UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

MÉMOIRE DE MASTER

Mécanisation de la Théorie de la Synchronisabilité pour les Automates Communicants avec Sémantique de Boîte aux Lettres

Nicole KETTLER

Laboratoire d'Informatique, de Signaux et Systèmes de Sophia Antipolis (I3S)
UMR7271 UCA CNRS

Présentée en vue de l'obtention du diplôme de master
en Informatique
d'Université Côte d'Azur

Stage encadré par :
Enrico FORMENTI, Professeur, Université Côte d'Azur
Cinzia DI GIUSTO, Professeur, Université Côte d'Azur
Kirstin PETERS, Professeur, Universität Augsburg
Javier ESPARZA, Professeur, Technische Universität München

Soutenue le : 09.09.2025

MÉCANISATION DE LA THÉORIE DE LA SYNCHRONISABILITÉ POUR LES AUTOMATES COMMUNICANTS AVEC SÉMANTIQUE DE BOÎTE AUX LETTRES

Mechanization of Synchronizability Theory for Communicating Automata with Mailbox Semantics

Nicole KETTLER

Stage encadré par :

Enrico FORMENTI, Professeur, Université Côte d'Azur
Cinzia DI GIUSTO, Professeur, Université Côte d'Azur
Kirstin PETERS, Professeur, Universität Augsburg

Université Côte d'Azur

Nicole KETTLER

Mécanisation de la Théorie de la Synchronisabilité pour les Automates Communicants avec Sémantique de Boîte aux Lettres

ix+42 p.

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : output.tex – 20/8/2025 – 19:05

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

Mécanisation de la Théorie de la Synchronisabilité pour les Automates Communicants avec Sémantique de Boîte aux Lettres

Résumé

Nous étudions les automates communicants, un modèle où les participants d'un protocole de communication sont représentés par des automates finis échangeant des messages. Ces échanges peuvent être synchrones (instantanés) ou asynchrones (via des files d'attente).
Dans des travaux antérieurs [1], nous avons analysé la synchronisabilité, une propriété garantissant que la communication asynchrone ne produit pas de comportements non autorisés par la communication synchrone.
Cet article introduit le problème généralisé de synchronisabilité, en utilisant des automates avec états finaux, et démontre son indécidabilité par une réduction du problème de correspondance de Post. Toutefois, pour certaines topologies comme les arbres, la décidabilité est rétablie en montrant que les langages de tampons sont réguliers et calculables. Cela généralise les résultats obtenus pour les topologies en anneau [4] et ouvre des perspectives vers les structures multi-trees.
Contrairement aux approches précédentes qui se fondaient sur la comparaison des traces d'envoi, cette étude propose une analyse directe du contenu des tampons pour évaluer la synchronisabilité.
Au cours de ce stage, l'objectif est de mécaniser les principales preuves de [1] dans l'assistant de preuve Isabelle.

Mots-clés : Synchronisabilité, Isabelle, Vérification.

Mechanization of Synchronizability Theory for Communicating Automata with Mailbox Semantics

Abstract

We study communicating automata, a model where participants in a communication protocol are represented as finite state machines exchanging messages. These exchanges can be synchronous (instant) or asynchronous (via buffers). In previous work (Di Giusto, Laversa, & Peters, 2024), we explored synchronisability, a property ensuring that asynchronous communication does not introduce behaviors beyond those allowed by synchronous communication.
This paper introduces the Generalised Synchronisability Problem, using automata with final states, and shows it is undecidable by reducing from the Post Correspondence Problem. However, decidability is restored for specific communication topologies such as trees, by proving that buffer languages in these settings are regular and computable. This extends results previously obtained for ring topologies [4], and opens avenues for future work on multitree structures.
A key methodological shift from prior approaches is the direct analysis of buffer contents, instead of comparing send traces, to assess synchronisability.
During this internship, we aim to mechanize the main proofs from (Di Giusto et al., 2024) using the Isabelle proof assistant.

Keywords: Synchronizability, Isabelle, Verification.

Remerciements

I’d like to thank my supervisors, my family and all of my friends.

Table des matières

CHAPITRE 1

Introduction

TODO : better intro

An attempt to bridge the gap between the truth and intuition, is through verification methods. Nowadays, theorem provers can be used to verify with a computer, whether some claims and proofs actually hold. Such an application, also referred to as proof assistant, checks for missing cases or counterexamples, and forces each step of a proof to be correct and complete. To use one, the user needs to rewrite their proofs into the syntax of the theorem prover, which then is able to verify their correctness. In this work, we will make use of the proof assistant Isabelle.

In this internship, the goal was to first get familiar with Isabelle and its syntax *, and then use it for the verification of the proofs in (Di Giusto et al., 2024). We have motivated why the usage of theorem provers in general is instrumental to computer science, now we will go over the paper to be verified.

The paper (Di Giusto et al., 2024) forms the basis of this work, and as such will be referred to as the source paper throughout this work. As part of their contributions, the authors investigated the decidability of synchronizability for Mailbox systems with tree topologies and proposed a method to decide this. One of their perspectives for future work lists the formalization of their main theorems and proofs in a theorem prover, which is this work's objective.

Distributed systems - i.e. systems where multiple peers can communicate with one another - are only verifiable when their communication is synchronous. This is a problem, as a plethora of systems communicate asynchronously instead, e.g. through the usage of buffers. In Mailbox systems, each peer has their own unique "mailbox", which is a buffer of (for us) arbitrary size. We only consider FIFO buffers. This means, whenever a peer q sends a message to some peer p, it is stored in p's buffer, and p starts reading the messages in the order of which they were sent. In other words, messages are read (if, at all) in the same order they were sent in.

Synchronizability is the question of whether an asynchronous system can be equated to one with synchronous communication. For our purposes, this means checking for a given asynchronous system, whether its exact sends can be performed by some synchronous system. This formulation of the problem only relies on the sends (and does not take into account when the receives happen), but it remains undecidable in the general case. We have motivated why verification is important ; thus, we would like to find sub-groups of instances where synchronizability is in fact decidable.

In (Di Giusto et al., 2024), the claim is that synchronizability is decidable for Mailbox systems with FIFO buffers, which we will begin to verify in this work.

todo

— introduce distributed systems

∗. Learning Isabelle could be compared in complexity to learning a programming language from scratch.

— explain verification only possible in synchronously communicating systems

— explain synchronizability checks whether the communication of async system is equal to a synchronous system (thus making it possible to verify that system)

— explain in general synchronizability is undecidable, but there is an ongoing search of certain problems in this domain (e.g. tree topology mailbox), that are decidable

Dans cette partie, qui doit être courte, vous devez présenter le sujet du stage et le contenu du rapport. Vous devez également présenter rapidement le contexte dans lequel s'est déroulé le stage (structure du service, organisation, buts, conditions de travail...). La présentation plus générale de l'entreprise doit être mise en annexe.

Notations

Pâtisserie

$\mathcal{D}$	l'ensemble des donuts
$\mathcal{M}$	l'ensemble des macarons
d_i	le donut i
m_i	le macaron i

Viennoiserie

$\mathcal{V}$	l'ensemble des viennoiseries
$\mathcal{C}$	l'ensemble des croissants
v_i	la viennoiserie i
c_i	le croissant i

CHAPITRE 2

Verification with Isabelle

Dans ce chapitre nous présentons l'histoire de la (k-)synchronisabilité et l'article qui forme la base de cette travail.

In this chapter, we present the history of (k-)synchronizability and the paper which this work is based upon.

2.1 Related Work

As this paper extends the work of (Di Giusto et al., 2024), the related work remains largely the same, with the addition of Isabelle-specific literature. The goal of this work is to verify the claims of the source paper with Isabelle, thus verification is the main topic and the context about synchronizability, etc., is relatively short as a consequence. This is not to diminish the importance of the literature we will highlight, but due to the brief nature of this report, the focus is on our contribution, which does not need much more context other than the source paper.

— (k-)Synchronizability

— 2016

— 2017

— source paper (Extension of k-Synchronisability to Trees)

2.2 Preliminaries

This section will detail all the necessary definitions and results from the source paper needed for the remainder of this work. This work exclusively relies on synchronous systems and Mailbox systems with First-In-First-Out (FIFO) buffers and tree topologies and thus only those will be introduced. Although it should be noted that both in the source paper and in most other related literature, additional topologies and communication protocols are routinely explored. To summarize, our preliminaries here consist of the relevant parts in the "Preliminaries" section of the source paper, as well as some of their fourth section "Synchronisability of Mailbox Communication for Tree-like Topologies". The latter of which contains the claims and results we will attempt to verify throughout this work. Some sections will be direct citations from the source paper, as we necessarily operate on the same definitions. (Di Giusto et al., 2024)

2.2.1 Communicating Automata and Formal Language

For a finite set Σ, a word $w = a_1 a_2 \dots a_n \in \Sigma^*$ is a finite sequence of symbols, such that $a_i \in \Sigma$, for all $1 \leq i \leq n$. We denote the concatenation of two words w_1 and w_2 with $w_1 \cdot w_2$, or simply $w_1 w_2$ if it is clear enough where one word starts and the other begins. The empty word is ε.

For the remainder of this work, some knowledge of non-deterministic finite state automata is necessary. A communicating automaton A is a finite state machine that performs actions of two kinds : either sends or receives which we may also refer to as outputs or inputs, respectively. $\mathcal{L}(A)$ denotes the language accepted by automaton A.

A *network* of communicating automata, or simply a network, is the parallel composition of a finite set $\mathbb{P}$ of participants (commonly referred to as peers) that exchange messages from a finite message set $\mathbb{M}$. We consider only automata - and hence networks - where all states are final. We denote $a^{p \to q} \in \mathbb{M}$ the message sent from peer p to q with finite payload a. Note that $p \neq q$, i.e. each message must have a distinct sender and receiver. An action is the send $!m$ or the reception $?m$ of a message $m \in \mathbb{M}$.

Définition 2.2.1 (Network of Communicating Automata)**.** $N = ((A_p)_{p \in \mathbb{P}}, \mathbb{M})$ is a network of communicating automata, where :

1. for each $p \in \mathbb{P}$, $A_p = (S_p, s_0^p, \mathbb{M}, \rightarrow_p, F_p)$ is a communicating automaton with S_p is a finite set of states, $s_0^p \in S_p$ the initial state, $\rightarrow_p \subseteq S_p \times Act_p \times S_p$ is a transition relation, and F_p a set of final states and
2. for each $m \in \mathbb{M}$, there are $p \in \mathbb{P}$ and $s_1, s_2 \in S_p$ such that $(s_1, !m, s_2) \in \rightarrow_p$ or $(s_1, ?m, s_2) \in \rightarrow_p$.

To identify the flow of messages and hierarchy between peers of a network, we define the topology as graph with edges from senders to receivers.

Définition 2.2.2 (Topology)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network of communicating automata. Its topology is an oriented graph $G(N) = (V, E)$, where $V = \{p \mid p \in \mathbb{P}\}$ and $E = \{(p, q) \mid \exists a^{p\rightarrow q} \in \mathbb{M}\}$.

We will focus exclusively on tree topologies, where each of the inner automata (nodes) receives messages by exactly one other peer. In contrast, the root receives messages from no other peer, and also no peer can send any message to it. The source paper denotes $\mathbb{P}^p_{send}$ (resp. $\mathbb{P}^p_{rec}$) as "the set of participants sending to (resp. receiving from) p". We will specify this further for this work, but we will first continue with the definitions as they were originally introduced.

Définition 2.2.3 (Tree topology)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network of communicating automata and $G(N) = (V, E)$ its topology. $G(N) = (V, E)$ is a tree if it is connected, acyclic, and $\mid \mathbb{P}^p_{send} \mid \leq 1$ for all $p \in \mathbb{P}$.

For networks, many different communication mechanisms and corresponding semantics exist. A *system* with a communication mechanism. We consider only synchronous systems $N_{\texttt{sync}}$, as well as the asynchronous Mailbox systems $N_{\texttt{mbox}}$. In synchronous communication, each send necessitates an immediate receive, whereas in the asynchronous kind, messages are sent to and received from memory (buffers), Here we only consider FIFO (First In First Out) buffers, which can be bounded or unbounded. Thus, asynchronous messages must be received in the same order that they are sent in (or not received at all). As an overview, we use the following two systems throughout this thesis :

— Synchronous (`sync`) : there are no buffers in the system, messages are immediately received when they are sent ;

— Mailbox (`mbox`) : each peer has their own unique buffer, each peer receives all its messages in this buffer, regardless of who sent it.

We use *configurations* to describe the state of a system and its buffers.

Définition 2.2.4 (Configuration)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network. A `sync` configuration (respectively a `mbox` configuration) is a tuple $C = ((s^p)_{p\in\mathbb{P}}, \mathbb{B})$ such that :

— s^p is a state of automaton A_p, for all $p \in \mathbb{P}$

— $\mathbb{B}$ is a set of buffers whose content is a word over $\mathbb{M}$ with :

  — an empty tuple for a `sync` configuration,

  — and a tuple $(b_1, \ldots, b_n)$ for a `mbox` configuration.

