Communicating automata and quasi-synchronous communications

Loïc Germerie Guizouarn

Automates communicants et communications quasi-synchrones

Loïc GERMERIE GUIZOUARN

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

Présentée en vue de l'obtention du grade de docteur en Informatique **d'**Université Côte d'Azur

Dirigée par : Étienne LOZES,
Professeur des Universités,
Université Côte d'Azur
Co-encadrée par : Cinzia DI GIUSTO,
Maîtresse de conférences,
Université Côte d'Azur
Soutenue le : 19 décembre 2023

Devant le jury, composé de :
Emilio TUOSTO,
Professeur associé,
Gran Sasso Science Institute in l'Aquila
Daniele VARACCA,
Professeur des Universités,
Université Paris Est - Créteil
Yves BERTOT,
Directeur de recherche,
INRIA Sophia-Antipolis
Ilaria CASTELLANI,
Chargée de recherche,
INRIA Sophia-Antipolis
Alain FINKEL,
Professeur des Universités,
Université Paris-Saclay
Damien ZUFFEREY,
Docteur,
SonarSource
Alan SCHMITT,
Directeur de recherche,
INRIA Rennes

AUTOMATES COMMUNICANTS ET COMMUNICATIONS QUASI-SYNCHRONES

Communicating automata and quasi-synchronous communications

Loïc GERMERIE GUIZOUARN

Jury :

Rapporteurs

Emilio TUOSTO,
Professeur associé,
Gran Sasso Science Institute in l'Aquila

Daniele VARACCA,
Professeur des Universités,
Université Paris Est - Créteil

Examinateurs

Yves BERTOT,
Directeur de recherche,
INRIA Sophia-Antipolis

Ilaria CASTELLANI,
Chargée de recherche,
INRIA Sophia-Antipolis

Alain FINKEL,
Professeur des Universités,
Université Paris-Saclay

Damien ZUFFEREY,
Docteur,
SonarSource

Directeur de thèse

Étienne LOZES,
Professeur des Universités,
Université Côte d'Azur

Co-encadrant de thèse

Cinzia DI GIUSTO,
Maîtresse de conférences,
Université Côte d'Azur

Membres invités

Alan SCHMITT,
Directeur de recherche,
INRIA Rennes

Université Côte d'Azur

Loïc GERMERIE GUIZOUARN

Automates communicants et communications quasi-synchrones

xi+122 p.

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

Impression : main.tex – 18/3/2024 – 23:36

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

Automates communicants et communications quasi-synchrones

Résumé

Les systèmes distribués sont le plus souvent basés sur l'échange asynchrone de messages entre des agents. La programmation par échanges de messages est largement utilisée en calcul haute performance, en programmation événementielle, dans les architectures orientées service, etc. Malheureusement du fait de la variété des modèles de communication, des ambiguïtés dans les spécifications, de la portabilité limitée du code, ou encore de la difficulté à exécuter des tests, il est très difficile de vérifier les systèmes communicants. Le model-checking de systèmes communicants vise à analyser des modèles formels de systèmes distribués et à détecter automatiquement des erreurs comme des pertes de messages ou des inter-blocages. Ces problèmes sont indécidables pour des systèmes à partir de deux machines, et plusieurs hypothèses restrictives ont été étudiées pour rendre les problèmes décidables. Nous définissons dans cette thèse une nouvelle classe de systèmes : les systèmes réalisables avec des communications synchrones (RSC pour faire court). Les comportements de ces systèmes approximent des comportement synchrones, où les messages sont envoyés et reçus simultanément. Nous nous basons sur cette définition pour étudier la généralisation d'une autre classe de systèmes : les systèmes half-duplex. Un système à deux machines est half-duplex si lorsqu'une machine envoie des messages, l'autre ne peut pas lui en envoyer. Nous étudions également un autre formalisme, permettant de raisonner sur les systèmes de manière globale : les chorégraphies. Ce formalisme décrit les exécutions de manière synchrone, et un des problèmes qui y est associé est de vérifier si la combinaison des comportements de chaque acteur qui y est décrit est conforme à la description globale. Nous proposons d'utiliser les propriétés des systèmes RSC pour traiter ce problème.

Mots-clés : Communications, vérification, automates communicants.

Communicating automata and quasi-synchronous communications

Abstract

Most of the distributed systems we use nowadays are based on the message-passing paradigm where systems are structured into parties that interact only by sending and receiving messages asynchronously. Message-passing programming is largely employed in high performance computing (MPI, OpenMP, etc), event-driven applications built on top of actor-based languages (Scala, Erlang, etc), service-oriented architectures, peer-to-peer applications, etc. Unfortunately, because of the variety of communication models (peer to peer, mailbox, etc), of the ambiguities of the specifications of the communication primitives, of a limited portability of the code, and of the difficulty of running representative tests, etc, it is error prone and therefore often reserved to experts. Model-checking of communicating automata aims at analysing formal models of distributed systems and discovering bugs like message loss or deadlocks. Due to the asynchronous nature of the communications, this problem is undecidable in general, even with two machines only, and several restrictions have been considered to restore decidability. We define a new one in this thesis: systems that are realisable with synchronous communications (RSC for short), that is the systems whose behaviours are equivalent to synchronous ones. We propose the class of RSC systems as a generalisation of half-duplex systems, which are system of two machines, where a machines does not send any message if it still has some pending messages to be received in its queue. We study another formalism as well: choreographies, which provide a way to reason globally on a system. Choreographies describe synchronous executions, and one of the problems associated with it is checking whether the combination of all participants of the described communication will behave accordingly to the global description. We propose to rely on the properties of RSC systems to study this problem.

Keywords: Communication, model-checking, communicating automata.

Remerciements

Je tiens à commencer par remercier tous les membres du jury d'avoir été là, que ça soit physiquement ou grâce à des technologies de communication efficaces, de leur intérêt, et de leur bienveillance. Merci tout particulièrement à Daniele et Emilio d'avoir accepté d'être rapporteurs, et d'avoir, par votre relecture attentive et vos remarques, amélioré ce manuscrit.

Merci également à Étienne et Cinzia qui ont su me guider dans cette entreprise, qui m'ont fait découvrir le monde des méthodes formelles, vers lequel je ne me serais sans doute pas aventuré, et qui m'ont donné les outils pour me permettre de l'explorer.

Le dernier chapitre de cet ouvrage doit beaucoup à Paul, dont je suis reconnaissant de la contribution sur ReSCu, et à Loïc, qui m'a accompagné dans ma première publication sans mes encadrants. Pour les aspects plus théoriques, je remercie également Amrita de son intérêt pour mon travail, et de ses remarques avisées qui ont permis de mettre en lumière quelques subtilités.

Merci à mes nouveaux collègues de Créteil, et tout particulièrement à Luc, pour l'accueil et pour avoir permis que la fin de ma rédaction se passe aussi bien que possible.

