfrak{S}$. One has :*

$$\exists e' : e.e' \sim e_{synch} \in \mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}) \iff \exists e_s : e \sim e_s \in \mathscr{L}(\mathscr{A}_{\mathsf{part}})$$

i.e., e is not deadlock, if and only if, e is causally equivalent to a partial execution of a synchronous execution.

Proof. The left implication is trivial, the right implication is a bit harder. The idea is that if there exists e' such that $e.e' \sim e_{\mathsf{synch}}$ then the execution " $e_{\mathsf{synch}} - e'$ " (the action resulted by removing the actions of e' in e_{synch}) satisfies our conditions; this is legitimate, because since e' can be executed after e, the actions of e happen before those of e', i.e., there is no a an action of e and a' an action of e' such that $a' \prec_{e_{synch}} a$, and then removing the actions of e' do not cause the removal of any action of e. Below is a diagram that illustrates the construction of e_s (where the red actions are the actions of e, the black ones of e', $m := |e|$ and $n := |e_{\mathsf{synch}}|$).

$$\begin{array}{cccccccccccc} e_{\mathsf{synch}} & = & a_1 & \dots & a_{i_1} & \dots & a_j & \dots & a_{i_m} & \dots & a_n \\ & & & & \downarrow & & & & \downarrow & & \\ e_s & = & & & a_{i_1} & \dots & & \dots & a_{i_m} & & \end{array}$$

See A.1 for a formal proof. □

Lemma 21. *Let $\mathfrak{S}$ be an RSC system and $e \in \mathscr{L}(\mathscr{A}_{\mathsf{part}})$. There exist a word $e' \in (\Sigma_{!?} \cup \Sigma_{?})^*$ (i.e. e' is made up of send-receive actions and/or unmatched receive actions)[4] such that $e \cdot e'$ is causally equivalent to a synchronous execution.*

Proof. See A.2. □

Lemma 22. *Let $\mathfrak{S}$ be an RSC system and let $e = u \cdot v \cdot w$ be an RSC execution, where v is a loop in $\mathscr{A}_{\mathsf{rsc}}$ that introduces messages which are not received. If the execution $e' := u \cdot w$ does not lead to a deadlock, then the buffer size of e can be reduced. That is, there exists an execution e'' such that the buffer size of $e \cdot e''$ is strictly smaller than that of e.*

Proof. See A.3 □

Lemma 23. *Let $\mathfrak{S}$ be an RSC system. We note $M := |\mathscr{A}_{\mathsf{rsc}}|$. $\mathfrak{S}$ is deadlock-free if and only if all executions whose length is less than M are not deadlock.*

Proof. See A.4. □

Theorem 24 (Decidablity of deadlock-freedom of RSC system)**.** *Knowing whether an RSC system is deadlock-free is decidable.*

Proof. See A.5. □

[4] $\Sigma_{!?}$ denotes the alphabet of send-receive actions, i.e.,, actions of the form $i!?a$, and $\Sigma_{?}$ denotes the alphabet of receive actions, i.e.,, actions of the form $i?a$.

References

[1] Gérard Cécé & Alain Finkel (2005): *Verification of programs with half-duplex communication*. *Information and Computation* 202(2), pp. 166–190, doi:https://doi.org/10.1016/j.ic.2005.05.006. Available at `https://www.sciencedirect.com/science/article/pii/S0890540105001082`.

[2] Cinzia Di Giusto, Loïc Germerie Guizouarn & Etienne Lozes (2021): *Towards Generalised Half-Duplex Systems*. *Electronic Proceedings in Theoretical Computer Science* 347, p. 22–37, doi:10.4204/eptcs.347.2. Available at `http://dx.doi.org/10.4204/EPTCS.347.2`.

[3] Loïc Germerie Guizouarn (2023): *Communicating automata and quasi-synchronous communications*. Theses, Université Côte d'Azur. Available at `https://theses.hal.science/tel-04524379`.

A Proofs

A.1 Proof of lemma 20

Lemma 20. *Let $\mathfrak{S}$ be an RSC system, and e an execution of $\mathfrak{S}$. One has :*

$$\exists e' : e.e' \sim e_{synch} \in \mathsf{executions}_{\mathsf{synch}}(\mathfrak{S}) \iff \exists e_s : e \sim e_s \in \mathscr{L}(\mathscr{A}_{\mathsf{part}})$$

i.e., e is not deadlock, if and only if, e is causally equivalent to a partial execution of a synchronous execution.