We write ε to denote an empty buffer, and $\mathbb{B}^\emptyset$ to denote that all buffers are empty. We write $\mathbb{B}\{b_i/b\}$ for the tuple of buffers $\mathbb{B}$, where b_i is substituted with b. The set of all configurations is

$\mathbb{C}$, $C_0 = \left((s_0^p)_{p\in\mathbb{P}}, \mathbb{B}^\emptyset\right)$ is the *initial configuration*, and $\mathbb{C}_F \subseteq \mathbb{C}$ is the set of *final configurations*, where $s^p \in F_p$ for all participants $p \in \mathbb{P}$. Thus, in our case, all configurations are final.

We describe the behavior of a system with *runs*. A run is a sequence of transitions starting from an initial configuration C_0. Let $\mathtt{com} \in \{\mathtt{sync}, \mathtt{mbox}\}$ be the type of communication. We define $\xrightarrow[\mathtt{com}]{}{}^*$ as the transitive reflexive closure of $\xrightarrow[\mathtt{com}]{}$.

Without loss of generality, we label the transition in the following definitions of executions and traces with the sending message $!a^{p\to q}$ for brevity.

In a synchronous communication, we consider the send and receive actions to occur together without delay ; thus, the synchronous relation sync-send merges these two actions.

Définition 2.2.5 (Synchronous system)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network. The synchronous system $N_{\mathtt{sync}}$ associated with N is the smallest binary relation $\xrightarrow[\mathtt{sync}]{}$ over sync-configurations such that :

$$(\mathtt{sync\text{-}send})\ \frac{s^p \xrightarrow{!a^{p\to q}}_p s'^p \qquad s^q \xrightarrow{?a^{p\to q}}_q s'^q}{\left((s^1,\dots,s^p,\dots,s^q,\dots,s^n), \mathbb{B}^\emptyset\right) \xrightarrow[\mathtt{sync}]{!a^{p\to q}} \left((s^1,\dots,s'^p,\dots,s'^q,\dots,s^n), \mathbb{B}^\emptyset\right)}$$

Définition 2.2.6 (Mailbox system)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network. The mailbox system $N_{\mathtt{mbox}}$ associated with N is the smallest binary relation $\xrightarrow[\mathtt{mbox}]{}$ over mbox-configurations such that for each configuration $C = ((s^p)_{p\in\mathbb{P}}, \mathbb{B})$, we have $\mathbb{B} = (b_p)_{p\in\mathbb{P}}$ and $\xrightarrow[\mathtt{mbox}]{}$ is the smallest transition such that :

$$(\mathtt{mbox\text{-}send})\ \frac{s^p \xrightarrow{!a^{p\to q}}_p s'^p}{((s^1,\dots,s^p,\dots,s^n), \mathbb{B}) \xrightarrow[\mathtt{mbox}]{!a^{p\to q}} ((s^1,\dots,s'^p,\dots,s^n), \mathbb{B}\{b_q/b_q \cdot a\})}$$

$$(\mathtt{mbox\text{-}rec})\ \frac{s^q \xrightarrow{?a^{p\to q}}_q s'^q \qquad b_q = a \cdot b'_q}{((s^1,\dots,s^q,\dots,s^n), \mathbb{B}) \xrightarrow[\mathtt{mbox}]{?a^{p\to q}} \left((s^1,\dots,s'^q,\dots,s^n), \mathbb{B}\{b_q/b'_q\}\right)}$$

To reason about the behavior of the introduced systems, we now define executions and traces. An execution e is a sequence of actions and the corresponding trace t is the projection of this sequence on only the send actions. *

Définition 2.2.7 (Execution)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network and $\mathtt{com} \in \{\mathtt{sync}, \mathtt{mbox}\}$ be the type of communication. $\mathsf{E}(N_{\mathtt{com}})$ is the set of executions, with C_0 as the initial configuration, C_n a final configuration and $a_i \in Act$ for all $1 \le i \le n$, by :

$$\mathsf{E}(N_{\mathtt{com}}) = \{a_1 \cdot \ \dots\ \cdot a_n \mid C_0 \xrightarrow[\mathtt{com}]{a_1} C_1 \xrightarrow[\mathtt{com}]{a_2} \ \dots\ \xrightarrow[\mathtt{com}]{a_n} C_n\}.$$

For a given word w over *actions*, let $w{\downarrow_!}$ (resp. $w{\downarrow_?}$) be its projection on only send (resp. receive) actions, let $w{\downarrow_P}$ its projection on actions that involve only the participants in a set P, let $w{\downarrow_p}$ be its projection on receives to p and sends from p, and let $w{\downarrow_{!?}}$ denote the word over

∗. In Definition 2.2.8, like the source paper (Di Giusto et al., 2024), we do not consider the content of buffers, in contrast to other works like (Finkel & Lozes, 2017), where the authors study stable configurations (where all buffers are empty).

messages in w, which is the result of removing all !- and ?-signs. We extend the operators $\downarrow_!$, $\downarrow_?$, $\downarrow_P$, $\downarrow_p$, and $\downarrow_{\not p?}$ to languages, by applying them on every word of the language. Note that, $w\downarrow_{\{p\}}$ is always empty, since each action involves two distinct peers and never just p, whereas $w\downarrow_p$ is the projection of w to actions in that p has an active role (sender in send actions and receiver in receive actions).

Définition 2.2.8 (Traces)**.** Let $N = ((A_p)_{p\in\mathbb{P}}, \mathbb{M})$ be a network and $\mathtt{com} \in \{\mathtt{sync}, \mathtt{mbox}\}$ be the type of communication. $\mathsf{T}(N_{\mathtt{com}})$ is the set of traces :

$$\mathsf{T}(N_{\mathtt{com}}) = \{e\downarrow_! \mid e \in \mathsf{E}(N_{\mathtt{com}})\}.$$

2.2.2 Synchronizability

We have provided all the general definitions of the source paper, and now move on to more in-depth definitions and notions needed for reasoning about such systems. A system is synchronizable if its asynchronous behavior can be related to its synchronous one. Thus, an asynchronous system is synchronizable if its set of traces is the same as the one obtained from the synchronous system.

Synchronizability We define the *Synchronizability Problem* as the decision problem of determining whether a given system is synchronizable or not.

The authors in (Di Giusto et al., 2024) have shown that the *General Synchronizability Problem* (where states are not constrained in any way) is undecidable for Mailbox systems. Thus, it is also undecidable for our networks where each state is final. As another contribution, they claim that for Mailbox systems with tree topologies, synchronizability is decidable, which we will aim to verify in this work.(Di Giusto et al., 2024)

We now also restate their notion of the shuffled language here, which they use in their main theorem. For a thorough explanation about the reasoning behind this construction, as well as an example [†], we defer to the source paper (Di Giusto et al., 2024). The word w' is a *valid input shuffle of* w, denoted as $w' \sqcup\!\!\sqcup_? w$, if w' is obtained from w by a (possibly empty) number of swappings that replace some $!x?y$ within w by $?y!x$.

Définition 2.2.9 (Shuffled Language)**.** Let $p \in \mathbb{P}$. We define its shuffled language as follows :

$$\mathcal{L}^{\emptyset}_{\sqcup\!\!\sqcup}(p) = \left\{w' \mid w \in \mathcal{L}^{\emptyset}(A_p) \wedge w' \sqcup\!\!\sqcup_? w\right\}$$

2.2.2.1 Sends and Receives

We have now introduced all definitions for which we have kept the formalization. As alluded to earlier, we now wish to specify the sets of $\mathbb{P}^p_{send}$ and resp. $\mathbb{P}^p_{rec}$ a bit further than is done in the original paper. The introduction of these two sets is slightly ambiguous, as one can understand them to either be defined on the network topology, or to be defined on each peer automaton A_p locally. In the former case, we cannot always determine the root directly, as exemplified in Figure 2.1 and elaborated more below. While this has no impact on the overall integrity of the theorem, we choose here to take the full network information into account, to allow for direct root identification and, consequently, clearer reasoning. Thus, throughout this work, we will denote $\mathbb{P}^p_{send}$ and $\mathbb{P}^p_{rec}$ as gathered from the topology.

†. Note that we found a small typo in their Example 4.3 : $!b^{q\to p}?a^{r\to q}$ should not be in the shuffled language, since shuffles can only occur in one direction and not be reversed.

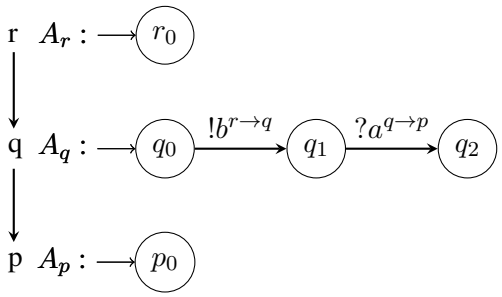

Figure 2.1 – In this network, r can be determined as root globally, but not directly from the local information alone, since both A_r and A_p contain no transitions and hence locally : $\mathbb{P}^r_{send} = \mathbb{P}^r_{rec} = \{\} = \mathbb{P}^p_{send} = \mathbb{P}^p_{rec}$.

In Figure 2.1, only A_q provides information on the messages being sent and received in this network (since both A_r and A_p only produce ϵ). Thus, locally A_r and A_p only know that they send and receive no messages, respectively. In a tree, only the root receives from no other peer, so both A_r and A_p could be the root from just their local information alone. With the addition of A_q, however, we can determine $\mathbb{P}^q_{send} = r$ and $\mathbb{P}^q_{rec} = p$, and with this we can then label r as the root. This highlights that at some point, we can receive the same knowledge about the network locally or from the topology, but the latter is more straightforward. Put differently, locally, $\mathbb{P}^r_{send} =$ does not directly confirm that r is the root (as is the case in 2.1), whereas it does when the full topology is taken into account.

We have now introduced and clarified all definitions, which we will need in the coming parts, but which we have not altered in their formalization, throughout the verification process.

2.3 Formalization Inconsistencies, Errors and Corrections

The aim of this work was to formalize the theorem (and the required lemmas for this) concerning synchronizability in Mailbox networks with tree topologies from (Di Giusto et al., 2024) in the proof assistant Isabelle, to verify their correctness.

Throughout this process, there were multiple moments where proving certain proof steps or lemmas was not possible with Isabelle. This can occur either when smaller lemmas are still needed, which have not yet been specified, or when the to-be-proven step is not actually correct. Because this project involves many of our own definitions and the project itself has a large scope, Isabelle is not able to find counterexamples automatically in most instances. ‡ This prompted a manual search of counterexamples and other formalization errors, which ended up yielding multiple results.

The counterexamples presented in the following sections consist of small but representative instances, found after much deliberation. The initial counterexamples were larger, and thus the progression from one revision to the next might seem less intuitive here than it was in reality, but for brevity, we have chosen to showcase smaller instances that function analogously to their initial drafts, respectively.

— todo : isabelle facts (where code is, how many lines, etc) -> describe the contribution ! ! and why challenging

‡. For easy contradictions, e.g., a lemma that assumes $x = 1$ and shows $x = 0$, Isabelle's automatic checks succeed, and it will warn that this lemma is wrong.

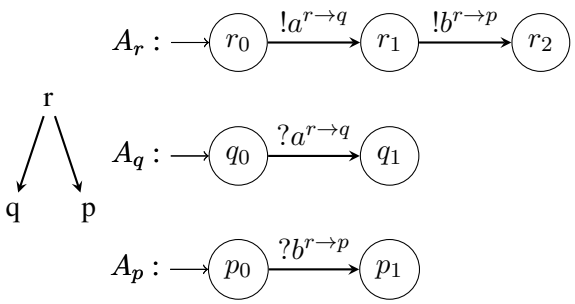

Figure 2.2 – Counterexample 0 : A network where the (original definition of the) influenced language of p contains only ϵ.