Au delà de la recherche, ces trois années ont également été l'occasion de belles rencontres, et je remercie tous ceux, à l'I3S, avec qui j'ai pu partager de bons moments. Merci aux doctorants qui étaient là, m'ont accueilli, et m'ont tout de suite fait me sentir chez moi : Arthur, Diana, Giulia, Laeti blonde (mais plus maintenant), Marie, Nico, Oussama, Piotr, Rémy, et Samvel. Merci aussi à ceux qui sont arrivés en cours de route : Aymeric, Florian, Margaux, Nina, Romain (change rien), ainsi que Steve et Victor qui m'auront supporté un peu plus longtemps. Je me dois aussi de mentionner Théo et Violette, qui, bien que n'ayant pas répondu positivement aux nombreuses sollicitations à faire une thèse, ont toujours été là, jusqu'au bout de cette aventure.

Remerciements particuliers pour le bureau 222. François et Sara vous avez été les meilleurs co-bureau, et sans vous ce doctorat n'aurait pas été le même. Grazie mille Sara, per la tua energia e la tua generosità, merci aussi de nous avoir rassemblés dans ce bureau ! François, compagnon d'infortune... ma mémoire sélective retiendra bien plus volontiers le compagnon que l'infortune. Si je devais dire si c'était plutôt une bonne ou une mauvaise idée de faire une thèse, je dirais que notre rencontre constitue un argument suffisant pour que je n'ai pas à hésiter un seul instant.

Au delà de toutes ces rencontres, et de ces soutiens, tout au long de mon parcours académique, je tiens à remercier ma famille pour son soutien. Merci tout particulièrement à mes parents qui ont toujours été là pour moi, et qui m'ont offert la chance de faire les études que je voulais.

Pour finir, merci à Laetitia. Même si ce n'est pas toi qui m'a amené à travailler sur les automates communicants, tu as été l'interlocutrice qui m'a fait aimer le faire. Ces mots peuvent sonner creux, mais ce sont ceux qui me semblent les plus justes : je n'aurais pas pu réussir sans toi. Merci pour tes relectures, tes conseils, tes avis éclairés, tes encouragements si précieux, tes figures, et merci d'être à mes côtés, pour tout le reste.

Contents

CHAPTER 1

Introduction

1.1 General context

As for any automated system, reliability of distributed systems is an important object of study. Such a system consists in different actors, which combine their work to achieve a common task. Distributed systems are ubiquitous in applications ranging from web services to high-performance computing. Design errors that lead to downtime of such systems can have enormous consequences for their users, including financial ones.

The 'distributed' aspect of these systems means that the actors are independent from each other, and exchange messages to synchronise or share information. The communication between these actors is therefore an important aspect of their operation. In fact, a potential weakness specific to distributed systems is communication errors. There are several error situations that can arise. For example, a system can end up in a *deadlock*, where the system is stuck because at the same time, all participants are waiting for a message from another one. Because they are all waiting, no participant is sending any messages, and therefore they all wait forever. The problem we address in this thesis is precisely preventing these errors, or rather to ensure that a system is free of them.

To isolate the communication of distributed systems, we reason on what we will call their *communication protocol*. Informally, a communication protocol is a description of the sequences of messages that may be exchanged between a fixed set of participants. Defining such a protocol allows to abstract away all the other aspects of the system, such as the internal behaviour of each participant, and to focus on the verification of its communication only.

In this context, formal methods are used to certify the safety of such protocols. The intuition behind these methods is to define a formalism in which the protocols can be modelled, and then to prove safety properties of these models. A slight variant to this approach is to define the formalism to constrain the models it describes such that they benefit from safety properties by construction.

1.2 Background

Use of formal methods in relation to verification of communications of distributed systems was extensively studied. Some early works used classical finite state automata to represent the states of a protocol [Bochmann 1978], and in the same spirit, [Zafiropulo et al. 1980] proposed a setting that is close to the communicating automata we will present in the next section.

Early on, Petri nets were used to model communications [David and Alla 1994]. In a Petri net, places may be filled with tokens, enabling transitions. Transitions can provide tokens to places, while consuming tokens from other places. Those tokens typically represent the availability of a resource. Research into Petri nets as a way to model communications has led to the emergence of trace theory. Trace theory consists in reasoning about the sequences of communication actions a

distributed system can generate. A notable example of this line of work is Mazurkiewicz traces [Mazurkiewicz 1986]. In a Mazurkiewicz trace, events of a communication that can happen concurrently can be commuted. Those traces are algebraically characterised thanks to free partial monoids. *The Book of Traces* [Diekert and Rozenberg 1995] provides a thorough collection of the works on traces as a way to represent concurrent executions.

Modern works on formalisation of communication protocols tend to fall in one of the two following categories: a global description of all the possible communications between a set of participants, and a distributed description, where each participant is specified independently and their interaction is studied afterward. We will discuss various formalisms from the former category after presenting one of the most common example of the latter: *communicating automata*.

1.2.1 Communicating automata

In a system of communicating automata, each participant of the modelled communication is represented by a finite state automaton. The transitions of these automata are labelled by actions: either to send or receive messages. Typically, a reception transition can be *executed* (that is, the automaton to which it belongs may go from the origin state of this transition to its destination) if the message is *available*: intuitively, it must have been sent already, and not yet received.

The two main semantics defining when a message is available, and more generally, how automata of such systems exchange messages, are *synchronous* and *asynchronous* communications. In systems working synchronously, a reception transition can be executed only if one other automaton of the system executes a send action for the same message, at the same time. In an asynchronous setting, systems are equipped with a fixed set of unbounded buffers, in which messages are sent, and where they remain until a reception removes them. Typically, buffers behave as First In First Out queues (FIFO) or as bags (out of order). In a FIFO buffer, the only message that is available for reception is the one that arrived first among the messages it contains, whereas any message in a bag buffer may be received at any time.

Verification of a system of communicating automata usually consists in checking whether some 'bad' configurations are reachable or not. Informally, a *configuration* is a snapshot of the system: the control states of each automaton and the content of the buffers (in an asynchronous setting). A configuration of a system is said *reachable* if the system can execute transitions until it reaches this configuration, and we call a sequence of transitions that can be executed by a system an *execution*. Bad configurations may represent the ones in which the system, or a single participant, is blocked. For instance, such an issue can occur when the automaton of each participants reach a control state from which they have only reception transitions, and all the buffers are empty (or no send action is executable in a synchronous setting). As no participant can execute any transition, they all end up stuck waiting for some messages that will never arrive. This example of bad configuration is a translation in the formalism of communicating automata of the deadlock situation we described earlier as an example of communication errors for distributed systems.