Proof. $\boxed{\Leftarrow}$ This implication is easy, suppose that there exists an exection $e_s \in \mathscr{L}(\mathscr{A}_{part}) \cap \mathscr{L}(\mathscr{A}_{synch})$ such that $e \sim e_s$. Since $e_s \in \mathscr{L}(\mathscr{A}_{part})$, there exists $e_{synch} \in \text{executions}_{synch}(\mathfrak{S})$ such that $e_s \trianglelefteq e_{synch}$. By lemma 6.3.1 [3], there exists e' such that $e_s.e' \sim e_{synch}$, hence $e.e' \sim e_s.e' \sim e_{synch}$.

$\boxed{\Rightarrow}$ Suppose that there exists e' such that $e.e' \sim e_{synch} \in \text{partials}(\text{executions}_{synch}(\mathfrak{S}))$. Let us denote $e'' := e.e' = a''_1 \dots a''_n$, $m := |e|$ and $e_{synch} = a_1^{synch}, \dots, a_n^{synch}$. There is then a permutation σ of $[|1,n|]$ such that :

- $\forall i \in [|1,n|]$, $a^{synch}_{\sigma(i)} = a''_i$, in particular $\forall i \in [|1,m|]$, $a^{synch}_{\sigma(i)} = a_i$
- $\forall j, j' \in [|1,n|]$, $j \prec_{e''} j' \iff \sigma(j) \prec_{e_{synch}} \sigma(j')$

The diagram below illustates the situation :

$$\begin{array}{rcccccccccccc}
e_{\mathsf{synch}} & = & a_1^{synch} & \cdot \dots \cdot & a_i^{synch} & \cdot \dots \cdot & a_j^{synch} & \cdot \dots \cdot & a_k^{synch} & \cdot \dots \cdot & a_n^{synch} \\
 & = & a''_j & \cdot \dots \cdot & a_{i_1} & \cdot \dots \cdot & a''_{j'} & \cdot \dots \cdot & a_{i_m} & \cdot \dots \cdot & a''_k \\
 & & & & \downarrow & & & & \downarrow & & \\
e_s & = & & & a_{i_1} & \cdot \dots \cdot & \dots & \cdot \dots \cdot & a_{i_m} & &
\end{array}$$

Let $\tau : [|1,m|] \to [|1,m|]$ be the function that maps each integer j to the index (in e) of the j-th action of e that appears in e_{synch}. In other words, $\tau(j)$ gives the position in e of the j-th action that appears in e_{synch}. We can define τ differently. Let us order the sequence $(\sigma(i))_{i \in [|1,m|]}$, one has $\sigma(\tau(1)) < \dots < \sigma(\tau(m))$ (in the diagram above, the permutation τ is the function such that $\tau(j) = i_j$ for $j \in \{1, \dots, m\}$). We denote then $e_s := a^{synch}_{\sigma\circ\tau(1)} \dots a^{synch}_{\sigma\circ\tau(m)} = a^s_1, \dots, a^s_m$. Let us first check that $e \sim e_s$. One has :

1. for all $i \in \{1, \dots, n\}$, $a^s_i = a^{synch}_{\sigma\circ\tau(i)} = a_{\tau(i)}$ because $\tau(i) \in \{1, \dots, m\}$

2. for all $j, j' \in \{1,\dots,n\}$,

$$\begin{aligned}\tau(j) \prec_e \tau(j') &\iff \sigma\circ\tau(j) \prec_{e_{synch}} \sigma\circ\tau(j') \\ &\iff j \prec_{e_s} j'\end{aligned}$$

This shows by the way that e_s is indeed an execution, because it is causally equivalent to an execution. Let us show now that $e_s \trianglelefteq e_{synch}$. Let $i \in [|1,m|]$, one has :