— what was there before (reference the original formalization which is also in the github, etc)

(Wenzel, s. d.) (Nipkow, Paulson, & Wenzel, s. d.) (Nipkow & Klein, 2023) (Nipkow, 2023)

2.3.1 Influenced Language / Initial Counterexample

In (Di Giusto et al., 2024), the authors introduce the definition of the influenced language to "capture the influence the automata have on the language of each other". The idea is to remove all words from the language of a peer p, which it cannot reasonably produce given the context of its network. If a peer receives some m, then it must be sent first by its parent, which in turn may have some receives that rely on its parent, and so on, until the root is reached. If somewhere along this unique path from the root to our peer p, some ancestor of p cannot provide a necessary send to its direct child, then p cannot produce its desired word in the end, either. Such words, for which there is no valid execution, are not present in the influenced language. This was formalized into the following definition :

Définition 2.3.1 (Original Influenced Language)**.** Let $p \in \mathbb{P}$. In (Di Giusto et al., 2024), the influenced language and its projections to sends/receives are defined as follows :

$$\mathcal{L}^{\emptyset}(A_p) = \begin{cases} \mathcal{L}(A_r) & \text{if } p = r \\ \left\{ w \mid w \in \mathcal{L}(A_p) \wedge (w{\downarrow_?}){\downarrow_{\not !?}} \in \left(\mathcal{L}^{\emptyset}_!(A_q)\right){\downarrow_{\not !?}} \wedge \mathbb{P}^p_{send} = \{q\} \right\} & \text{otherwise} \end{cases}$$

$$\mathcal{L}^{\emptyset}_?(A_p) = \mathcal{L}^{\emptyset}(A_p){\downarrow_?}$$

$$\mathcal{L}^{\emptyset}_!(A_p) = \mathcal{L}^{\emptyset}(A_p){\downarrow_!}$$

However, as it is written down, this formalization is too strong : it removes words for which there are valid executions, which is undesired. For trees where each parent sends to at most one child, respectively, this definition works as expected. However, if any parent sends a message to two children within one word, the influenced language removes this word, even if there is a valid execution for it. To exemplify this, consider Figure 2.2.

In this synchronous network, there is an execution where p can perform $?b^{r\to p}$, since r can send the matching message as its second action. Here, r is the root and p's only ancestor, thus r

requires no sends (as it has no ancestors) and there is no other peer on the path from r to p. Hence, one would expect $?b^{r\to p} \in \mathcal{L}^{\emptyset}(A_p)$, but actually

$$\mathcal{L}^{\emptyset}(A_r) = \mathcal{L}(A_r) = \{\varepsilon, !a^{r\to q}, !a^{r\to q}!b^{r\to p}\} = \mathcal{L}^{\emptyset}_{!}(A_r) \qquad \text{since } r \text{ is root}$$

We consider all states accepting, thus $\mathcal{L}(A_p) = \{\varepsilon, ?b^{r\to p}\}$ can be inferred from the automaton in Figure 2.2. With this, and knowing that

$$(?b^{r\to p})\downarrow_{!?} = b^{r\to p},$$

we can perform the check in the second part of the conjunction of the influenced language definition :

$$\{\varepsilon,\ !a^{r\to q},\ !a^{r\to q}!b^{r\to p}\}\downarrow_{!?} = \{\varepsilon,\ a^{r\to q},\ a^{r\to q}b^{r\to p}\},$$
$$b^{r\to p} \notin \{\varepsilon,\ a^{r\to q},\ a^{r\to q}b^{r\to p}\}.$$

Since this check fails, $?b^{r\to p} \notin \mathcal{L}^{\emptyset}(A_p) = \{\varepsilon\}$, i.e. the definition is not working as intended. To summarize, the influenced language is meant to remove words from peers for which there are no valid executions anyways, i.e. to remove uninteresting [§] cases for the synchronizability decision procedure.

In fact, this issue directly harms the integrity of the original main theorem in (Di Giusto et al., 2024). The illustrated network is synchronizable, since the only traces possible are exactly the words in $\mathcal{L}^{\emptyset}(A_r)$, and both children are ready to receive without any blocking sends. Thus, we satisfy the left side of the *if and only if* of the original theorem below :

Définition 2.3.2 (Original Mailbox Synchronizability Theorem)**.** Let N be a network such that $\mathbb{C}_F = \mathbb{C}$ and $G(N)$ is a tree. Then $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$ iff, for all $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, we have $\left(\mathcal{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}}\right)\downarrow_{!?} \subseteq \mathcal{L}^{\emptyset}(A_p)\downarrow_{!?}$ and $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$.

We will refer to the rightmost part of the theorem ($\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$) as the *shuffled language condition*, and to the other condition on the right side (currently : $(\mathcal{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}})\downarrow_{!?} \subseteq \mathcal{L}^{\emptyset}(A_p)\downarrow_{!?}$) as the *subset condition*.

We should consequently satisfy the right side as well. However, for p and r, where $\mathbb{P}^p_{send} = \{r\}$, we have

$$(\mathcal{L}^{\emptyset}_{!}(A_r)\downarrow_{\{p,r\}})\downarrow_{!?} = \{\varepsilon,\ !b^{r\to p}\}\downarrow_{!?} = \{\varepsilon,\ b^{r\to p}\},$$
$$\mathcal{L}^{\emptyset}(A_p)\downarrow_{!?} = \{\varepsilon\},$$
$$(\mathcal{L}^{\emptyset}_{!}(A_r)\downarrow_{\{p,r\}})\downarrow_{!?} = \{\varepsilon,\ b^{r\to p}\} \not\subseteq \{\varepsilon\} = \mathcal{L}^{\emptyset}(A_p)\downarrow_{!?}.$$

To conclude, the definition in the source paper not only does not fully work as intended, but in turn renders the entire theorem [¶] to be false. The solution to this oversight is to project $\mathcal{L}^{\emptyset}_{!}(A_q)$ in the definition 2.3.1 to the two peers p and q. Since $\mathcal{L}^{\emptyset}_{!}(A_q)$ is the language - projected to only

§. A trace requires at least one valid execution, i.e. one that is possible for the network, thus if an execution is not possible, it cannot produce a trace, either.

¶. Both Lemma 4.4 and Theorem 4.5 in (Di Giusto et al., 2024) are affected by this, but the fix for both is to add the projection to the set $\{p, q\}$, so Lemma 4.4 is not explicitly stated for brevity.

sends - of A_q, it only contains actions of q. We need to ensure that only sends from q to p are considered, thus a projection on *actions where p is the second participant*, i.e., here the recipient, is necessary. In the remainder of this work, whenever we refer to the *influenced language* of some peer and its projections (as defined in the original definition), we will refer instead to this revised version :

Définition 2.3.3 (Revised Influenced Language)**.** Let $p \in \mathbb{P}$. We define the influenced language as follows :

$$\mathcal{L}^{\emptyset}(A_p) = \begin{cases} \mathcal{L}(A_r) & \text{if } p = r \\ \left\{ w \mid w \in \mathcal{L}(A_p) \wedge (w{\downarrow_?}){\downarrow_{\not{!}\not{?}}} \in \left((\mathcal{L}^{\emptyset}_{!}(A_q)){\downarrow_{\{p,q\}}} \right){\downarrow_{\not{!}\not{?}}} \wedge \mathbb{P}^p_{send} = \{q\} \right\} & \text{otherwise} \end{cases}$$

The projections of the influenced language to sends and receives, respectively, remain the same, thus they are not repeated here. The only difference to the original definition is the added pair projection in the second case of the conjunction. Lastly, we can also apply the definition in the following manner : given a $w \in \mathcal{L}^{\emptyset}(A_p)$ we can infer that there exists some $w' \in \mathcal{L}^{\emptyset}(A_q)$ such that $((w'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not{!}\not{?}}} = (w{\downarrow_?}){\downarrow_{\not{!}\not{?}}}$.

2.3.2 Revisions of the Main Theorem

Starting with this section, we will describe the iterative process of contradicting and then revising the theorem 4.5 of (Di Giusto et al., 2024). The original theorem can be referenced in Theorem 2.3.2, and we will from here on use this theorem with our revised influenced language from Definition 2.3.3. As explained below the Theorem 2.3.2, we refer to the two conditions on the right side as subset condition and shuffled language condition, respectively. The subset condition will be revisited in the following sections, but it will always be noted which revision, i.e. which specific instance of the subset condition definition we are currently applying. The theorem will change with each revision, the general construction will remain the following, where **the subset condition** is a place holder for any of its revisions :

Définition 2.3.4 (Mailbox Synchronizability Theorem General Scheme)**.** Let N be a network such that $\mathbb{C}_F = \mathbb{C}$ and $G(N)$ is a tree. Then $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$ iff, for all $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, we have **the subset condition** and $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ (the shuffled language condition).

2.3.2.1 First Counterexample

For this counterexample, we are using the original Theorem 2.3.2. Since the influenced language occurs on the right side of the original theorem, and the influenced language has already posed a problem, we explored the possibility of there being more issues related to its usage. The original subset condition only requires the sends of a parent to be receivable by the respective children. It doesn't restrict other branches, however. This can be exploited by having one branch in the child-automaton which receives exactly the sends to it from the parent, and another branch which does some send, but performs no receives. With such a network as in Figure 2.3, the right side of the theorem is satisfied, but we cannot perform the trace $!a^{r\to q}!b^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$ synchronously. q cannot receive $a^{r\to q}$ before, or after sending $!b^{q\to p}$, as the two actions are on different branches of A_q and thus mutually exclusive. In Mailbox, however, the message can be sent, since

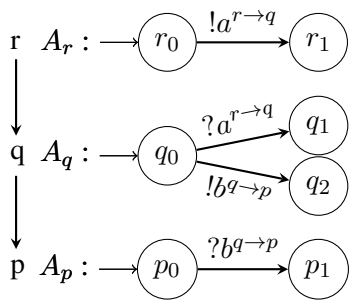

Figure 2.3 – Counterexample 1 : A network that is not synchronizable, but satisfies the original right side of the theorem

no message ever *needs* to be received. Thus, we have found a trace that is in $\mathsf{T}(N_{\texttt{mbox}})$ but not in $\mathsf{T}(N_{\texttt{sync}})$, which contradicts the *if and only if* in the original theorem, as the right side is satisfied. The shuffled language condition is trivially satisfied, since each possible word in any peer's language consists of exactly one action only, not allowing for a shuffle. The subset condition is also satisfied, as is inferred in the following :

$$\mathcal{L}(A_r) = \{\varepsilon, !a^{r\to q}\} = \mathcal{L}^{\emptyset}(A_r) = (\mathcal{L}^{\emptyset}_{!}(A_r))\downarrow_{\{p,q\}}$$
$$\mathcal{L}(A_q) = \{\varepsilon, ?a^{r\to q}, !b^{q\to p}\} = \mathcal{L}^{\emptyset}(A_q) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(q)$$
$$\mathcal{L}(A_p) = \{\varepsilon, ?b^{q\to p}\} = \mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$$

Then,

$$((\mathcal{L}^{\emptyset}_{!}(A_r))\downarrow_{\{p,q\}})\downarrow_{!?} \subseteq (\mathcal{L}^{\emptyset}(A_q))\downarrow_{!?}$$

since,

$$\{\varepsilon, !a^{r\to q}\}\downarrow_{!?} \subseteq \{\varepsilon, ?a^{r\to q}, !b^{q\to p}\}\downarrow_{!?}$$

We can observe, the original subset condition is not strong enough. In particular, the problem here is that something is sent by the child, but after that initial send, it may not be able to receive everything the parent could send anymore. Put differently, sends as prefixes to the words of some peer might pose problems concerning the synchronizability. With this in mind, we revised the subset condition to the following :

Définition 2.3.5 (Subset Condition First Revision)**.** Let $outputprefixes$ denote all prefixes $w \in \mathcal{L}^{\emptyset}(A_p)$, s.t. $w\downarrow_{!} = w$, i.e. $outputprefixes$ is the set of words in $\mathcal{L}^{\emptyset}(A_p)$ consisting only of outputs. Then, for $p, q \in \mathbb{P}$ with $\mathbb{P}^{p}_{send} = \{q\}$

$$\forall w \in outputprefixes\left(\mathcal{L}^{\emptyset}(A_p)\right).\left((\mathcal{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}})\right)\downarrow_{!?} \subseteq \{x | wx \in \mathcal{L}^{\emptyset}(A_p) \wedge x = x\downarrow_{?}\}\downarrow_{!?}$$

The definition checks for each output prefix w of some child p with parent q, whether there is some suffix x, such that x consists only of inputs and $w \cdot x \in \mathcal{L}^{\emptyset}(A_p)$. This is to enforce that after performing any amount of initial send actions, there is still some suffix x that p can perform, which receives all the sends its parent might send.

2.3.2.2 Second Counterexample

Continuing on with the revised subset condition from Definition 2.3.5. This condition specifically only checks output prefixes; thus, the idea arose to check whether prefixes starting in

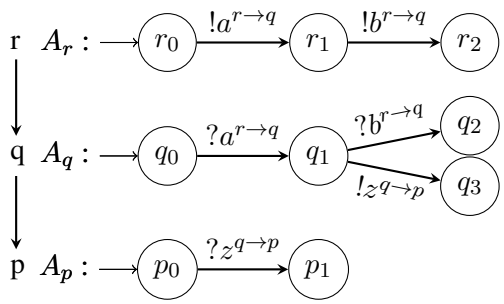

Figure 2.4 – Counterexample 2 : A network that is not synchronizable, but satisfies the right side of the theorem after the first revision of the subset condition.

inputs can break this new condition. This lead turned out to be correct, resulting in the next counterexample in Figure 2.4. This new network functions analogously to the previous one, with the exception that A_r sends a second message to q, so q can first perform an input, and otherwise the automata have not changed. r and q now first exchange the message $a^{r\to q}$ and then have the same structure as the previous example (after states r_1 and q_1, respectively).