Given a finite set of bad configurations of a system, checking whether at least one of them is reachable using synchronous communications is decidable. However, the actual behaviour of distributed systems is often asynchronous. Indeed, distributing a system is only relevant if each participant is allowed to proceed with its actions at its pace, not waiting to send messages. Unfortunately, using asynchronous communications, reachability of a configuration is undecidable in general [Brand and Zafiropulo 1983a]. Intuitively, this is because unbounded FIFO buffers can

mimic the tape of a Turing machine, and therefore the halting problem reduces to the reachability of a configuration.

From this result, an interesting research topic regarding communicating automata is the definition of *classes* of systems for which the reachability (and possibly other verification problems) is decidable. Ideally, *membership* to these classes, that is to say, deciding whether a given system is a member of the class, should be decidable as well. Finally, these classes should not be over-restrictive, ie a significant proportion of actual protocols should fall into them.

1.2.2 Global approaches

One of the most prominent formalism to reason globally on a communication protocol is *multiparty session types* [Bettini et al. 2008; Honda et al. 2008]. They are an extension to any number of participants of session types, which defined a type system applied to communications between two participants [Honda 1993; Takeuchi et al. 1994]. A global type describes the communication as a whole, and *local types* are deduced for each participant in an operation called *projection*. Typicaly, this operation is only defined when the global type is *well-formed*. It is sufficient to check each participant's implementation against its local type, independently, to ensure that their combination will be well-behaved.

There are several notions of well-formedness of global types in the various works on multiparty session types. Sometimes, well-formedness is explicitly defined, and sometimes it can be implicitly deduced from the projection function. In the latter case, a well-formed global type is one for which projection is defined. A key difference between this approach and communicating automata in general is that here, safety properties of the protocol are ensured by construction. A distributed implementation typed by the projection of a well-formed global type will benefit from various safety properties, depending on the definition of well-formedness. For instance, for many multiparty session type settings, such an implementation may never reach a deadlock as defined in the previous section.

There are some natural connections between multiparty session types and communicating automata however, and using the latter as local types was studied in [Deniélou and Yoshida 2012a]. Following this work, characterisations of the systems of communicating automata that could have been obtained by projection of a well-formed global type were proposed in [Deniélou and Yoshida 2013; Lange, Tuosto et al. 2015a]. By definition, such systems benefit from the same safety properties ensured for multiparty session types.

Without the background of type theory, *choreographies* are a generic name for formalisms where the communication is described as a whole, and behaviours for the participants are extracted from this description. A problem of interest for choreographic settings is deciding the *realisability* of a global description. A global description is realisable if there exists a local description of all participants, so that the combination of these local descriptions behave exactly as described globally. Using communicating automata as a formalism of these local behaviours, as well as realisability, has been studied in [Barbanera et al. 2020; Basu, Bultan and Ouederni 2012a].

The last approach we mention here makes use of MSCs. They are a visual representation of interactions introduced in [ITU-TS 1993] (the most recent revision of this norm is [ITU-TS 2011]). They are composed by a vertical line per participant, messages are represented by arrows between their sender and their receiver, and time flows vertically from top to bottom. An arrow represented above another means the message it represents was exchanged before the other. This formalism connects back to Mazurkiewicz traces, as like them, they are essentially partial ordering of the

events of a communication. This formalism was extensively studied [Genest, Muscholl and Peled 2003], and languages of MSCs have been proposed as a global description of protocols [Alur, Etessami et al. 2003; Alur and Yannakakis 1999; Genest, Muscholl, Seidl et al. 2006; Muscholl and Peled 1999]. For these languages, the *realisability* problem is similar to that of choreographies. Knowing whether some behaviours, not described as part of an MSC language, are implied by this MSC language, is a difficult problem, and is undecidable in general [Lohrey 2003].

1.3 Our objectives

The aim of the current work is twofold. First, we want to characterise, in communicating automata, a bridge between the decidability of the verification of synchronous communications and the expressiveness of asynchronous ones. An important aspect for us is the practicality of the techniques we develop. This means that the complexity of these techniques should be low enough to make them usable. Second, we want to explore the links between this characterisation and the various global approaches we mentioned in the previous section. More precisely, we aim at establishing a characterisation of well-formedness of choreographies.

To this aim, we rely heavily on an existing notion: *Realisable with Synchronous Communications* (RSC for short). This notion was defined for *computations* [Charron-Bost et al. 1996], but we could adapt it to define a class of systems of communicating automata. It defines computations (partial orders on communicating actions: sending or receptions of messages) that *could* have been obtained in a synchronous setting. The idea behind such a computation is that between the sending of any message and its reception, nothing *has to* happen.

1.4 Contributions and outline

Relying on the RSC notion introduced in [Charron-Bost et al. 1996], we introduce the class of RSC systems in Chapter 3. To do so, we define RSC executions, and we discuss their relation with synchronous executions. We show that the set of RSC executions a system can produce is regular and we provide a way to build its representation for any given system of communicating automata.

In Chapter 4, we discuss decidability of model-checking problems for RSC systems. More precisely, we show that checking if a system of communicating automata is RSC is decidable, and that various safety properties are decidable for RSC systems. In fact, all regular safety properties, that is the ones that can be expressed as the reachability of a regular set of configurations, are decidable. This work was initially published in [Di Giusto, Germerie Guizouarn and É. Lozes 2021], were the RSC notion was named ‘greedy’, and later refined in a long version of this paper [Di Giusto, Germerie Guizouarn and E. Lozes 2023].

After this study of the intrinsic properties of RSC communicating automata, we explore, in Chapters 5 and 6, some of their applications. Namely, we propose RSC systems as the generalisation to multiparty of the binary half-duplex systems from [Cécé and Finkel 2005] (Chapter 5). This comparison was initially published in [Di Giusto, Germerie Guizouarn and E. Lozes 2023].

In Chapter 6, we define a choreographic setting, and we characterise its well-formedness using RSC communicating automata. We conclude this chapter with a comparison of our choreographic setting with various works based on global description approaches. These works range from multiparty session types to languages of MSCs.

The final contribution of this thesis is ReSCu, a tool that implements the model-checking techniques developed in Chapter 4, as evidence of our interest in the practicality of our approach. This tool, introduced in [Desgeorges and Germerie Guizouarn 2023], is presented in Chapter 7.

Following this introduction, Chapter 2 provides the formal definitions necessary to the rest of the work.