1. $a_i^s = a^{synch}_{\sigma\circ\tau(i)}$
2. for all $j \in [|1,m|]$ one has :
   - $i < j \iff \sigma\circ\tau(i) < \sigma\circ\tau(j)$ by construction of $\sigma\circ\tau$
   - $i = j \iff \sigma\circ\tau(i) = \sigma\circ\tau(j)$ by injectivity
3. let $k \in [|1,n|]$ such that $a_k^{synch} \prec_{e_{synch}} a^{synch}_{\sigma\circ\tau(i)}$. Since $e'' \sim e_{synch}$, $a''_{\sigma^{-1}(k)} \prec_{e''} a''_{\tau(i)}$ so $\sigma^{-1}(k) < \tau(i)$ and $a''_{\sigma^{-1}(k)}$, $a''_{\tau(i)}$ do not commute. One has $\tau(i) \in [|1,m|]$, then $\sigma^{-1}(k) \in [|1,m|]$. So, there exists $j \in [|1,m|]$ such that $a_j = a''_{\sigma^{-1}(k)}$, hence $a^{synch}_{\sigma(j)} = a_k^{synch}$. By bijectivity of τ, there exists $j' \in [|1,m|]$ such that $j = \tau(j')$ hence $a^{synch}_{\sigma\circ\tau(j)} = a_k^{synch}$ so $\sigma\circ\tau(j) = k$.

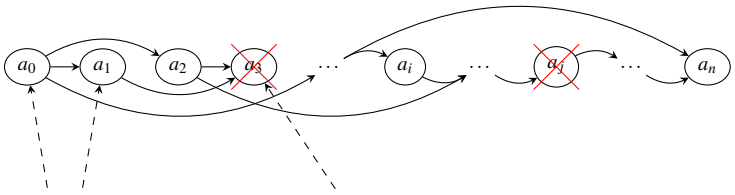
Hence the desired result. See B.2 for more explanation. □

A.2 Proof of lemma 21

Lemma 21. *Let $\mathfrak{S}$ be an RSC system and $e \in \mathscr{L}(\mathscr{A}_{\mathsf{part}})$. There exist a word $e' \in (\Sigma_{!?} \cup \Sigma_?)^*$ (i.e. e' is made up of send-receive actions and/or unmatched receive actions)[5] such that $e\cdot e'$ is causally equivalent to a synchronous execution.*

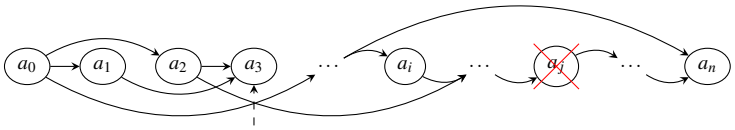
Proof. This result is quite intuitive. Before embarking on a formal proof, let us explain the idea behind this lemma. Speaking in terms of partial execution, if a reception has been removed (in a synchronous execution in order to obtain a partial execution), then all actions that depend on the removed one have also been removed. Therefore, the removed reception is independent of all subsequent executions. Therefore, the reception is executable. Likewise, if a send has been removed, its reception has also been removed, and therefore this communication (send-receive) commutes with everything that follows it, and is therefore executable. In automaton terms, if a receive has been removed, then its channel and its participant have been blocked, thus remaining in the same state (in terms of both the control state and the state of the channel concerned) for the entire remainder of the execution. And so we are able to execute this reception at the end of the execution. The same applies to a send-receive. In terms of action graphs, here is an explanation for a deleted reception, this explanation is the same for a send-receive (where the following graph is the graph of a synchronous execution $e = a_1 \cdot \ldots \cdot a_n$ and the red cross is the delete action (thus the resulting subgraph, removing the deleted nodes as well as the nodes that have a re-entrant edge coming from a deleted node, is the action graph of the partial execution in question):

[5] $\Sigma_{!?}$ denotes the alphabet of send-receive actions, i.e.,, actions of the form $i!?a$, and $\Sigma_?$ denotes the alphabet of receive actions, i.e.,, actions of the form $i?a$.



a_0 is a send action and a_1 is the reception because e is synchronous.

This deleted action (reception) (assuming it is the first action deleted) can be executed here, so by keeping this execution and removing those initially deleted (which is legitimate, because no action on which a_3 depends has been deleted since it is the first deletion), we obtain an execution (denoted e') whose action graph is shown below.