Since it functions the same as the previous example, we will not reiterate the equivalences needed to contradict the theorem here, again. The problematic trace for this network is $!a^{r\to q}!b^{r\to q}!z^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$ and $!a^{r\to q}!b^{r\to q}!z^{q\to p} \notin \mathsf{T}(N_{\texttt{sync}})$. For the only existing output prefix ϵ of both q and p respectively, the revised subset condition is satisfied. This is because from the initial state, descendants of the root can receive all the sends they might receive. For example, q is able to receive all possible sends of r, namely $\epsilon, a^{r\to q}$ and $a^{r\to q}b^{r\to q}$. However, once q has entered the branch where it sends $z^{q\to p}$, it can no longer receive $b^{r\to q}$, even though r could have already sent it or can still send it in the future. To conclude, the issue is that words starting with receives may also clash with the possible synchronizability of the network, e.g. if some branching like in our examples occurs.

Taking this into account, we expanded the previous subset condition to take into account *all* possible prefixes, which are exactly all words in the language of a peer, since all states are accepting. The second iteration of the subset condition can be found in Definition 2.3.6 :

Définition 2.3.6 (Subset Condition Second Revision)**.** Let *is prefix of* refer to the non-strict standard prefix relation, e.g. $!a^{q\to p}$ is a prefix of $!a^{q\to p}$. Then, for $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, have $\forall w \in \mathcal{L}^{\emptyset}(A_p)$.
$\mathcal{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}}\downarrow_{!?} \subseteq \{yx | wx \in \mathcal{L}^{\emptyset}(A_p) \wedge x = x\downarrow_{?} \wedge \text{y is prefix of } (w\downarrow_{?})\}\downarrow_{!?}$

2.3.2.3 Third Counterexample

With the latest revision of the subset condition, as denoted in Definition 2.3.6, we now enforce that parent sends can *always* be received fully, regardless of the child's current prefix. In other words, no matter which prefix we have read in the child automaton, we can still perform the matching receives that might come from the parent. Another addition to this revision is that the right set considers the receives that have already occurred in the child, as well. This was another oversight, since otherwise we would force the child to receive something again, even if it already received it. The *prefix* relation is used to capture the ϵ, which is in each peer's language and would not be captured in this condition if $w\downarrow_{?} \neq \epsilon$.

One can see directly, that the right set in the subset relation is more nuanced than the left. This sparked the idea that perhaps the left set is too general, i.e. that now the condition has become too strong. We enforce that regardless of the prefix $w \in \mathcal{L}^{\emptyset}(A_p)$, *any* send of the parent can still be received (if it hasn't been received already). This does not consider the case in which the parent q has already entered some branch in which it can no longer send *all* the words in $\mathcal{L}^{\emptyset}(A_q)$ anymore, and consequently also not those in $\mathcal{L}^{\emptyset}_{!}(A_q)\downarrow_{\{p,q\}}$. Since we now allow for inputs in the prefixes of p as well, we indirectly force q to take certain actions.

Consider the network in Figure 2.5, which illustrates the issue that arises by not considering the parent's branching. Here, r can either send $a^{r\to q}$, or $b^{r\to q}$, as its first nonempty action. Consider $w = ?a^{r\to q} \in \mathcal{L}^{\emptyset}(A_p)$, then in all valid executions where q can perform this action, r must have sent it already, as q can only receive when that exact message is in in the first position of its buffer. Then, the possible values for y in the right set of our subset condition are $y \in \{\epsilon, ?a^{r\to q}\}$, as those are the prefixes of our w. Since q must have entered its lower branch by performing $?a^{r\to q}$, the only possibility for x is $x \in \{\epsilon\}$. Then we can infer,

$$\{yx | wx \in \mathcal{L}^{\emptyset}(A_q) \wedge x = (x\downarrow_{?}) \wedge \text{y is prefix of } (w\downarrow_{?})\}\downarrow_{!?} = \{\epsilon, ?a^{r\to q}\}\downarrow_{!?} = \{\epsilon, a^{r\to q}\} \tag{2.1}$$

$$\mathcal{L}^{\emptyset}_{!}(A_r)\downarrow_{\{q,r\}}\downarrow_{!?} = \{\varepsilon, !a^{r\to q}, !b^{r\to q}, !b^{r\to q}!a^{r\to q}\}\downarrow_{!?} = \{\varepsilon, a^{r\to q}, b^{r\to q}, b^{r\to q}a^{r\to q}\} \tag{2.2}$$

$$\{\varepsilon, a^{r\to q}, b^{r\to q}, b^{r\to q}a^{r\to q}\} \not\subseteq \{\epsilon, a^{r\to q}\} \tag{2.3}$$

Combining all previous equations, we then have

$$\left((\mathcal{L}^{\emptyset}_{!}(A_r))\downarrow_{\{q,r\}}\right)\downarrow_{!?} \not\subseteq \{yx | wx \in \mathcal{L}^{\emptyset}(A_q) \wedge x = x\downarrow_{?} \wedge \text{y is prefix of } (w\downarrow_{?})\}\downarrow_{!?} \tag{2.4}$$

It is clear that the subset condition does not hold for this network. However, this network is synchronizable, contradicting the current formalization of the main theorem (2.3.4) once again. The network is synchronizable, since there are only two peers, and only the root r is performing sends, which q can all receive. In other words, the only possible traces are exactly the language of r : $\mathcal{L}(A_r) = \{\varepsilon, !a^{r\to q}, !b^{r\to q}, !b^{r\to q}!a^{r\to q}\}$. By observation, it is clear that q can receive each of the sends belonging to each respective trace in this set. Thus, $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$ and the network is synchronizable.

To conclude, this counterexample contradicts the current theorem, because the subset condition is now too strong. It does not track which actions are even still possible for the parent, but instead forces the child to be able to receive all sends to it, starting from the parent's initial state. This is undesired, as in our example, it forces q to be able to receive sends that r cannot perform anymore because it must have sent $!a^{r\to q}$ already. By sending $!a^{r\to q}$ so that q can receive $?a^{r\to q}$, both q and r must enter their respective lower branches (states r_3 and q_3). From r_3, r cannot send $!b^{r\to q}$ or $!b^{r\to q}!a^{r\to q}$ anymore, but the condition still forces q to be able to receive those two impossible sends.

This in turn means, that we must track both the parent and the child, depending on which receives have already occurred. Since if the child receives a message, the parent must have sent it. Thus we can disregard all words of the parent, where these matching sends have either not occurred (completely), or are not in the correct order. This led us to the third and final revision of the subset condition in Definition 2.3.7 and the matching rendition of the main theorem in Theorem 2.3.1.

2.3.2.4 Final Formalization

As touched upon lightly in the previous section, the final subset condition now tracks both the parent q and the child p for each parent-child pair. For each word w of the child, the condition

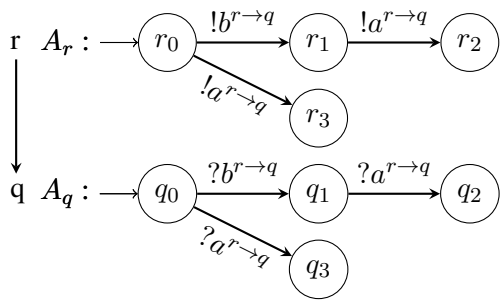

Figure 2.5 – Counterexample 3 : A network that is synchronizable, but does not satisfy the right side of the theorem after the second revision of the subset condition.

tracks all possible w' in the parent's language, which provide *exactly* the matching sends to the receives $w{\downarrow_?}$ of w. We do not constrict these w' otherwise. In particular, w' may contain any number of sends to children other than p, or receives if q is not the root. We only choose the words that exactly provide what p needs from q, but disregard all actions to other peers. This ensures that w' really captures all possible words for q, and in turn all possible executions that might hinder the synchronizability of a network. We know that at least one such w' must exist, otherwise $w \notin \mathcal{L}^{\emptyset}(A_p)$ by Definiton 2.3.3. Now for every such pair w and w', we need to ensure that for possible suffixes x' s.t. $w'x' \in \mathcal{L}^{\emptyset}(A_q)$, there is at least one suffix x, s.t. $((x'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = (x{\downarrow_?}){\downarrow_{\not!?}}$. Put into words, we check for the current branch of q and of p, whether p can still receive everything q is still able to send after that point. With this, we have weakened the condition again slightly, so that it checks only the words that can be reached, given the current prefix of p and the resulting words of q.

Définition 2.3.7 (Subset Condition Final Revision)**.** For $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, we have
$\forall w \in \mathcal{L}^{\emptyset}(A_p).\ \forall w' \in \mathcal{L}^{\emptyset}(A_q) : \Big(((w'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = (w{\downarrow_?}){\downarrow_{\not!?}}\Big) \longrightarrow$
$\Big(\{(x'{\downarrow_!}){\downarrow_{\{p,q\}}} \mid w'x' \in \mathcal{L}^{\emptyset}(A_q)\}{\downarrow_{\not!?}} \subseteq \{x{\downarrow_?} \mid wx \in \mathcal{L}^{\emptyset}(A_p)\}{\downarrow_{\not!?}}\Big)$

Théorème 2.3.1 (Final Mailbox Synchronizability Theorem)**.** *Let N be a network such that $\mathbb{C}_F = \mathbb{C}$ and $G(N)$ is a tree. Then $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$ iff, for all $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$, we have the subset condition (1) and the shuffled language condition (2) :*

(1) $\forall w \in \mathcal{L}^{\emptyset}(A_p).\ \forall w' \in \mathcal{L}^{\emptyset}(A_q) : \Big(((w'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = (w{\downarrow_?}){\downarrow_{\not!?}}\Big) \longrightarrow$
$\Big(\{(x'{\downarrow_!}){\downarrow_{\{p,q\}}} \mid w'x' \in \mathcal{L}^{\emptyset}(A_q)\}{\downarrow_{\not!?}} \subseteq \{x{\downarrow_?} \mid wx \in \mathcal{L}^{\emptyset}(A_p)\}{\downarrow_{\not!?}}\Big)$

(2) $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$

Now that our final version of the theorem is formalized, which we hope is correct, we will start proving and verifying it. The original proof of the theorem in (Di Giusto et al., 2024) is now sadly mostly obsolete, as the change of both the influenced language and subset condition affects the previous reasoning a great amount.

In the source paper, the following lemma is proposed and proven, albeit unverified :

Lemme 2.3.2. *Let N be a network such that $\mathbb{C}_F = \mathbb{C}$, $G(N)$ is a tree, $q \in \mathbb{P}$, and $w \in \mathcal{L}^{\emptyset}(A_q)$. Then there is an execution $w' \in \mathsf{E}(N_{\texttt{mbox}})$ such that $w'{\downarrow_q} = w$ and $w'{\downarrow_p} = \varepsilon$ for all $p \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$.*

This lemma is built upon the original version of the influenced language in Defintion 2.3.1, which we have revised to Definiton 2.3.3 for this work. This has implications for the proof in the paper as well, since it has the same oversight as the original influenced language. In fact, the proof can be corrected by adding the missing projection to $\downarrow_{\{p,q\}}$ to the parts of the proof where the influenced language is actively used. Apart from this, the original proof follows directly from the construction of the influenced language, and thus we will not rewrite the entire proof here, as adding the projections to the original proof ‖ is the only change needed. The proof works as follows. First, we obtain the unique ** path from the root r to the peer q. Since $w \in \mathcal{L}^{\emptyset}(A_q)$, either $q = r$ and we are done, or q has some parent h, and by the definition of the influenced language, there is some $w' \in \mathcal{L}^{\emptyset}(A_h)$ such that $((w'\downarrow_!)\downarrow_{\{q,h\}})\downarrow_{!?} = (w\downarrow_?)\downarrow_{!?}$. This reasoning can be repeated until the root is reached. Simultaneously, we accumulate all the visited words s.t. in the end we receive a word $w_r \dots w'w$, where the leftmost component is the root word, and the rightmost component is the word of q we started with. Through this construction, all sends happen in the correct order and all existing receives also happen in the correct order. We must make the distinction of existing receives, since we only visit the peers on the path, and so only these peers may perform actions in the accumulated word. However, peers can send messages to multiple children, so there may be some sends in the accumulation that are never received, but this is not a problem as we are in the asynchronous case within this lemma. To summarize, we can accumulate this word, by the construction of the influenced language, and this word is a valid Mailbox execution, because all sends happen before the receives (by construction), and all receives happen in the correct order.

We verified most steps of (the proof of) this lemma in Isabelle, but some minor results are missing. Most missing parts are fairly clear, however, for example the property that a unique path from the root to q exists. This should follow directly from the tree topology, as a tree is connected and each peer has at most one parent. Having warned the reader that this lemma is not completely verified, we are going to assume this lemma to be correct and use it in the proof of the main theorem.

Next, we will present the adjusted proof for the main theorem. The Isabelle code for this can be found in Appendix .1.4, but we will detail the proof here in a more regular format. Our new proof is almost completely different in execution from the one presented in the source paper, due to the subset condition changing, with the exception of one case of the backwards direction.