Notations

Finite state automata and regular languages

Σ	an alphabet
$\mathcal{L}$	a language
s	a letter
w	a word
$\mathsf{letters}(w)$	the multiset of the letters of w
$\mathcal{A}$	a finite state automaton
Q	a set of control states
F	a subset of accepting control states
q^0	an initial control state
q	a control state
$(\mathsf{q}, w, \mathsf{q}')$	a sequence of transitions from q to q' while reading letters of w
$w \sqsubseteq w'$	w is a subword of w'
$[w]$	encoding of w

Systems of communicating automata

$\mathfrak{S}$	a system of communicating automata
L	a set of control states of a communicating automaton
$\mathbb{V}$	a set of messages
$\mathbb{P}$	a set of processes
$\mathbb{I}$	a set of buffers
$\mathbb{I}^F$	a set of FIFO buffers
$\mathbb{I}^B$	a set of bag buffers
Λ	an alphabet of system actions
Ω	an alphabet of communications
Υ	an alphabet of matched communications
S	a set of send actions
R	a set of receive actions
Act	a set of actions of a system
γ	a configuration
γ_0	an initial configuration
l	a local control state
$\mathbf{l}$	a global control state (vector of local control states)
$\mathbf{b}$	a tuple of buffer contents
$\mathbf{b}^\varnothing$	a tuple of empty buffers
$RS(\mathfrak{S})$	the reachability set of $\mathfrak{S}$
Γ	a set of configurations

Communicating actions

v	a message
ι	a buffer identifier
$a_{\mathcal{A}}$	a communicating automaton action
a	a system action
b	a buffer content
$p, q, \ldots$	some participants of the communication
$\iota?^{p}\mathsf{v}$	the reception of message v from buffer ι by participant p
$\iota!^{p}\mathsf{v}$	the send action of message v in buffer ι by participant p
c	a communication
$\iota!^{p}?^{q}\mathsf{v}$	the communication gathering the actions $\iota!^{p}\mathsf{v}$ and $\iota?^{p}\mathsf{v}$
$\mathsf{process}\,(a)$	the process of action a
$\mathsf{buffer}\,(a)$	the buffer of action a
$\mathsf{message}\,(a)$	the message of action a
$\gamma \xrightarrow[\mathfrak{S}]{a} \gamma'$	the transition of system $\mathfrak{S}$ going from configuration γ to configuration γ' by executing action a
$\gamma \xRightarrow[\mathfrak{S}]{e} \gamma'$	the sequence of transitions in system $\mathfrak{S}$ going from configuration γ to configuration γ' executing the sequence of actions e

Choreographies

$\mathcal{C}$	a choreography
$\alpha\,(\mathcal{C})$	the implementation of $\mathcal{C}$
$w \upharpoonright p$	projection of w on participant p
$\mathcal{A}_{part}$	automaton recognising the partial closure of a choreography
$\mathfrak{p}$	set of blocked participants

Graphs

G	a graph
V	a set of vertices
v	a vertex
A	a set of arcs
$\mathsf{d}^{-}(v)$	the input degree of vertex v
$\mathsf{d}^{+}(v)$	the output degree of vertex v

Executions and their graphical representation

μ	an MSC
$\mathsf{agraph}(e)$	the action graph of execution e
λ	an action graph labelling function
$\mathsf{lin}\,(\mathsf{agraph}(e))$	the set of linearisations of $\mathsf{agraph}(e)$
$\mathsf{cgraph}\,(e)$	the conflict graph of execution e
$\rightarrow_e$	an arc of the conflict graph of e
κ_e	the labelling function of the conflict graph of e
$e \sim e'$	executions e and e' are causally equivalent
$[\![e]\!]_\sim$	set of executions causally equivalent to e
$\widetilde{\mathcal{L}}$	the causal closure of the language of executions $\mathcal{L}$
$e \prec e'$	execution e is a prefix of e'
$\mathsf{pre}\,(e)$	the prefixes of e
$e \trianglelefteq e'$	execution e is a partial execution of e'
$e \not\trianglelefteq e'$	execution e is not a partial execution of e'
$\mathsf{partials}\,(\mathcal{L})$	the closure by partial executions of the language of executions $\mathcal{L}$
P	a safety property
$P\,(\mathfrak{S})$	the configurations of $\mathfrak{S}$ satisfying P
$\mathcal{A}_{P(\mathfrak{S})}$	the automaton recognising $P\,(\mathfrak{S})$
e	an execution
$\mathsf{com}\,(e)$	the set of communications of e
$\mathsf{cte}\,(w)$	the execution corresponding to the sequence of communications w
$\mathsf{etc}\,(c)$	the execution corresponding to the actions composing c
$e \restriction_\iota$	the sequence of actions of e whose buffer is ι
$\mathsf{unmatched}\,(w)$	the subword of w containing only the unmatched communications
$\mathsf{executions}\,(\mathfrak{S})$	the set of executions of $\mathfrak{S}$
$\mathsf{executions}_{\mathsf{rsc}}\,(\mathfrak{S})$	the set of RSC executions of $\mathfrak{S}$
$\mathsf{executions}_{\mathsf{sync}}\,(\mathfrak{S})$	the set of synchronous executions of $\mathfrak{S}$
$\mathcal{A}_{rsc}$	the automaton recognising all RSC executions of a system
$\mathfrak{f}$	a set of blocked FIFO buffers
$\mathfrak{b}$	a set of blocked messages in a bag buffer

CHAPTER 2

Preliminaries

We begin with some generally accepted definitions about standard formalisms in computer science. These include graphs (we will only use the directed variant in this thesis), regular languages and finite state automata. We will follow by defining a setting of communicating automata, with particular emphasis on the definitions of the sequences of actions of such systems, and of their graphical representations. We will conclude this chapter with a discussion about how our setting compares with previous works.

2.1 General definitions

For a finite set S, S^* denotes the set of finite words over S, $w \cdot w'$ denotes the concatenation of words w and w', $|w|$ denotes the length of word w, and ε denotes the empty word. We write a letter in bold to denote a vector (for example $\mathbf{b}$), and b_i for the i-th component of $\mathbf{b}$.

2.1.1 Finite State Automata and regular languages

We recall usual definitions and results in the field of automata and languages.

Definition 2.1.1 (Finite State Automata)**.** Let Σ be an alphabet, a finite state automaton over Σ, denoted $\mathcal{A}$, is a tuple $(\mathsf{Q}, \delta, \mathsf{q}^0, \mathsf{F})$ where

- Q is a finite set of control states,
- $\mathsf{q}^0 \in \mathsf{Q}$ is the initial control state,
- $\mathsf{F} \subseteq \mathsf{Q}$ is the set of accepting control states, and
- $\delta \subseteq \mathsf{Q} \times \Sigma \times \mathsf{Q}$ is the transition function.

A word $w \in \Sigma^*$ is *recognised* by $\mathcal{A} = (\mathsf{Q}, \delta, \mathsf{q}^0, \mathsf{F})$ if there exist a sequence of transitions $t_1, t_2, \ldots, t_n$ such that for all $i \in \{1, \ldots, n\}, t_i \in \delta$ with $t_i = (\mathsf{q}_{i-1}, s, \mathsf{q}_i)$, $\mathsf{q}_0 = \mathsf{q}^0$, and $\mathsf{q}_n \in \mathsf{F}$. We say that q_n is an *accepting state of* w. The set of all words recognised by $\mathcal{A}$ is called the *language of* $\mathcal{A}$, and is denoted $\mathcal{L}(\mathcal{A})$. A language $\mathcal{L}$, or set of words, is *regular* if there exists a finite state automaton $\mathcal{A}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{A})$.