This action (reception) is independent of all the actions that follow it (because we removed the initially deleted actions, except for this one, and therefore the actions that depend on this action and that follow it are deleted). Thus this action commutes with all the actions that follow it, and therefore it can be placed at the end. The word obtained (by placing this action at the end) is causally equivalent to the execution e' and is therefore an execution. Thus, reception is executable at the end. By reasoning by induction, we show the desired result

Let us return to the formal proof. We proceed by strong induction on the number of actions deleted in e_s (a synchronous execution from which results e, the partial execution, by deleting actions). For $n \in N$, we denote by $\mathscr{P}(n)$: «for all $e_s \in \mathscr{A}_{\mathsf{synch}}$ and $e \in \mathscr{A}_{\mathsf{part}}$ such that $e \trianglelefteq e_s$ and $|e_s| - |e| = n$ (i.e., n actions of e_s have been deleted to obtain e), there exists a word $e' \in (\Sigma_{!?} \cup \Sigma_?)^*$ such that $e \cdot e'$ is causally equivalent to e_s ».

Initialisation : For $n = 0$, no action has been deleted, So $e = e_s$ is synchronous and $e \cdot \varepsilon \sim e_s$, the searched word e' is empty.

Heredity : Let $n \in \mathbb{N}$ such that $\forall k \leq n$, $\mathscr{P}(k)$. Let us show $\mathscr{P}(n+1)$. Let $e_s \in \mathscr{A}_{\mathsf{synch}}$ and $e \in \mathscr{A}_{\mathsf{part}}$ such that $e \trianglelefteq e_s$ and $|e_s| - |e| = n+1$. We write $e_s = a_{s_1} \cdot a_{s_2} \cdot \ldots \cdot a_{s_m}$ and $e = a_1 \cdot \ldots \cdot a_\ell$. Let i be the index of the first action deleted in e_s (i then satisfies $a_1 \cdot \ldots \cdot a_{i-1} = a_{s_1} \cdot \ldots \cdot a_{s_{i-1}}$). We distinguish two cases :

- first case : let us suppose that a_{s_i} is a send. Since e_s is synchronous, $a_{s_{i+1}}$ is the corresponding reception. The pair $\{i, i+1\}$ is a matching one in e_s, so $a_{s_{i+1}}$ depend on a_{s_i}. Thus $a_{s_{i+1}}$ has also been removed. Let e_h be the execution which results by keeping a_{s_i} et $a_{s_{i+1}}$ in e_s and removing all the initially deleted actions (in order to obtain e). The non deleted actions (in e_s) that follow this pair do not depend on it. Thus, $e_h \sim e \cdot a_{s_i} \cdot a_{s_{i+1}}$. Since $|e_s| - |e_h| = n-1$, By applying $\mathscr{P}(n-1)$, there exists $e'_h \in (\Sigma_{!?} \cup \Sigma_?)^*$ such that $e_h \cdot e'_h \sim e_s$. Hence $e_h \cdot e'_h \sim e_h \cdot a_{s_i} \cdot a_{s_{i+1}} \cdot e'_h \sim e_s$. Thus $e' := a_{s_i} \cdot a_{s_{i+1}} \cdot e'_h$ suits.
- second case : let us suppose that a_{s_i} is a reception. Likewise, we show the result.

Hence the heredity, hence $\forall n \in \mathbb{N}$, $\mathscr{P}(n)$. So that, if $e \in \mathscr{A}_{\mathsf{part}}$, there exists $e_s \in \mathscr{A}_{\mathsf{synch}}$ such that $e \trianglelefteq e_s$ and so by applying $\mathscr{P}$ we conclude. □