For proving the direction that assumes syncrhonizability and shows our two conditions hold, we need to define two constructions first. In the following, we will quickly explain the main idea and show pseudocode for those constructions. The Isabelle code for them - which is quite similar in syntax to functional programming languages - can be referenced in Appendices .1.1 and .1.2. respectively. Let $w[i]$ denote word w at index i. It is important to note, that the following two constructions implicitly assume that each message is unique, i.e. there is no such case where the message a is sent between more than one unique pair of peers. Additionally, within the language of one peer, all messages are also distinct, this follows from the previous point. ††

For $=>$ 1., we use a construction which mixes two words $w \in \mathcal{L}^{\emptyset}(A_p)$ and $w' \in \mathcal{L}^{\emptyset}(A_q)$, where p is q's direct child, in the following manner :

‖. The original lemma is "Lemma 4.4" in (Di Giusto et al., 2024), and its proof is directly below.

**. Since the network is a tree, there can only be *exactly one* path between each peer pair, respectively.

††. If messages are not unique, they can be made unique e.g. by adding subscripts that tag the peers involved to the message.

Définition 2.3.8. (mix_pair Pseudocode)

Iterate through w with (index i) and w' (with index j).
In the following cases, let a denote an arbitrary message (from q to p).

Cases :

- $w'[j] = !a^{q\to p}$:
  if $w[i] = ?a^{q\to p}$ then put $!a^{q\to p}, ?a^{q\to p}$and increment both i and j by one, respectively ;
  else put $w[i]$ and increment i until $w[i] = ?a^{q\to p}$.
- $w'[j] \neq !a^{q\to p}$:
  put $w'[j]$ and increment j.

We denote with (mix_pair (w', w)) a call to the function defined above over w' and w. Intuitively, whenever a send from q to p occurs in w', we append the matching receive of p for that message directly afterwards. All actions not concerning p and q, e.g. sends and receives from q involving other peers, or sends from p to its children, are appended s.t. their original order is kept. In particular, (mix_pair (w', w))$\downarrow_q = w'$ and (mix_pair (w', w))$\downarrow_p = w$ hold. We alternate sends and receives of p and q. If the returned word is part of an execution e, s.t. $e \in \mathsf{E}(N_{\texttt{mbox}})$ and e contains no other actions of p or q, then a synchronous execution with the same trace as e would also satisfy $e\downarrow_q = w'$ and $\downarrow_p = w$.

For $=>$ 2., we utilize mix_pair as explained above to construct a slightly different version of a mix between two words.

Définition 2.3.9 (mix_triple)**.** Let $p, q, x \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}, \mathbb{P}^x_{send} = \{p\}$, let $(xs{\cdot}!a^{p\to x}{\cdot}?b^{q\to p} \cdot ys) = w \in \mathcal{L}^{\emptyset}(A_p)$ and $(xs{\cdot}?b^{q\to p}{\cdot}!a^{p\to x} \cdot ys) = w' \in \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ s.t. $w' \sqcup\!\sqcup_? w$.
Let $(as{\cdot}!b^{q\to p} \cdot bs) = w'' \in \mathcal{L}^{\emptyset}(A_q)$.
Then we construct
mix_triple $(w'', w', w) :=$ ((mix_pair (as, xs))$\cdot!b^{q\to p}{\cdot}!a^{p\to x}{\cdot}?b^{q\to p} \cdot$ (mix_pair (bs, ys)))

As intuition, w and w' in this construction are equal, except for one input-output pair in the middle, i.e. exactly one shuffle occurred from w to w'. We can infer (mix_triple (w'', w', w))$\downarrow_q = w''$, by the construction of (mix_pair (w'', w)). We can observe that (mix_triple (w'', w', w)) performs (mix_pair(w'', w)), except for the part where w' and w differ. However, (mix_triple (w'', w', w))$\downarrow_p = w$, since the ordering is kept. To showcase the effect of this construction, consider an execution e, s.t. (mix_triple (w'', w', w)) occurs as a whole in $e \in \mathsf{E}(N_{\texttt{mbox}})$ and $e\downarrow_q = w''$ and $\downarrow_p = w$. Due to the placement of $!b^{q\to p}$ before $!a^{p\to x}{\cdot}?b^{q\to p}$, a synchronous execution e' with the same trace as e would now satisfy $\downarrow_p = w'$. Otherwise, not every send would immediately be followed by the matching receive, and e' wouldn't be synchronous. We will explain how this and the first construction are relevant when they occur in the proof, as without their context, they may seem quite arbitrary.

As hinted at before, most of our proof execution is entirely different from the source, but some of the proof structure and ideas were extremely helpful in our formalization. The original proof uses a similar approach in the first direction of the theorem, but their construction is not sufficiently specific, and thus the claim is not proven in all cases of the arbitrary words they fix. For first case of the second direction, we operate with the same idea, but due to the subset condition becoming stricter, this proof part also had to become more elaborate. In other words, in some of the original proof, cases were missed needed to prove the claim, and in other parts of the proof, we have

changed the formalization. Because of this, most of the original proof is obsolete regardless of the exact reasons; thus, we will not detail all formalization errors or small oversights in of original proof, and instead provide and explain our new version below. In general, we have made some of the implicit steps from the source explicit, and provided definitions for some constructions used. To conclude, for the first direction, the proof idea is similar to the original one, although the execution is different and more detailed. For the second direction, the case distinction and the proof for the second case remain true to the original, but the first case is completely changed due to the change in subset condition.

Proof of Final Theorem.

$\Rightarrow$ Assume $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$. We have to show that

1. $\forall w \in \mathcal{L}^{\emptyset}(A_p).\ \forall w' \in \mathcal{L}^{\emptyset}(A_q) : \Big(((w'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = (w{\downarrow_?}){\downarrow_{\not!?}}\Big) \longrightarrow$
$\Big(\{(x'{\downarrow_!}){\downarrow_{\{p,q\}}} \mid w'x' \in \mathcal{L}^{\emptyset}(A_q)\}{\downarrow_{\not!?}} \subseteq \{x{\downarrow_?} \mid wx \in \mathcal{L}^{\emptyset}(A_p)\}{\downarrow_{\not!?}}\Big)$

and 2. $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ for all $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$.

1. Assume $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$. To show : the subset condition holds.
Proof : Fix w, w' s.t. $\Big(((w'{\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = (w{\downarrow_?}){\downarrow_{\not!?}}\Big)$ as in the precondition of the subset condition; fix x' s.t. $w'x' \in \mathcal{L}^{\emptyset}(A_q)$. We now have to show, that there exists some x with $wx \in \mathcal{L}^{\emptyset}(A_p)$ and $(((w'x'){\downarrow_!}){\downarrow_{\{p,q\}}}){\downarrow_{\not!?}} = ((wx){\downarrow_?}){\downarrow_{\not!?}}$, which is the subset condition but rewritten on words instead of subsets. With these assumptions, we will now do a case distinction over whether q is the root or not.
If q is the root, then $w'x'$ consists only of sends and hence $w'x' \in \mathsf{E}(N_{\texttt{mbox}})$, because $w'x'$ cannot contain receives and sends cannot be blocked without receives.
Then obtain $w_{mix} = (\text{mix\_pair}\ (w', w))$ and append x' to this to receive the execution $e = w_{mix} \cdot x' \in \mathsf{E}(N_{\texttt{mbox}})$. This is an execution, since both w' and x' cannot receive anything from any other peer (as q is the root), and w' provides all necessary sends and in the correct order for w, by assumption. Then using synchronizability, obtain a synchronous execution e' for the trace of the constructed execution, s.t. $e{\downarrow_!} = e'{\downarrow_!}$ and $e' \in \mathsf{E}(N_{\texttt{sync}})$. Then we can obtain x where $e'{\downarrow_p} = w \cdot x$. This is possible, since p only receives from q and by the construction of w_{mix}, each send of q to p in w' is directly followed by the receive in w. If x' now contains any further sends to p, p will perform the receives in e' after performing the actions in w. Thus, $e'{\downarrow_p}$ consists of exactly w and some x as a peer word of p, which receives $w' \cdot x'$ exactly, and the subset condition is proven if q is the root.
This leaves open the case where q is some node in the tree with some parent r. The construction process of the two executions is analogous, but since q is not the root, finding e is more complicated. First, we obtain some w'' s.t. $\Big(((w''{\downarrow_!}){\downarrow_{\{q,r\}}}){\downarrow_{\not!?}} = ((w' \cdot x'){\downarrow_?}){\downarrow_{\not!?}}\Big)$, which must exist since $(w' \cdot x')\mathcal{L}^{\emptyset}(A_q)$ by assumption.
Then, we apply Lemma 2.3.2 to obtain execution e for w''. By construction, p and q perform no actions here and q gets sent the necessary sends to perform $w' \cdot x'$, while p gets sent nothing, and p and q send and receive nothing in e. We could append $w' \cdot x'$ directly to e, this is possible since w'' provides all sends for $w' \cdot x'$ and $e{\downarrow_r} = w''$. Instead, we first want to "mix" w' and w to construct a certain appendix for e. So, we obtain $w_{mix} = (\text{mix\_pair}\ (w'', w))$ and in turn $e' = e \cdot w_{mix} \cdot x'$: this is possible, since e provides all the necessary sends for $w' \cdot x'$, and w_{mix} is constructed s.t. each receive

of w has the corresponding send of q directly before it (and by assumption w' and w have matching sends and receives). Furthermore, x' can be performed at the end, since $w_{mix}{\downarrow_q} = w'$ and thus q is still in the position to perform x' (peers cannot block each other's actions directly, so whether w is performed inbetween e and x' or not doesn't hinder q). Since the network is synchronizable, obtain the synchronous execution e'' with the same trace as e'. By construction of e', e'' and e' projected to only actions between p and q, before x' (i.e. sends from q to p and receives of these sends) are equal. Since w_{mix} performs each send of q directly before the receive of p (i.e., simulating the synchronous execution between these two peers), e'' must also contain w in its execution, otherwise a different trace is performed. By construction, then $e''{\downarrow_p} = w \cdot x$ for some x, s.t. $wx \in \mathcal{L}^{\emptyset}(A_p)$ and $(((w'x'){\downarrow_!}){\downarrow_{\{p,q\}}}{\downarrow_{!?\not!}} = ((wx){\downarrow_?}){\downarrow_{!?\not!}}$. In e'', the projection on q may not be $w'x'$ (if w' contains any receives, the receives will get pulled further left, where the respective sends to q are located by construction of e). The important part is, however, that $(e'{\downarrow_q}){\downarrow_!} = (e{\downarrow_q}){\downarrow_!}$, i.e. the sends of this projection are the same as the sends in $w'x'$, which is exactly what is needed for p (p's actions are not influenced by what receives or sends to different children q does). Finally, we have found a matching x for our arbitrary x' and thus shown the subset condition also for the case where q is not the root.

2. By definition, $\mathcal{L}^{\emptyset}(A_p) \subseteq \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$. Assume $v' \in \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$. We have to show that $v' \in \mathcal{L}^{\emptyset}(A_p)$. Since $v' \in \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$, then by definition of the shuffled language, there is some v such that $v \in \mathcal{L}^{\emptyset}(A_p)$ and $v' \sqcup\!\sqcup_? v$. We prove $v' \in \mathcal{L}^{\emptyset}(A_p)$ by induction over the shuffle $v' \sqcup\!\sqcup_? v$. There are three cases : either no shuffle occurred, exactly one occurred, or v' was received through transitivity of the shuffle relation from Definition 2.2.9. The first and last case are both trivial, and Isabelle was able to prove them directly, so the handwritten proof of these will be left as a small exercise to the reader. We will now detail the only complicated case, where exactly one shuffle occurred, so v and v' can be decomposed into some $(xs \cdot ?b^{q \to p} \cdot !a^{p \to x} \cdot ys) = v' \in \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ and $(xs \cdot !a^{p \to x} \cdot ?b^{q \to p} \cdot ys) = v \in \mathcal{L}^{\emptyset}(A_p)$.
We can infer that p is a node in the tree with some parent q, as the root cannot receive messages, ruling out any non-trivial shuffles. Then, obtain $vq \in \mathcal{L}^{\emptyset}(A_q)$, s.t. $((vq{\downarrow_!}){\downarrow_{\{p,q\}}}{\downarrow_{!?\not!}} = (v{\downarrow_?}){\downarrow_{!?\not!}}$, by applying the definition of the influenced language. Similar to the previous proof part, we now do a case distinction over whether q is the root or some node with parent r.
Case : q is the root.
Then vq has no receives and consequently vq is a valid execution in $\mathsf{E}(N_{\texttt{mbox}})$. Next, we construct $w_{mix} = (\text{mix\_triple}\ (vq, v', v))$, i.e. mix vq with v, which then is also an execution $\in \mathsf{E}(N_{\texttt{mbox}})$ by construction, as explained beneath Definition 2.3.9. Using synchronizability, we then obtain the synchronous execution $e' \in \mathsf{E}(N_{\texttt{sync}})$ with the same trace as w_{mix}. By construction, $e'{\downarrow_p} = v'$ and consequently $v' \in \mathcal{L}^{\emptyset}(A_p)$, otherwise e' is not an execution.
Case : q is a node with parent r.
Then obtain $vr \in \mathcal{L}^{\emptyset}(A_r)$ which provides exactly matching sends for vq using the defintion of the influenced language. Then apply Lemma 2.3.2 to vr and receive execution $e \in \mathsf{E}(N_{\texttt{mbox}})$ which contains no actions of p or q, respectively, by construction.