For a word $w = s_1 \cdot \ldots \cdot s_n$ in an alphabet Σ, and a word $w' = s'_1 \cdot \ldots \cdot s'_m$, we say that w' is a *subword* of w (denoted $w' \sqsubseteq w$) if there exists a function α such that for all $i \in \{1, \ldots, m\}$, $\alpha(i) \in \{1, \ldots, n\}$ and $s'_i = s_{\alpha(i)}$, and for all $\{i, i'\} \subseteq \{1, \ldots, m\}, i < i'$ if and only if $\alpha(i) < \alpha(i')$. Intuitively, a subword of w is a word that can be obtained by removing some letters of w.

For a word $w = s_1 \cdot \ldots \cdot s_n$ in an alphabet Σ, $\mathsf{letters}(w) = \{s_1, \ldots, s_n\}$ is the multiset containing all the letters of w, as many times as they were present. By abuse of notation, we write $s \in w$ if $s \in \mathsf{letters}(w)$.

2.1.2 Graphs

Definition 2.1.2 (Directed labelled graph)**.** Given a set of labels L, a *directed labelled graph* is a tuple (V, A, l) where

- V is a finite set of *vertices*,
- $A \subseteq V \times V$ is a set of *arcs*: for $v, v' \subseteq V, (v, v') \in A$ means that v is connected to v',
- $l : V \to L$ is a *labelling function*, assigning a label to each vertex.

For a directed graph $G = (V, A, l)$ and a vertex $v \in V$, we write $\mathsf{d}^-(v)$ (respectively $\mathsf{d}^+(v)$) for the number of vertices v' such that $(v', v) \in A$ (respectively $(v, v') \in A$).

Definition 2.1.3 (Graph isomorphism)**.** Let L be a set of labels, $G = (V, A, l)$ and $G' = (V', A', l')$ two directed labelled graphs, G and G' are isomorphic if there exists a bijection f between V and V', such that for all $v \in V, l(v) = l'(f(v))$, and for $v, v' \in V$, $(v, v') \in A$ if and only if $(f(v), f(v')) \in A'$.

Informally, the *induced subgraph* of a graph G is a graph obtained by removing vertices from G, alongside with all the vertices that were reachable through an arc from the removed ones.

Definition 2.1.4 (Consistent induced subgraph)**.** Let $G = (V, A, l)$ and $G' = (V', A', l')$, G is a consistent induced subgraph of G' if

1. G is an induced subgraph of G',
2. for all $\{v, v'\} \subseteq V$, if $(v, v') \in A$, and $v' \in V'$, then $v \in V'$.

2.2 Communicating automata

Generally speaking, a communication protocol is a set rules describing the possible interactions between actors, called *participants*. For a given protocol, the set of all participants of the communication is denoted $\mathbb{P}$. We begin with an informal description of a simple communication protocol, that we will use as a running example.

Example 2.2.1 – We will consider a generic client/server protocol, enhanced with a database logging activity. In this protocol, the client may send a request to the server, and when it receives a result for this request, it sends an acknowledgement back to the server. The server waits for a request, and upon receiving it, it sends a result to the client. After that, it waits for an acknowledgement from the client and sends a logging message to the database. Those behaviours can be repeated indefinitely.

Systems of communicating automata are a model used to formally represent communication protocols. Each participant is represented by a finite state automaton, the transitions of which are labelled by actions, either to send or receive a message. Communications modelled by communicating automata are asynchronous, which means that messages are sent to a specified buffer, where they wait until they are explicitly received. These buffers can be *First In First Out*, or *FIFO* for short, or *bag*. The latter are buffers from which messages can be received out of order, whereas, in FIFO buffers, they have to be received in the order in which they were sent.

2.2.1 Communicating automaton

We now define formally a communicating automaton.

Definition 2.2.1 (Communicating automaton)**.** A *communicating automaton* $\mathcal{A}$ is a tuple $\left(L, \mathbb{V}, \mathbb{I}^F, \mathbb{I}^B, \mathsf{Act}, \delta, l^0\right)$ where:

- L is a finite set of control states,
- $\mathbb{V}$ is a finite set of *messages*,
- $\mathbb{I} = \mathbb{I}^F \cup \mathbb{I}^B$ with $\mathbb{I}^B \cap \mathbb{I}^F = \varnothing$ is a finite set of *buffer identifiers* where $\mathbb{I}^B$ (respectively $\mathbb{I}^F$) is the subset of bag (respectively FIFO) buffer identifiers,
- $\mathsf{Act} \subseteq \mathbb{I} \times \{!, ?\} \times \mathbb{V}$ is a finite set of *communicating automaton actions*,
- $\delta \subseteq L \times \mathsf{Act} \times L$ is a finite set of *transitions*, and
- l_0 is the initial control state.

A *communicating automaton action* (denoted $a_{\mathcal{A}}$) can be a send action: $\iota!\mathsf{v}$, meaning 'send message v in buffer ι', or a reception: $\iota?\mathsf{v}$ meaning 'receive message v from buffer ι'. For $a_{\mathcal{A}} = \iota \dagger \mathsf{v}$ with $\dagger \in \{!, ?\}$, $\mathsf{buffer}\,(a_{\mathcal{A}}) = \iota$ and $\mathsf{message}\,(a_{\mathcal{A}}) = \mathsf{v}$.

We mostly consider *deterministic* communicating automata, as defined below. Except when explicitly mentioned, a communication automaton is deterministic.

Definition 2.2.2 (Deterministic communicating automaton)**.** Let $\mathcal{A} = \left(L, \mathbb{V}, \mathbb{I}^F, \mathbb{I}^B, \mathsf{Act}, \delta, l^0\right)$ be a communicating automaton, $\mathcal{A}$ is *deterministic* if for all $l \in L$, for all communicating automaton action $a_{\mathcal{A}} \in \mathsf{Act}$, for all pair of control states $(l', l'') \in L^2$, if $(l, a_{\mathcal{A}}, l') \in \delta$ and $(l, a_{\mathcal{A}}, l'') \in \delta$, then $l' = l''$.

A *non-deterministic* communicating automaton may violate the condition of Definition 2.2.2, and its transition function δ is a subset of $L \times (\mathsf{Act} \cup \{\varepsilon\}) \times L$, meaning its transitions may have no action labelling them.