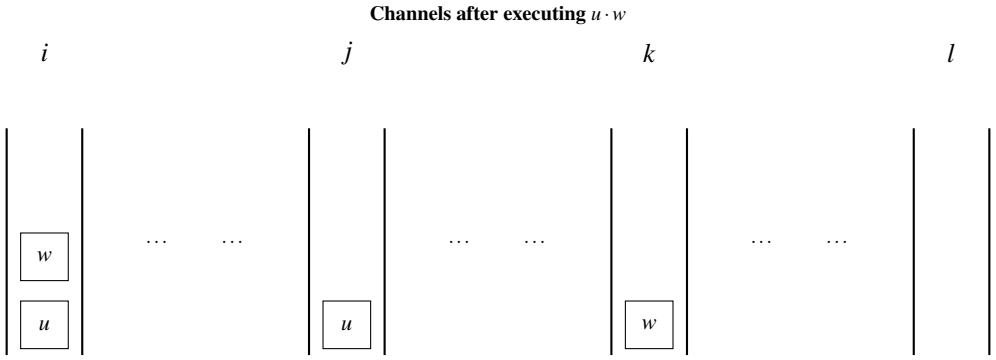
A.3 Proof of lemma 22

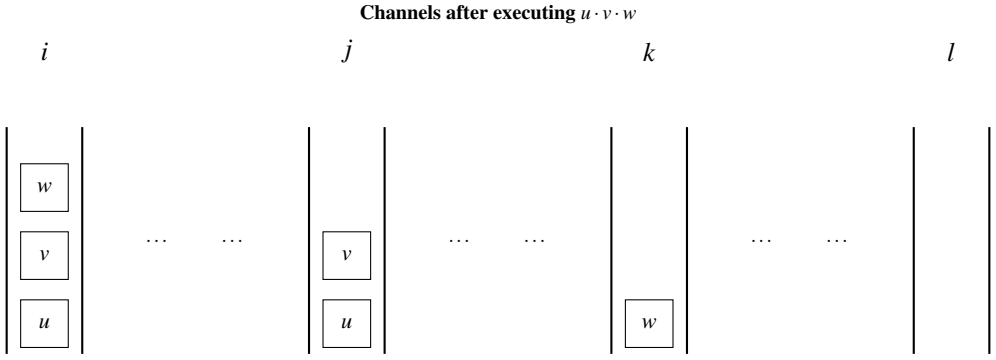
Lemma 22. *Let $\mathfrak{S}$ be an RSC system and let $e = u \cdot v \cdot w$ be an RSC execution, where v is a loop in $\mathscr{A}_{\mathsf{rsc}}$ that introduces messages which are not received. If the execution $e' := u \cdot w$ does not lead to a deadlock, then the buffer size of e can be reduced. That is, there exists an execution e'' such that the buffer size of $e \cdot e''$ is strictly smaller than that of e.*

Proof. First of all, since all the states of $\mathscr{A}_{\mathsf{rsc}}$ are accepting, e' is an RSC execution of $\mathfrak{S}$. Since e' is not deadlock, according to lemma 20, there exists $e_p \in \mathscr{L}(\mathscr{A}_{\mathsf{part}})$ such that $e' \sim e_p$. According to lemma 21, there exists $e_c \in (\Sigma_{!?} \cup \Sigma_?)^*$ such that $e_p \cdot e_c \sim e_s \in \mathscr{A}_{\mathsf{synch}}$. Since v is a loop in $\mathscr{A}_{\mathsf{rsc}}$, after executing v (after u) we return to the same state in $\mathscr{A}_{\mathsf{rsc}}$, i.e., u and $u \cdot v$ lead to the same state in $\mathscr{A}_{\mathsf{rsc}}$. So, v does not cause any channel blockage. However, v contains unreceived messages, so, u has already blocked the channels used by the unreceived messages of v. So $u \cdot w$ contains unreceived messages, and then e_c contains receive only actions. Let a_r be the first receive action in e_c. So, we have :

$$e_c = \underbrace{i!?v_1 \cdot j!?v_2 \cdot k!?v_3 \cdot \dots}_{\text{These actions use empty channels, because they immediatly receive the sent messages. Since } v \text{ does not use empty channels for its unreceived messages, these actions are executable afer } u \cdot v \cdot w} \cdot \overbrace{a_r}^{\text{This receiption is associated to an unreceived message in } u, \text{ it is still executable if we add } v \text{ because we still in the same state and } v \text{ does not change anything on the unreceived messages of } u \text{ (it just adds some messages in the already blocked channels).} } \cdot \dots$$

So the sequence of actions $e'' := i!?V_1 \cdot j!?v_2 \cdot \ldots \cdot a_r$ is executable after $u \cdot v \cdot w$, then the size of channels of $e \cdot e''$ is strictly smaller than that of e. The picture below shows the state of the channels after executing $u \cdot w$ and $u \cdot v \cdot w$:





□

A.4 Proof of lemma 23

Lemma 23. *Let $\mathfrak{S}$ be an RSC system. We note $M := |\mathscr{A}_{\mathsf{rsc}}|$. $\mathfrak{S}$ is deadlock-free if and only if all executions whose length is less than M are not deadlock.*

Proof. $\boxed{\Rightarrow}$ The right implication is trivial, if $\mathfrak{S}$ is deadlock-free then all the executions of $\mathfrak{S}$ are not deadlock, in particular those of size less than M (whatever M is) are not.