Next, obtain $w_{mix} = (\text{mix\_triple}\ (vq, v', v))$. Then, $e \cdot w_{mix} \in \mathsf{E}(N_{\texttt{mbox}})$, since both p and q are in their initial states after e, and e provides all sends for q in the correct order and w_{mix} keeps all sends of q before the receives of p. Now we again use synchronizability to receive $e' \in \mathsf{E}(N_{\texttt{sync}})$ with $e'{\downarrow_!} = (e \cdot w_{mix}){\downarrow_!}$. By construction of w_{mix} and e', we know that $e'{\downarrow_p} = v'$. Consequently, $v' \in \mathcal{L}^{\emptyset}(A_p)$ must hold, or e' is no execution.

In all cases and for an arbitrary v', we have shown that $v' \in \mathcal{L}^{\emptyset}(A_p)$, so we can conclude that $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$.

$\Leftarrow$ Assume that the subset condition (as in Definition 2.3.7) and $\mathcal{L}^{\emptyset}(A_p) = \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(p)$ hold for all $p, q \in \mathbb{P}$ with $\mathbb{P}^p_{send} = \{q\}$. We have to show that $\mathsf{T}(N_{\texttt{mbox}}) = \mathsf{T}(N_{\texttt{sync}})$.

$w \in \mathsf{T}(N_{\texttt{mbox}})$ **:** Let w' be the word obtained from w by adding the matching receive action directly after every send action. ‡‡ We show that $w' \in \mathsf{E}(N_{\texttt{mbox}})$, by an induction on the length of w.

Base Case : If $w = !a^{q\to p}$, then $w' = !a^{q\to p}?a^{q\to p}$. Since $w \in \mathsf{T}(N_{\texttt{mbox}})$, A_q is able to send $!a^{q\to p}$ in its initial state within the system $N_{\texttt{mbox}}$. Then $!a^{q\to p} \in \mathcal{L}^{\emptyset}_{!}(A_q)$. We apply the subset condition to $\epsilon \in \mathcal{L}^{\emptyset}(A_p)$ and $\epsilon \in \mathcal{L}^{\emptyset}(A_q)$ and the suffix $!a^{q\to p}$. Since $\epsilon \cdot !a^{q\to p} = !a^{q\to p} \in \mathcal{L}^{\emptyset}(A_q)$, p must then be able to perform some word $x \in \mathcal{L}^{\emptyset}(A_q)$ from its initial state, which has $?a^{q\to p}$ as the only receive. We only know about x that it contains exactly this input, but it may also contain sends. This is undesired, given that we want the execution where p only does this receive, so we fully shuffle x : We know that $?a^{q\to p} = x{\downarrow_?}$; let $ys = x{\downarrow_!}$ be the possibly existing sends of x, then obtain $(?a^{q\to p} \cdot ys) \sqcup\!\sqcup_? x$, by repeatedly shuffling $?a^{q\to p}$ further left until it cannot be shuffled anymore. Then, $?a^{q\to p} \cdot ys \in \mathcal{L}^{\emptyset}(A_q)$ by the shuffled language condition. Thus, $?a^{q\to p} \in \mathcal{L}^{\emptyset}(A_q)$, as all states of our network are accepting. Then $w' \in \mathsf{E}(N_{\texttt{mbox}})$.

Inductive Step : If $w = v \cdot !a^{q\to p}$ with $v \cdot !a^{q\to p} \in \mathsf{T}(N_{\texttt{mbox}})$, then $w' = v' \cdot !a^{q\to p} \cdot ?a^{q\to p}$. By induction, $v' \in \mathsf{E}(N_{\texttt{mbox}})$. To show : $w' \in \mathsf{E}(N_{\texttt{mbox}})$. This can be decomposed into the following subgoals :

(1) $(v'{\downarrow_q}) \cdot !a^{q\to p} \in \mathcal{L}^{\emptyset}(A_q)$,

(2) from this $v' \cdot !a^{q\to p} \in \mathsf{E}(N_{\texttt{mbox}})$.

(3) Then $(v' \cdot !a^{q\to p}){\downarrow_p} \cdot ?a^{q\to p} \in \mathcal{L}^{\emptyset}(A_p)$,

(4) hence $w' \in E_{mbox}$.

—

Proof of (1).

Case 1 : q is the root. Then $v'{\downarrow_q} = v{\downarrow_q}$, otherwise q would be receiving messages despite being the root. Since w is a valid trace by assumption and consequently q must be able to perform all its actions in w, we can conclude $w{\downarrow_q} = w'{\downarrow_q} = (v{\downarrow_q}) \cdot !a^{q\to p} \in \mathcal{L}^{\emptyset}(A_q)$.

Case 2 : q is a node with parent r. Since w is a valid trace, there exists some $wq \in \mathcal{L}^{\emptyset}(A_q)$ and $e \in \mathsf{E}(N_{\texttt{mbox}})$, s.t. $e{\downarrow_!} = w$ and $e{\downarrow_q} = wq$. By construction, $(wq){\downarrow_!} = (v'{\downarrow_q} \cdot !a^{q\to p}){\downarrow_!}$ and in general $(v'{\downarrow_q} \cdot !a^{q\to p}){\downarrow_?} = (v'{\downarrow_q}){\downarrow_?}$, since $!a^{q\to p}$ is not an input. We infer that $(wq){\downarrow_?}$ is a prefix of $(v'{\downarrow_q}){\downarrow_?}$. If we assume the contrary, then the ordering of receives is different in v' and wq, but since v' receives each

‡‡. A function in Isabelle that does this named *add_matching_recvs*, can be found in the Appendix .1.

send directly afterwards, that would mean that in wq, q receives something that is not in the first position of the buffer, contradicting the assumption that e is an execution.
Moreover, $((wq)\downarrow_?)\downarrow_{\not!\not?}$ must be a prefix of $((w\downarrow_!)\downarrow_{\{r,q\}})\downarrow_{\not!\not?}$, since $((w\downarrow_!)\downarrow_{\{r,q\}})\downarrow_{\not!\not?} = ((w'\downarrow_?)\downarrow_{\{r,q\}})\downarrow_{\not!\not?}$ by construction of w' and we just showed that $(wq)\downarrow_?$ is a prefix of $(v'\downarrow_q)\downarrow_?$. Since $w'\downarrow_? = w = e\downarrow_?$, r sends the same messages to q in both e and w', but wq may receive only a prefix of them.
As a reminder, q is a node with parent r, and $wq \in \mathcal{L}^{\emptyset}(A_q)$, so there exists $wr' \cdot x' \in \mathcal{L}^{\emptyset}(A_r)$ such that

$$((wr' \cdot x')\downarrow_!)\downarrow_{\{r,q\}} = (w\downarrow_!)\downarrow_{\{r,q\}} \text{ and } ((wq)\downarrow_?)\downarrow_{\not!\not?} = (((wr')\downarrow_!)\downarrow_{\{r,q\}})\downarrow_{\not!\not?}.$$

Intuitively, we obtain all sends from r to q, and decompose these into wr' and x', so that wr' provides exactly the sends that wq receives from q (since wq may not receive all sends).
By the subset condition, since r can perform wr' with the x', q must also be able to perform some x after wq s.t.

$$wq \cdot x \in \mathcal{L}^{\emptyset}(A_q) \text{ and } ((wq \cdot x)\downarrow_?)\downarrow_{\not!\not?} = (((wr' \cdot x')\downarrow_!)\downarrow_{\{q,r\}})\downarrow_{\not!\not?}.$$

We only know about x that it contains exactly the needed receives, but it may also contain sends. This is undesired, so we fully shuffle it : let $xs = x\downarrow_?$ and $ys = x\downarrow_!$ be the sends/receives of x, then $(xs \cdot ys) \sqcup\!\sqcup_? x$. Then we prepend wq to this shuffle, to obtain $wq \cdot xs \cdot ys \sqcup\!\sqcup_? wq \cdot xs$. Then, $wq \cdot xs \cdot ys \in \mathcal{L}^{\emptyset}(A_q)$ by the shuffled language condition. Thus, $wq \cdot xs \in \mathcal{L}^{\emptyset}(A_q)$ with projections matching those of $w'\downarrow_q$, since all states are accepting and thus any prefix of a valid word is also valid.
We can now infer that $((v'\downarrow_q)\cdot !a^{q\to p}) \sqcup\!\sqcup_? (wq \cdot xs)$ must hold ; otherwise some send/receive ordering would contradict the trace :

> Let $!x$ be an arbitrary send and $?y$ a receive. Write $!x <?y$ in a word w to denote that $!x$ occurs earlier (further left) in w than $?y$. Then there is some output–input pair $!x, ?y$ such that $!x <?y$ in $((v'\downarrow_q)\cdot !a)$ but $?y <!x$ in $(wq\cdot xs)$, meaning that a conflicting shuffle from wq to $(v'\downarrow_q\cdot a)$ occurred. By construction of w', we have $!x <?y$ in w, but $wq \cdot xs$ has $?y <!x$, requiring y to be sent (and to be received) before $!x$ can be sent ; hence $wq \cdot xs$ cannot be part of an execution that produces trace w. However, $(e \cdot xs)$ is a valid execution (since xs only receives elements that are already in q's buffer after e and q cannot be blocked by any other peer at the end of e), and $(e \cdot xs)\downarrow_! = e\downarrow_! = w = w'\downarrow_!$ by our assumptions. But $(e \cdot xs)\downarrow_!$ has $?y <!x$ while w has $!x <?y$, yielding a different trace than assumed $\lightning$.

So $((v'\downarrow_q)\cdot !a^{q\to p}) \sqcup\!\sqcup_? (wq \cdot xs)$ holds and in turn by the shuffled language condition, $(v'\downarrow_q\cdot !a^{q\to p}) \in \mathcal{L}^{\emptyset}_{\sqcup\!\sqcup}(q) = \mathcal{L}^{\emptyset}(A_q)$ holds as well. This completes (1).
From this, (2) follows immediately, since sends cannot be blocked by other peers and we have shown that q can perform $(v'\downarrow_q\cdot !a^{q\to p}) \in \mathcal{L}^{\emptyset}(A_q)$, hence

Any word can be "fully" shuffled, until all receives come first, followed by all sends of the given word. We have proven this in Isabelle, but it follows from Definition 2.2.9 relatively directly.

$v' \cdot !a^{q \to p} \in \mathsf{E}(N_{\texttt{mbox}})$. Using the subset condition and shuffling out the possible sends analogously to before , (3) also follows. Finally, (4) holds because p's buffer cannot contain $a^{q \to p}$, and receives also cannot be blocked by other peers. This concludes the inductive step, we have shown that $w' \in E_{mbox}$. Since $w'{\downarrow_!} = w$, then $w \in \mathsf{E}(N_{\texttt{sync}}) = \mathsf{T}(N_{\texttt{sync}})$.

$w \in \mathsf{T}(N_{\texttt{sync}})$ **:** For every output in w, $N_{\texttt{sync}}$ was able to send the respective message and directly receive it. Let w' be the word obtained from w by adding the matching receive action directly after every send action. Then $N_{\texttt{mbox}}$ can simulate the run of w in $N_{\texttt{sync}}$ by sending every message first to the mailbox of the receiver and then receiving this message. Then $w' \in \mathsf{E}(N_{\texttt{mbox}})$ and, thus, $w = w'{\downarrow_!} \in \mathsf{T}(N_{\texttt{mbox}})$.

□

As a note, the last case ("$w \in \mathsf{T}(N_{\texttt{sync}})$") is directly cited from (Di Giusto et al., 2024), but we have omitted quotation marks for visual clarity. All other cases have changed considerably through our work.

2.4 Conclusion

This work has provided another case study of why verification is so instrumental. We have verified (except for some remaining obvious helper lemmas), that synchronizability for trees is in fact decidable, as was the claim in (Di Giusto et al., 2024). Through the mechanization of the proposed algorithm and formalization into Isabelle, several gaps in the original theorem came to light and were then corrected in multiple iterations. We have rewritten the theorem and thus the algorithm to determine synchronizability of some tree, and provided a largely complete Isabelle formalization for this as well.

— iteratively found places in proofs/ definitions where formalization was not correct or incomplete (steps missing, etc.)