2.2.2 System of communicating automata

A *system of communicating automata*, denoted by $\mathfrak{S}$, is a family of communicating automata, one per participant $p \in \mathbb{P}$, where actions of each automaton are tagged with the identifier of its participant.

Definition 2.2.3 (System of communicating automata)**.** Let $\mathbb{P}$ be a finite set of participant, a system of communicating automata is a family $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$, where for all $p \in \mathbb{P}$, $\mathcal{A}_p$ is the communicating automaton representing participant p.

Example 2.2.2 – Figure 2.1 displays the graphical representation of $\mathfrak{S}_{csd}$, a system of communicating automata encoding formally the protocol from Example 2.2.1. We equipped each participant with a buffer (not represented graphically) from which it receives all its messages. To improve clarity, we named the buffers with the initial of the participant it is associated to.

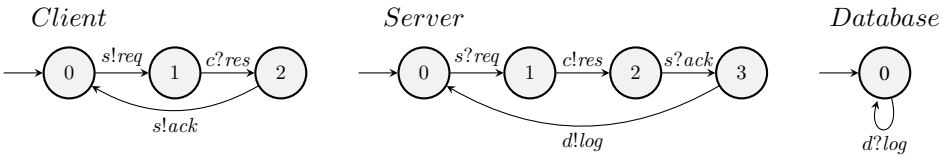


Figure 2.1: System $\mathfrak{S}_{csd}$ of Communicating Automata encoding the protocol from Example 2.2.1

Definition 2.2.4 (Product of a system)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, where for all $p \in \mathbb{P}$, $\mathcal{A}_p = \left(L_p, \mathbb{V}_p, \mathbb{I}_p^F, \mathbb{I}_p^B, \mathsf{Act}_p, \delta_p, l_p^0\right)$ is the communicating automaton representing participant p, then $\mathsf{product}\,(\mathfrak{S}) = \left(L_\mathfrak{S}, \mathbb{V}_\mathfrak{S}, \mathbb{I}_\mathfrak{S}, \mathsf{Act}_\mathfrak{S}, \delta_\mathfrak{S}, \mathbf{l}_\mathfrak{S}^0\right)$ is the product of $\mathfrak{S}$, where:

- $L_\mathfrak{S} = \prod_{p\in\mathbb{P}} L_p$ is the set of global control states of the system: for $\mathbf{l} \in L_\mathfrak{S}$, $\mathbf{l} = (l_p)_{p\in\mathbb{P}}$ is a vector of control states, where for each participant p, l_p is a control state of the automaton representing p;
- $\mathbf{l}^0 = \left(l_p^0\right)_{p\in\mathbb{P}}$ is the initial global state;
- $\mathbb{V}_\mathfrak{S} = \bigcup_{p\in\mathbb{P}} \mathbb{V}_p$ is the set of messages;
- $\mathbb{I}_\mathfrak{S} = \mathbb{I}_\mathfrak{S}^F \cup \mathbb{I}_\mathfrak{S}^B$ is the set of buffer identifiers, where $\mathbb{I}_\mathfrak{S}^F = \bigcup_{p\in\mathbb{P}} \mathbb{I}_p^F$ is the set of FIFO buffers identifiers and $\mathbb{I}_\mathfrak{S}^B = \bigcup_{p\in\mathbb{P}} \mathbb{I}_p^B$ is the set of bag buffers identifiers;
- $\mathsf{Act}_\mathfrak{S} = \bigcup_{p\in\mathbb{P}} \{\iota\,\dagger^p\,\mathsf{v} \mid \dagger \in \{!,?\}, \iota\,\dagger\,\mathsf{v} \in \mathsf{Act}_p\}$ is the set of *system actions*;
- $\delta_\mathfrak{S} = \left\{(\mathbf{l}, a, \mathbf{l}') \mid \exists p \in \mathbb{P}, (l_p, a, l'_p) \in \delta_p, \forall q \neq p, l_q = l'_q\right\}$.

A *system action* is a communicating automaton action tagged with the process performing it. This allows to ensure that all sets of actions of the participants in a system are disjoint. We extend the definition of $\mathsf{buffer}\,(a)$ and $\mathsf{message}\,(a)$ to system actions as expected, and for a system action $a \in \mathsf{Act}_\mathfrak{S}$, with $a = \iota\,\dagger^p\,\mathsf{v}$ for $\dagger \in \{!,?\}$ and $p \in \mathbb{P}$, $\mathsf{process}\,(a) = p$. We will refer to system actions as *actions*.

Independently of a system, given a set of processes $\mathbb{P}$, a set of buffer identifier $\mathbb{I}$, and a set of messages $\mathbb{V}$, the alphabet of send actions is $S_{\mathbb{P},\mathbb{I},\mathbb{V}} = \{\iota!^p\mathsf{v} \mid \iota \in \mathbb{I}, p \in \mathbb{P}, \mathsf{v} \in \mathbb{V}\}$, the alphabet of receptions $R_{\mathbb{P},\mathbb{I},\mathbb{V}} = \{\iota?^p\mathsf{v} \mid \iota \in \mathbb{I}, p \in \mathbb{P}, \mathsf{v} \in \mathbb{V}\}$, and the alphabet of actions is $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}} = S_{\mathbb{P},\mathbb{I},\mathbb{V}} \cup R_{\mathbb{P},\mathbb{I},\mathbb{V}}$. Often, $\mathbb{P}, \mathbb{I}$ and $\mathbb{V}$ are not necessary, or obvious from context, and to alleviate notation we simply write S, R and Λ.

For a system $\mathfrak{S}$ with $\mathsf{product}\,(\mathfrak{S}) = \left(L_\mathfrak{S}, \mathbb{V}_\mathfrak{S}, \mathbb{I}_\mathfrak{S}, \mathsf{Act}_\mathfrak{S}, \delta_\mathfrak{S}, \mathbf{l}_\mathfrak{S}^0\right)$, $R_\mathfrak{S} = R_{\mathbb{P},\mathbb{I},\mathbb{V}} \cap \mathsf{Act}_\mathfrak{S}$ is the set of receptions of $\mathfrak{S}$, and $S_\mathfrak{S} = S_{\mathbb{P},\mathbb{I},\mathbb{V}} \cap \mathsf{Act}_\mathfrak{S}$ its set of send actions. To refer to the buffers in which a participant p may receive messages, we write $\mathbb{I}_p^?$, defined as $\{\iota \mid \exists \mathsf{v} \in \mathbb{V}_\mathfrak{S},\ \iota!^p\mathsf{v} \in \mathsf{Act}_\mathfrak{S}\}$.

We say that a state is *final* if there are no transitions available from it.