$\boxed{\Leftarrow}$ We proceed by contraposition. Let us show that if $\mathfrak{S}$ is not deadlock-free then there exists a deadlock execution of size less than M. Suppose that $\mathfrak{S}$ is not deadlock-free. Then there exists a deadlock execution, let us take one of minimal size of channels that is minimal in size (of minimal length among all deadlock executions whose buffer size is minimal (see B.1 for more details)), that we denote e_m. If we consider an equivalent execution of e_m, we can suppose that e_m is RSC (because $\mathfrak{S}$ is RSC). Let us show that $|e_m| \leq M$. We proceed by contradiction : assume that $|e_m| > M$. Since $|e_m| > M$, we can write $e_m = u \cdot v \cdot w$ with v a loop in $\mathscr{A}_{\mathsf{rsc}}$. If v does not contain unreceived messages (i.e., it contains only send-receive communication) then $u \cdot w$ and e_m lead to the same configuration, which is absurd by size minimality of e_m (the two executions have the same channels size). Then, suppose that v contains unreceived messages. According to 22, and by contraposition, $e' := u \cdot v$ is deadlock : absurd because the size of channels of e' is strictly smaller than that of e_m. Hence $|e_m| \leq M$. □

A.5 Proof of theorem 24

Theorem 24. *Knowing whether an RSC system is deadlock-free is decidable.*

Proof. For an RSC execution e, we note $\mathsf{cl}_{\sim}(e) := \{e' \in \mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) \,|\, e \sim e'\}$ the equivalence RSC class of e. According to lemma 20, e is deadlock if and only if $\mathsf{cl}_{\sim}(e) \cap \mathscr{L}(\mathscr{A}_{part}) = \varnothing$. According to lemma 23, we just have to check that for all executions of length less than M. □

B Additional Notes on Proofs

B.1 Clarification on the construction of e_m

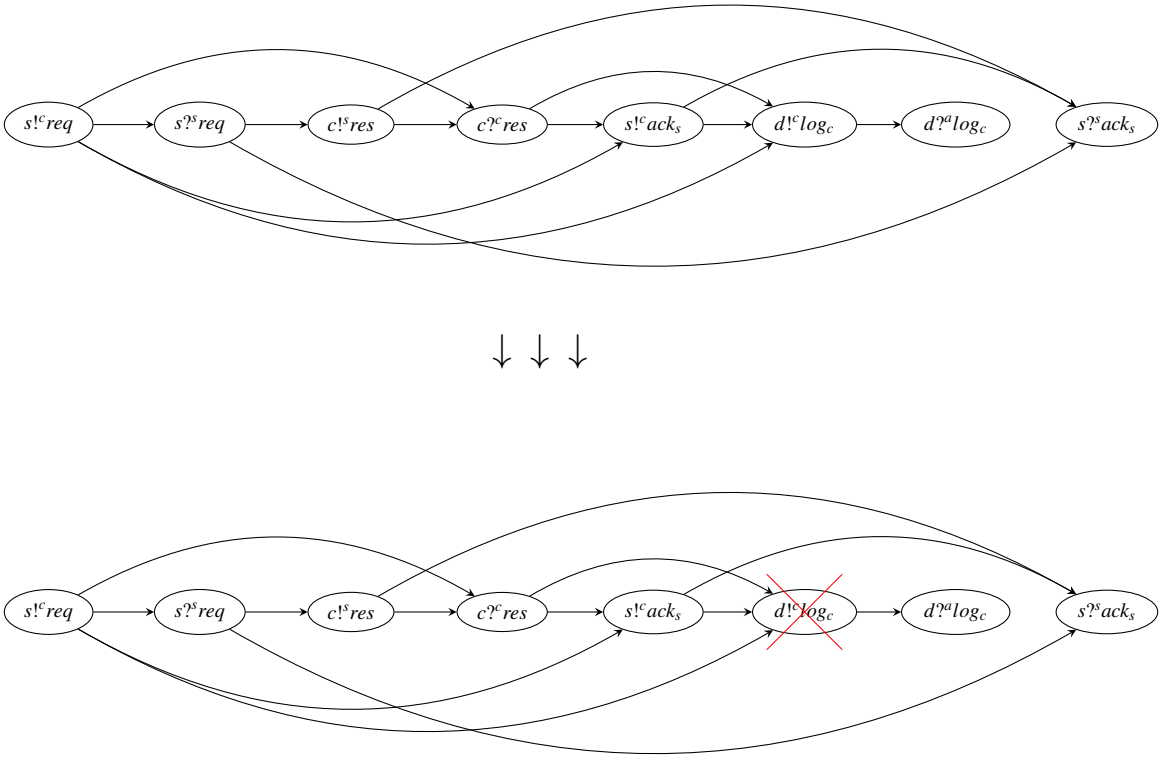
Let $\mathfrak{S}$ be a deadlock RSC system. Define the set $S \subseteq \mathsf{executions}(\mathfrak{S})$ as the set of all deadlock executions : $D := \{e \in \mathsf{executions}(\mathfrak{S}) | e \text{ is deadlock}\}$. Since $\mathfrak{S}$ is deadlock, S is non-empty. Let $\phi : \mathsf{executions}(\mathfrak{S}) \rightarrow$