— intuition correct but formally not

— now the intuition is in tune with the results

— rewrote the entire proof of theorem 4.5 (except 2.<-) because the conditions changed and it was necessary

— learned Isabelle

— mostly verfified that synchronizability for mailbox with tree topology is decidable (i.e. mostly verified the claim from express)

2.5 Perspectives

The perspectives are twofold, we will first detail those concerning our Isabelle formalization, and then those concerning the results of this work in general.

As previously mentioned, there still remain some small lemmas to be proven to fully verify our claims. However, all of these are intuitive and trivial to do on paper, but with the current

Same construction as in the base case or with $wr'x'$ and x, to receive xs.

We use the same reasoning as in the steps with wr' and x' where we received xs.

formalization in Isabelle and the time constraints, this was not yet possible for us to complete. Several missing lemmas concern the prefixes of words in a language also being in that language. Since we focus solely on automata where each state is final, this is trivially true. In other words, if we know that an automaton accepts a word $w = u \cdot v$, then we can also stop after reading u and still have an accepting run because each step starts and ends in an accepting state by assumption. The rest of the remaining lemmas are similar in complexity, thus proving them will take some additional time, mostly because of the current state of the Isabelle formalization.

As alluded to, the current Isabelle formalization is not ideal for proving all needed lemmas. The issues lies in the encoding of the tree topology, which does not directly yield any information about the peers and their messages (and consequently whether they are the root or a node). This adds complexity to any proof in need of these relations, which is why some intuitive lemmas are still unproven. In other words, from the tree topology as it is currently defined in our Isabelle formalization, we cannot directly infer any information about the peers, especially concerning their messages or place in the topology. This formalization was present before this work began, thus, we continued with it and built the rest of the Isabelle framework around it. Consequently, it is now difficult to adapt this tree topology definition, without affecting the rest of the code. We have the following dilemma : as explained in the previous paragraph, not all lemmas can be easily proven because the tree topology's formalization in Isabelle is not ideal, however, changing this formalization means having to - in the worst case - rewrite or reprove parts of definitions, lemmas, etc. where the tree topology currently occurs. In short, it would be best to formalize a stronger tree topology in Isabelle which can then be used for the missing proof steps in some of the unproven lemmas.

With or without an improved Isabelle formalization, another avenue would be to formalize the counterexamples we have found in Isabelle. We have encountered similar difficulties as in the process of proving the remaining helper lemmas, which is why the formalization was not completed. However, formalizing the counterexamples as well would contribute to the goal of fully verifying this entire project.

Lastly, this result could also possibly be extended to both Peer-to-Peer networks as well as networks with different variants of tree topologies, as was previously mentioned in the source paper. To reiterate their points, networks consisting of multiple or reversed trees may be able to be investigated in terms of their synchronizability, with a similar approach to our end result here. Since we have found a procedure to decide the synchronizability of one tree, one could apply this for each tree of a multitree network. For both of these variants, our requirement of having pairwise unique paths between nodes is still satisfied. To extend this procedure to reversed trees, some of the procedure would likely need to be reversed, to account for the change in properties. Additionally, the authors claim that the set of executions of a network with tree topology should be equal for Mailbox and Peer-to-Peer systems. This remains to be formally proven and hopefully verified, however. In short, we have found the synchronizability problem to be decidable for tree topologies, and propose a similar approach for investigating the decidability of this problem in related contexts. (Di Giusto et al., 2024)

We are confident that we have achieved a good foundation of the last perspective proposed in (Di Giusto et al., 2024), namely the mechanisation of the paper's proofs in Isabelle, despite its challenging nature. With this foundation and the future prospects detailed above, we hope to inspire the reader to continue improving this project or to delve into their own verification endeavors.

The entirety of this work has been an excellent case study as to why proper verification is so important ; regardless of how intuitive the reasoning may seem, formal correctness should always be respected and ensured.

Références

Di Giusto, C., Laversa, L., & Peters, K. (2024). Synchronisability in mailbox communication. *arXiv preprint arXiv :2411.14580*.

Finkel, A., & Lozes, E. (2017). Synchronizability of Communicating Finite State Machines is not Decidable. In I. Chatzigiannakis, P. Indyk, F. Kuhn, & A. Muscholl (Eds.), *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)* (Vol. 80, pp. 122 :1–122 :14). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Consulté le 2021-08-06, sur http://drops.dagstuhl.de/opus/volltexte/2017/7402 (ISSN : 1868-8969) doi: 10.4230/LIPIcs.ICALP.2017.122

Nipkow, T. (2023). *Concrete semantics : With isabelle/hol.* Consulté sur http://www.concrete-semantics.org/slides-isabelle.pdf

Nipkow, T., & Klein, G. (2023). *Concrete semantics : With isabelle/hol*. Springer International Publishing. Consulté sur http://www.concrete-semantics.org/concrete-semantics.pdf

Nipkow, T., Paulson, L. C., & Wenzel, M. (s. d.). A proof assistant for higher-order logic. *13.03.2025*. Consulté sur https://isabelle.in.tum.de/doc/tutorial.pdf

Wenzel, M. (s. d.). The isabelle/isar reference manual. *13.03.2025*. Consulté sur https://isabelle.in.tum.de/doc/isar-ref.pdf

Annexes

.1 Isabelle Proof for Synchronizability for Trees

.1.1 First Mix Construction in Isabelle

fun mix_pair :: "('information, 'peer) action word ⇒ ('information, 'peer) action word ⇒ ('information, 'peer) action word ⇒ ('information, 'peer) action word" **where**
"mix_pair [] [] acc = acc" |
"mix_pair (a # w') [] acc = mix_pair w' [] (a # acc)" |
"mix_pair [] (a # w) acc = mix_pair [] w (a # acc)" |
"mix_pair (a # w') (b # w) acc = (if a = !⟨get_message b⟩
then (if b = ?⟨get_message a⟩ then mix_pair w' w (a # b # acc) else mix_pair (a # w') w (b # acc))
else mix_pair w' (b # w) (a # acc))"

.1.2 Second Mix Construction in Isabelle

inductive mix_shuf :: "('information, 'peer) action word ⇒ ('information, 'peer) action word ⇒ ('information, 'peer) action word ⇒ ('information, 'peer) action word ⇒ bool" **where**
mix_shuf_constr: "⟦vq↓!↓$_{-p,q''}$↓!? = v↓?↓!?; v' ∈ $\mathcal{L}^*_{⊔⊔}$(p); v' ⊔⊔? v; v ∈ $\mathcal{L}^*$(p); vq ∈ $\mathcal{L}^*$(q);
vq = (as ▪ a_send # bs); v = xs ▪ b # a_recv # ys; get_message a_recv = get_message a_send; is_input a_recv; is_output a_send; is_output b⟧
⟹ mix_shuf vq v v' ((mix_pair as xs []) ▪ a_send # b # a_recv # (mix_pair bs ys []))"

.1.3 add_matching_recvs

fun add_matching_recvs :: "('information, 'peer) action word ⇒ ('information, 'peer) action word" **where**
"add_matching_recvs [] = []" |
"add_matching_recvs (a # w) = (if is_output a
then a # (?⟨get_message a⟩) # add_matching_recvs w
else a # add_matching_recvs w)"

.1.4 Main Theorem in Isabelle

theorem synchronisability_for_trees:
assumes "tree_topology"
shows "is_synchronisable ⟷ ((∀ p ∈ $\mathcal{P}$. ∀ q ∈ $\mathcal{P}$. ((is_parent_of p q) ⟶ ((subset_condition p q) ∧ (($\mathcal{L}^*$(p)) = ($\mathcal{L}^*_{⊔⊔}$(p)))))))" (**is** "?L ⟷ ?R")

proof

assume "?L"

```
    show "?R"
    proof clarify
     fix p q
     assume q_parent: "is_parent_of p q"
     have sync_def: "𝒯_None⇂! = ℒ₀" using ‹?L› by simp
     show "subset_condition p q ∧ ℒ* p = ℒ*_⊔⊔ p"
     proof (rule conjI)

      show "subset_condition p q" unfolding subset_condition_def
      proof auto

       fix w w' x'
       assume w_Lp: "is_in_infl_lang p w"
        and w'_Lq: "is_in_infl_lang q w'"
        and w'_w_match: "filter (λx. is_output x ∧ (get_object x = q ∧ get_actor x = p
              ∨ get_object x = p ∧ get_actor x = q)) w'↓!? = w↓?↓!?"
        and w'x'_Lq: "is_in_infl_lang q (w' ▪ x')"
      then show "∃ wa. filter (λx. is_output x ∧ (get_object x = q ∧ get_actor x = p ∨ get_object
x = p ∧
            get_actor x = q)) x'↓!? = wa↓!? ∧ (∃ x. wa = x↓? ∧ is_in_infl_lang p (w ▪ x))"
       using w_Lp w'_Lq w'_w_match w'x'_Lq
      proof (cases "is_root q")
       case True
       then have "(w' ▪ x') ∈ ℒ q" using w'x'_Lq w_in_infl_lang by auto


       then have "(w' ▪ x') ∈ 𝒯_None" using root_lang_is_mbox True by blast

       have "w'↓!↓_{-p,q''}↓!? = w↓?↓!?" using w'_w_match by force


       let ?mix = "(mix_pair w' w [])"
       have "?mix ▪ x' ∈ 𝒯_None" sorry
        then obtain t where "t ∈ ℒ₀ ∧ t ∈ 𝒯_None⇂! ∧ t = (?mix ▪ x')↓!" using sync_def by
fastforce
        then obtain xc where t_sync_run : "sync_run 𝒞_ℐ₀ t xc" using SyncTraces.simps by
auto

          then have "∃ xcm. mbox_run 𝒞_ℐm None (add_matching_recvs t) xcm" using
empty_sync_run_to_mbox_run sync_run_to_mbox_run by blast

       then have sync_exec: "(add_matching_recvs t) ∈ 𝒯_None" using MboxTraces.intros by
auto

       then have "∃ x. (add_matching_recvs t)↓_p = w ▪ x" sorry
       then obtain x where x_def: "(add_matching_recvs t)↓_p = w ▪ x" by blast
```

then have w'x'_wx_match: "(w' ▪ x')$\downarrow_!\downarrow_{-p,q''}\downarrow_{!?}$ = (w ▪ x)$\downarrow_?\downarrow_{!?}$" **sorry**
have "(w ▪ x) ∈ $\mathcal{L}^*$ p" **using** sync_exec x_def **by** (metis mbox_exec_to_infl_peer_word)
have "∃ wa. x'$\downarrow_!\downarrow_{-p,q''}\downarrow_{!?}$ = wa$\downarrow_{!?}$ ∧ (∃ x. wa = x$\downarrow_?$ ∧ is_in_infl_lang p (w ▪ x))" **using** ‹w ▪ x ∈ $\mathcal{L}^*$ p› ‹w'$\downarrow_!\downarrow_{-p,q''}\downarrow_{!?}$ = w$\downarrow_?\downarrow_{!?}$› w'x'_wx_match **by** auto
then show ?thesis **by** simp
next
case False
then have "is_node q" **by** (metis NetworkOfCA.root_or_node NetworkOfCA_axioms assms)
then obtain r **where** qr: "is_parent_of q r" **by** (metis False UNIV_I path_from_root.simps path_to_root_exists paths_eq)

have "(w' ▪ x') ∈ $\mathcal{L}^*$ q" **by** (simp add: w'x'_Lq)
then have "∃ w''. w'' ∈ $\mathcal{L}^*$(r) ∧ (((w' ▪ x')$\downarrow_?$)$\downarrow_{!?}$) = (((w''$\downarrow_{-q,r''}$)$\downarrow_!$)$\downarrow_{!?}$)"
using infl_parent_child_matching_ws[of "(w' ▪ x')" q r] **using** qr **by** blast
then obtain w'' **where** w''_w'_match: "w''$\downarrow_!\downarrow_{-q,r''}\downarrow_{!?}$ = (w' ▪ x')$\downarrow_?\downarrow_{!?}$" **and** w''_def: "w'' ∈ $\mathcal{L}^*$ r" **by** (metis (no_types, lifting) filter_pair_commutative)

have "∃ e. (e ∈ $\mathcal{T}_{None}$ ∧ e$\downarrow_r$ = w'' ∧ ((is_parent_of q r) ⟶ e$\downarrow_q$ = ε))" **using** lemma4_4[of
w'' r q] **using** ‹w'' ∈ $\mathcal{L}^*$ r› assms **by** blast
then obtain e **where** e_def: "e ∈ $\mathcal{T}_{None}$" **and** e_proj_r: "e$\downarrow_r$ = w''"
and e_proj_q: "e$\downarrow_q$ = ε" **using** qr **by** blast

let ?mix = "(mix_pair w' w [])"

have "e ▪ ?mix ▪ x' ∈ $\mathcal{T}_{None}$" **sorry**

then obtain t **where** "t ∈ $\mathcal{L}_\mathbf{0}$ ∧ t ∈ $\mathcal{T}_{None}\downarrow_!$ ∧ t = (e ▪ ?mix ▪ x')$\downarrow_!$" **using** sync_def **by** fastforce
then obtain xc **where** t_sync_run : "sync_run $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ t xc" **using** SyncTraces.simps **by** auto