Definition 2.2.5 (Final state)**.** Let $\mathfrak{S}$ be a system with $\mathsf{product}\,(\mathfrak{S}) = \left(L_\mathfrak{S}, \mathbb{V}_\mathfrak{S}, \mathbb{I}_\mathfrak{S}, \mathsf{Act}_\mathfrak{S}, \delta_\mathfrak{S}, \mathbf{l}_\mathfrak{S}^0\right)$. A state $\mathbf{l} \in L_\mathfrak{S}$ is *final* if for all $\mathbf{l}' \in L_\mathfrak{S}$, for all $a \in \mathsf{Act}_\mathfrak{S}$, $(\mathbf{l}, a, \mathbf{l}') \notin \delta_\mathfrak{S}$.

A configuration of a system is a snapshot of all its components at a given time: it is the control state of all automata composing the system, as well as the content of all the buffers.

Definition 2.2.6 (Configuration)**.** Let $\mathfrak{S}$ be a system of communicating automata, a *configuration* of $\mathfrak{S}$ (denoted γ) is a pair $(\mathbf{l}, \mathbf{b})$ where $\mathbf{l} \in L_{\mathfrak{S}}$ is a global control states, and $\mathbf{b} = (b_\iota)_{\iota \in \mathbb{I}_{\mathfrak{S}}}$ is a vector of buffers: for each $\iota \in \mathbb{I}$, $b_\iota \in (\mathbb{V}_{\mathfrak{S}})^*$ is the concatenation of the messages contained in the buffer ι.

The initial configuration of a system $\mathfrak{S}$ is denoted γ_0, and is defined as $(\mathbf{l}^0, \mathbf{b}^\varnothing)$, with $\mathbf{b}^\varnothing = (\varepsilon)_{\iota \in \mathbb{I}_{\mathfrak{S}}}$. Although we represent the content of bag buffers as words, we consider these words as unordered. This means that two configurations $(\mathbf{l}, \mathbf{b})$ and $(\mathbf{l}', \mathbf{b}')$ are equal if $\mathbf{l} = \mathbf{l}'$, for all FIFO buffer identifier ι, $b_\iota = b'_\iota$, and for all bag buffer identifier ι', $\mathsf{letters}(b_{\iota'}) = \mathsf{letters}(b'_{\iota'})$.

Intuitively, a configuration where a buffer contains messages represents a transient state of the system: we expect the message to be received at some point. In contrast, we say that a configuration in which no message is waiting to be received is *stable*.

Definition 2.2.7 (Stable configuration)**.** A configuration $(\mathbf{l}, \mathbf{b})$ is *stable* if $\mathbf{b} = \mathbf{b}^\varnothing$.

Definition 2.2.8 (Transition)**.** A *transition* of $\mathfrak{S}$ is a tuple (γ, a, γ'), often written $\gamma \xrightarrow[\mathfrak{S}]{a} \gamma'$, where $\gamma = (\mathbf{l}, \mathbf{b})$ and $\gamma' = (\mathbf{l}', \mathbf{b}')$ are two configurations, a is an action, and the following holds:

- $(\mathbf{l}, a, \mathbf{l}') \in \delta_{\mathfrak{S}}$
- if $a = \iota!^p\mathsf{v}$, then $b'_\iota = b_\iota \cdot \mathsf{v}$, and for all $j \in \mathbb{I}_{\mathfrak{S}}, j \neq \iota, b_j = b'_j$
- if $a = \iota?^p\mathsf{v}$, then for all $j \in \mathbb{I}_{\mathfrak{S}}, j \neq \iota, b_j = b'_j$ and
  - if $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ then $b_\iota = \mathsf{v} \cdot b'_\iota$
  - if $\iota \in \mathbb{I}^B_{\mathfrak{S}}$ then $\exists w, w' \in (\mathbb{V}_{\mathfrak{S}})^*$, $b_\iota = w \cdot \mathsf{v} \cdot w'$, and $b'_\iota = w \cdot w'$.

If a system is equipped exclusively with FIFO buffers, we sometimes say that it is a FIFO system. Similarly, a bag system is equipped with bag buffers only.

2.2.3 Communication architectures

In the setting we chose, we allow any communication architecture: there is no relation *a priori* between the set of processes and the set of buffers. A participant may read and write from the same buffer, and there might be more than one buffer between a given pair of participants. However, most of the work in the literature uses a specific communication architecture.

The most common one is peer-to-peer, where a buffer is used in each direction between each pair of participants.

Definition 2.2.9 (Peer-to-peer systems)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be a system of communicating automata, such that $\mathsf{product}\,(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. The system $\mathfrak{S}$ is *peer-to-peer* if $\mathbb{I}_{\mathfrak{S}} = \mathbb{I}^F_{\mathfrak{S}}$ and if there exists an injection $r : \mathbb{I} \to \mathbb{P} \times \mathbb{P}$ such that for all $p \in \mathbb{P}$:

1. if $\iota!\mathsf{v} \in \mathsf{Act}_p$, then $r(\iota) \in \{p\} \times (\mathbb{P} \setminus \{p\})$, and
2. if $\iota?\mathsf{v} \in \mathsf{Act}_p$, then $r(\iota) \in \mathbb{P} \setminus p \times \{p\}$.

Note that a participant is not allowed to send and receive in the same buffer. Another common architecture is *mailbox*, where each participant is equipped with a single buffer, in which it receives messages from all participants.

Definition 2.2.10 (Mailbox systems)**.** A system $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$, such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$, is mailbox if there exist a injection $r : \mathbb{I}_{\mathfrak{S}} \to \mathbb{P}$ such that for all $p \in \mathbb{P}$:

1. for all $a_{\mathcal{A}} \in \mathsf{Act}_p$, if $a_{\mathcal{A}} = \iota?\mathsf{v}$ then $r(\iota) = p$, and
2. if $a_{\mathcal{A}} = \iota!\mathsf{v}$, then $r(\iota) \neq p$.

Observe that systems with two participants are peer-to-peer if and only if they are mailbox. They are equipped with two buffers, one in each direction between the two participants. These systems are referred to as *binary* systems.

2.3 Executions

Given an alphabet of actions Λ, an *execution* is a word on Λ: it is a finite sequence of actions. An execution is *feasible* in a system if this system is able to exhibit this sequence of actions.

Definition 2.3.1 (Feasible execution)**.** An execution $e = a_1 \cdot a_2 \cdot \ldots \cdot a_n$ is *feasible* in $\mathfrak{S}$ if there exists a sequence of configurations $\gamma_1, \gamma_2, \ldots, \gamma_n$ such that for all $i \in \{1, \ldots, n\}$, $\gamma_{i-1} \xrightarrow[\mathfrak{S}]{a_i} \gamma_i$.