$\mathbb{N}$ be the function that maps each execution to its buffer size. Since $\phi\left[D\right] \subseteq \mathbb{N}$, it contains a minimal element, which we denote by m. Define the set of deadlock executions with minimal buffer size as : $E_m := \phi^{-1}(m) \cap D$. This set E_m contains all deadlock executions whose buffer size is exactly m, and is therefore minimal with respect to buffer size. Now, $\psi : \mathsf{executions}(\mathfrak{S}) \to \mathbb{N}$ be the function that maps each execution to its length (i.e.,, the number of actions). Then $\psi\left[E_m\right] \subseteq \mathbb{N}$ so has a minimum, which we denote by ℓ. The set $\psi^{-1}(\ell)$ is non-empty and contains all deadlock executions of minimal length among those with minimal buffer size.

B.2 Commentary on the proof of lemma 20

The intuition behind the proof is as follows: when transforming $e \cdot e'$ into the synchronous execution e_{synch} via causal equivalence, the actions from e are simply reordered. The function τ records the original positions of these actions within e. For example, if $e = a_1 \cdot a_2 \cdot a_3$, $e' = a'_1 \cdot a'_2$, and $e \cdot e' \sim e_{synch} = a'_1 \cdot a_2 \cdot a_3 \cdot a'_2 \cdot a_1$, then $\tau(1) = 2$, $\tau(2) = 3$ and $\tau(3) = 1$. Since e_{synch} is causally equivalent to $e \cdot e'$, no conflicts (i.e., swaps of dependent actions) occur with τ—those dependencies are preserved through the equivalence transformation.

C Figures



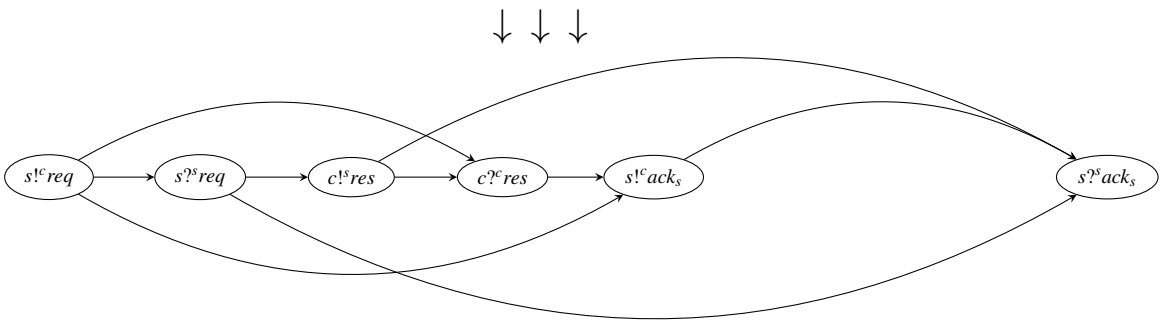


Figure 4: An exemple of action graph and a partial execution.