then have "∃ xcm. mbox_run $\mathcal{C}_{\mathcal{I}\mathbb{m}}$ None (add_matching_recvs t) xcm" **using** empty_sync_run_to_mbox_run sync_run_to_mbox_run **by** blast

then have sync_exec: "(add_matching_recvs t) ∈ $\mathcal{T}_{None}$" **using** MboxTraces.intros **by** auto

then have "∃ x. (add_matching_recvs t)$\downarrow_p$ = w ▪ x" **sorry**
then obtain x **where** x_def: "(add_matching_recvs t)$\downarrow_p$ = w ▪ x" **by** blast
then have w'x'_wx_match: "(w' ▪ x')$\downarrow_!\downarrow_{-p,q''}\downarrow_{!?}$ = (w ▪ x)$\downarrow_?\downarrow_{!?}$" **sorry**

have "(w ▪ x) ∈ $\mathcal{L}^*$ p" **using** sync_exec x_def **by** (metis mbox_exec_to_infl_peer_word)
have "w'$\downarrow_!\downarrow_{-p,q''}\downarrow_{!?}$ = w$\downarrow_?\downarrow_{!?}$" **using** w'_w_match **by** force

```
          have "∃ wa. x'↓!↓₋p,q″↓!? =  wa↓!? ∧  (∃ x. wa = x↓? ∧ is_in_infl_lang p (w ▪ x))"  using
‹w ▪ x ∈ 𝓛* p› ‹w'↓!↓₋p,q″↓!? = w↓?↓!?› w'x'_wx_match by auto
           then show ?thesis by simp
         qed
        qed

        show "𝓛*(p) = 𝓛*⧢⧢(p)"
        proof

         show "𝓛*(p) ⊆ 𝓛*⧢⧢(p)" using language_shuffle_subset by auto

         show "𝓛*⧢⧢(p) ⊆ 𝓛*(p)"
         proof
          fix v'

          assume "v' ∈ 𝓛*⧢⧢(p)"
            then obtain v where v_orig: "v' ⧢⧢? v" and orig_in_L: "v ∈ 𝓛*(p)" using shuf-
fled_infl_lang_impl_valid_shuffle by auto
          then show "v' ∈ 𝓛*(p)"
          proof (induct v v')
           case (refl w)
           then show ?case by simp
          next
           case (swap b a w xs ys)

           then have "∃ vq.  vq ∈ 𝓛*(q) ∧ ((w↓?)↓!?) = (((vq↓₋p,q″)↓!)↓!?)"
            using infl_parent_child_matching_ws[of w p q] orig_in_L q_parent by blast
            then obtain vq where vq_v_match: "((w↓?)↓!?) = (((vq↓₋p,q″)↓!)↓!?)" and vq_def:
"vq ∈ 𝓛* q" by auto
           have lem4_4_prems: "tree_topology ∧ w ∈ 𝓛*(p) ∧ p ∈ 𝒫" using assms swap.prems
by auto
           then show ?case using assms swap vq_v_match vq_def lem4_4_prems
           proof (cases "is_root q")
            case True
            have "vq ∈ ℒ q" using vq_def w_in_infl_lang by auto
            then have "vq ∈ 𝒯_None" using root_lang_is_mbox True by simp

            let ?w' = "xs ▪ a # b # ys"
            have "∃ acc. mix_shuf vq v v' acc" sorry
            then obtain mix where "mix_shuf vq v v' mix" by blast
            let ?mix = "mix"
            have "?mix ∈ 𝒯_None" sorry
              then obtain t where "t ∈ ℒ₀ ∧ t ∈ 𝒯_None↓! ∧ t = (?mix)↓!" using sync_def by
fastforce
            then obtain xc where t_sync_run : "sync_run 𝒞_ℐ₀ t xc" using SyncTraces.simps by
auto
```

then have "∃xcm. mbox_run $\mathcal{C}_{\mathcal{I}\mathtt{m}}$ None (add_matching_recvs t) xcm" **using** empty_sync_run_to_mbox_run sync_run_to_mbox_run **by** blast

then have sync_exec: "(add_matching_recvs t) ∈ $\mathcal{T}_{None}$" **using** MboxTraces.intros **by** auto

then have "(add_matching_recvs t)↓$_p$ = ?w'" **sorry**
then have "?w' ∈ $\mathcal{L}^*$ p" **using** sync_exec mbox_exec_to_infl_peer_word **by** metis
then show ?thesis **by** simp
next
case False
then have "is_node q" **by** (metis NetworkOfCA.root_or_node NetworkOfCA_axioms assms)
then obtain r **where** qr: "is_parent_of q r" **by** (metis False UNIV_I path_from_root.simps path_to_root_exists paths_eq)
then have "∃vr. vr ∈ $\mathcal{L}^*$(r) ∧ ((vq↓$_?$)↓$_{!?}$) = (((vr↓$_{-q,r''}$)↓$_!$)↓$_{!?}$)"
using infl_parent_child_matching_ws[of vq q r] orig_in_L vq_def **by** blast

then obtain vr **where** vr_def: "vr ∈ $\mathcal{L}^*$(r)" **and** vr_vq_match: "((vq↓$_?$)↓$_{!?}$) = (((vr↓$_{-q,r''}$)↓$_!$)↓$_{!?}$)" **by** blast

have "∃ e. (e ∈ $\mathcal{T}_{None}$ ∧ e↓$_r$ = vr ∧ ((is_parent_of q r) ⟶ e↓$_q$ = ε))" **using** lemma4_4[of
vr r q] **using** ‹vr ∈ $\mathcal{L}^*$ r› assms **by** blast
then obtain e **where** e_def: "e ∈ $\mathcal{T}_{None}$" **and** e_proj_r: "e↓$_r$ = vr"
and e_proj_q: "e↓$_q$ = ε" **using** qr **by** blast

let ?w' = "xs ▪ a # b # ys"
have "∃ acc. mix_shuf vq v v' acc" **sorry**
then obtain mix **where** "mix_shuf vq v v' mix" **by** blast
let ?mix = "mix"

have "e ▪ ?mix ∈ $\mathcal{T}_{None}$" **sorry**

then obtain t **where** "t ∈ $\mathcal{L}_{\mathbf{0}}$ ∧ t ∈ $\mathcal{T}_{None}$⌊$_!$ ∧ t = (e ▪ ?mix)↓$_!$" **using** sync_def **by** fastforce
then obtain xc **where** t_sync_run : "sync_run $\mathcal{C}_{\mathcal{I}\mathbf{0}}$ t xc" **using** SyncTraces.simps **by** auto

then have "∃xcm. mbox_run $\mathcal{C}_{\mathcal{I}\mathtt{m}}$ None (add_matching_recvs t) xcm" **using** empty_sync_run_to_mbox_run sync_run_to_mbox_run **by** blast

then have sync_exec: "(add_matching_recvs t) ∈ $\mathcal{T}_{None}$" **using** MboxTraces.intros **by** auto

```
            then have "(add_matching_recvs t)↓p = ?w'" sorry
            then have "?w' ∈ L* p" using sync_exec mbox_exec_to_infl_peer_word by metis
            then show ?thesis by simp
           qed
          next
           case (trans w w' w'')
           then show ?case by simp
          qed
         qed
        qed
       qed

    next

      assume "?R"
      show "?L" — show that TMbox = TSync, i.e. L> = (i.e. the sends are equal
      proof auto — cases : w in TMbox, w in TSync
       fix w
       show "w ∈ T_None ⟹ w↓! ∈ L_0"
       proof -
        assume "w ∈ T_None"
        then have "(w↓!) ∈ T_None↓!" by blast

        then have match_exec: "add_matching_recvs (w↓!) ∈ T_None"
          using mbox_trace_with_matching_recvs_is_mbox_exec ‹∀p∈P. ∀q∈P. is_parent_of p
q ⟶ subset_condition p q ∧ L* p = L*⊔⊔ p› assms theorem_rightside_def
         by blast
         then obtain xcm where "mbox_run C_Im None (add_matching_recvs (w↓!)) xcm" by
(metis MboxTraces.cases)
            then show "(w↓!) ∈ L_0" using SyncTraces.simps ‹w↓! ∈ T_None↓!› mat-
ched_mbox_run_to_sync_run by blast
       qed
      next — w in TSync –> show that w' (= w with recvs added) is in EMbox
       fix w
       show "w ∈ L_0 ⟹ ∃w'. w = w'↓! ∧ w' ∈ T_None"
       proof -
        assume "w ∈ L_0"
          — For every output in w, Nsync was able to send the respective message and directly
receive it
        then have "w = w↓!" by (metis SyncTraces.cases run_produces_no_inputs(1))
        then obtain xc where w_sync_run : "sync_run C_I0 w xc" using SyncTraces.simps ‹w ∈
L_0› by auto
        then have "w ∈ L_∞" using ‹w ∈ L_0› mbox_sync_lang_unbounded_inclusion by blast
        obtain w' where "w' = add_matching_recvs w" by simp
```

— then Nmbox can simulate the run of w in Nsync by sending every mes- sage first to the mailbox of the receiver and then receiving this message
then show ?thesis
proof (cases "xc = []") — this case distinction isn't in the paper but i need it here to properly get the simulated run
case True
then have "mbox_run $\mathcal{C}_{\mathcal{I}\mathtt{m}}$ None (w') []" **using** ‹w' = add_matching_recvs w› empty_sync_run_to_mbox_run w_sync_run **by** auto
then show ?thesis **using** ‹w ∈ $\mathcal{T}_{None}\lfloor_{!}$› **by** blast
next
case False
then obtain xcm **where** sim_run: "mbox_run $\mathcal{C}_{\mathcal{I}\mathtt{m}}$ None w' xcm ∧ (∀ p. (last xcm p) = ((last xc) p, ε))"
using ‹w' = add_matching_recvs w› sync_run_to_mbox_run w_sync_run **by** blast
then have "w' ∈ $\mathcal{T}_{None}$" **using** MboxTraces.intros **by** auto
then have "w = w'$\downarrow_{!}$" **using** ‹w = w$\downarrow_{!}$› ‹w' = add_matching_recvs w› adding_recvs_keeps_send_order **by** auto
then have "(w'$\downarrow_{!}$) ∈ $\mathcal{L}_{\infty}$" **using** ‹w' ∈ $\mathcal{T}_{None}$› **by** blast
then show ?thesis **using** ‹w = w'$\downarrow_{!}$› ‹w' ∈ $\mathcal{T}_{None}$› **by** blast
qed
qed
qed
qed

Mécanisation de la Théorie de la Synchronisabilité pour les Automates Communicants avec Sémantique de Boîte aux Lettres

Nicole KETTLER

Résumé

Nous étudions les automates communicants, un modèle où les participants d'un protocole de communication sont représentés par des automates finis échangeant des messages. Ces échanges peuvent être synchrones (instantanés) ou asynchrones (via des files d'attente).
Dans des travaux antérieurs [1], nous avons analysé la synchronisabilité, une propriété garantissant que la communication asynchrone ne produit pas de comportements non autorisés par la communication synchrone.
Cet article introduit le problème généralisé de synchronisabilité, en utilisant des automates avec états finaux, et démontre son indécidabilité par une réduction du problème de correspondance de Post. Toutefois, pour certaines topologies comme les arbres, la décidabilité est rétablie en montrant que les langages de tampons sont réguliers et calculables. Cela généralise les résultats obtenus pour les topologies en anneau [4] et ouvre des perspectives vers les structures multi-trees.
Contrairement aux approches précédentes qui se fondaient sur la comparaison des traces d'envoi, cette étude propose une analyse directe du contenu des tampons pour évaluer la synchronisabilité.
Au cours de ce stage, l'objectif est de mécaniser les principales preuves de [1] dans l'assistant de preuve Isabelle.

Mots-clés : Synchronisabilité, Isabelle, Vérification.

Abstract

We study communicating automata, a model where participants in a communication protocol are represented as finite state machines exchanging messages. These exchanges can be synchronous (instant) or asynchronous (via buffers). In previous work (Di Giusto, Laversa, & Peters, 2024), we explored synchronisability, a property ensuring that asynchronous communication does not introduce behaviors beyond those allowed by synchronous communication.
This paper introduces the Generalised Synchronisability Problem, using automata with final states, and shows it is undecidable by reducing from the Post Correspondence Problem. However, decidability is restored for specific communication topologies such as trees, by proving that buffer languages in these settings are regular and computable. This extends results previously obtained for ring topologies [4], and opens avenues for future work on multitree structures.
A key methodological shift from prior approaches is the direct analysis of buffer contents, instead of comparing send traces, to assess synchronisability.
During this internship, we aim to mechanize the main proofs from (Di Giusto et al., 2024) using the Isabelle proof assistant.

Keywords: Synchronizability, Isabelle, Verification.