The set of all feasible executions of $\mathfrak{S}$ is denoted $\mathsf{executions}(\mathfrak{S})$. We write $\gamma_0 \xRightarrow[\mathfrak{S}]{e} \gamma_n$ for $\gamma_0 \xrightarrow[\mathfrak{S}]{a_1} \gamma_1 \xrightarrow[\mathfrak{S}]{a_2} \cdots \xrightarrow[\mathfrak{S}]{a_n} \gamma_n$, and by abuse of notation we write $a \in e = a_0 \cdot \ldots \cdot a_n$ if there exists $i \in \{1, \ldots, n\}$ such that $a = a_i$. If an execution e is feasible in a system $\mathfrak{S}$, we also say that $\mathfrak{S}$ *admits* e.

A configuration γ of $\mathfrak{S}$ is reachable if there exists an execution $e \in (\mathsf{Act}_{\mathfrak{S}})^*$ such that $\gamma_0 \xRightarrow[\mathfrak{S}]{e} \gamma$. The set of all reachable configurations of $\mathfrak{S}$, called *reachability space*, is denoted $RS(\mathfrak{S})$.

For an execution $e = a_1 \cdot \ldots \cdot a_n$ over an alphabet $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$, we write $\mathsf{message}(e)$ for the word $\mathsf{message}(a_1) \cdot \ldots \cdot \mathsf{message}(a_n)$ in $\mathbb{V}^*$. We write $e' = e \restriction_\iota$ for $\iota \in \mathbb{I}$ if $e' \sqsubseteq e$ such that $\mathsf{letters}(e) \cap \Lambda_{\mathbb{P},\{\iota\},\mathbb{V}} = \mathsf{letters}(e') \cap \Lambda_{\mathbb{P},\{\iota\},\mathbb{V}}$. The execution e' is e restricted to actions implying buffer ι. The notion of prefix of an execution is defined as it is for words.

Definition 2.3.2 (Prefix of an execution)**.** Let e_1 and e_2 be two executions. We say that e_1 is a *prefix* of e_2, denoted $e_1 \prec e_2$, if there exists e' such that $e_1 \cdot e' = e_2$. Given a language of executions $\mathcal{L}$, the prefix closure of $\mathcal{L}$ is $\mathsf{pre}(\mathcal{L}) = \{e \mid \exists e' \in \mathcal{L}, e \prec e'\}$.

In an execution, we say that the set formed by the index of the send action of a message and that of the reception of this very message is a *matching pair*.

Definition 2.3.3 (Matching pair)**.** Let $e = a_1 \cdot \ldots \cdot a_n$ be an execution over $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$. A set of two indices $\{j, j'\} \subseteq \{1, \ldots, n\}$ with $j < j'$ is a *matching pair* if there exist $\iota \in \mathbb{I}$, $\mathsf{v} \in \mathbb{V}$, and $(p, q) \in \mathbb{P}^2$ such that:

1. $a_j = \iota!^p\mathsf{v}$,
2. $a_{j'} = \iota?^q\mathsf{v}$,

3. and there exists k such that
   (a) if $\iota \in \mathbb{I}^F$ then a_j (respectively $a_{j'}$) is the k-th send action (respectively reception) on ι in e,
   (b) else $\left(\iota \in \mathbb{I}^B\right)$, a_j (respectively $a_{j'}$) is the k-th send action (respectively reception) of message v on ι in e.

For bag buffers, we say that when the same message is sent several times to a buffer, receptions of this message match the send actions in their order. A bag buffer can be seen as a set of FIFO buffers, one per message name. This choice is explained in the discussion, Section 2.5.2.

An action a_j is *unmatched* in e if there is no j' such that $\{j, j'\}$ is a matching pair. By abuse of notation, we often use the term matching pair to refer to the actions whose indices form a matching pair in a given execution.

Note that, as executions are words on an alphabet of actions, with no restriction whatsoever, some of them are ill-formed: receptions could occur before send actions for example. Because of the $j < j'$ condition in Definition 2.3.3, such reception would not be part of a matching pair, meaning they are receptions of messages that were not sent in this execution. Intuitively, we call ill-formed any execution that could not be feasible in any system.

Example 2.3.1 – Consider an execution $e = \iota?\mathsf{v}_1 \cdot \iota!\mathsf{v}_1 \cdot \iota'?\mathsf{v}_2 \cdot \iota'!\mathsf{v}_3 \cdot \iota'?\mathsf{v}_3$, where all the buffers are FIFO. It is ill-formed: because of the semantics rules of Definition 2.2.8, a reception cannot happen from the initial configuration of a system, because all the buffers are empty in this configuration. Another issue is what happens in buffer ι': because $\iota'!\mathsf{v}_3$ is the first send action in this buffer, and $\iota'?\mathsf{v}_3$ is the second reception, these actions do not form a matching pair. In fact, there are no matching pairs at all in this execution.

To avoid corner cases that could happen because of these ill-formed executions, we will focus on what we call well-formed executions.

Definition 2.3.4 (Well-formed execution)**.** Let Λ be an alphabet of actions. An execution $e = a_0 \cdot \ldots \cdot a_n$ over Λ^* is well-formed if for all $i \in \{1, \ldots, n\}$, if $a_i \in R$ then there exists $j \in \{1, \ldots, i-1\}$ such that $\{j, i\}$ is a matching pair in e.

We say that two actions of a well-formed execution *commute* if:

- they do not form a matching pair,
- they are not actions of the same type on the same FIFO buffer,
- they are not actions of the same type with the same message on the same bag buffer,
- and they are not performed by the same participant.

Intuitively, two actions commute if their order was not observable from the viewpoint of any individual component of the system, participant or buffer.

For an execution $e = a_1 \cdot \ldots \cdot a_n$, we say that $j \prec_e j'$, with $\{j, j'\} \subseteq \{1, \ldots, n\}$ if $j < j'$ and a_j does not commute with $a_{j'}$. The relation $\prec_e$ represents causal dependencies between actions of an execution.

We say that two executions that are equivalent up to reordering of actions that commute are *causally equivalent*.

Definition 2.3.5 (Causal equivalence)**.** Let Λ be an alphabet of actions. Two executions $e = a_1 \cdot \ldots \cdot a_n \in \Lambda^*$ and $e' = a'_1 \cdot \ldots \cdot a'_n \in \Lambda^*$ are *causally equivalent*, denoted $e \sim e'$, if there exists a permutation σ of $\{1, \ldots, n\}$ such that:

(1) for all $i \in \{1, \ldots, n\}, a'_{\sigma(i)} = a_i$, and

(2) for all $\{j, j'\} \subseteq \{1, \ldots, n\}, j \prec_e j'$ if and only if $\sigma(j) \prec_{e'} \sigma(j')$.

We denote with $[\![e]\!]_\sim$ the set of all executions causally equivalent to e. We defined the causal closure of a set of execution $\mathcal{L}$ as follow: $\mathcal{L}^{\widetilde{}} = \bigcup_{e \in \mathcal{L}} [\![e]\!]_\sim$.

The conditions on the actions on bag buffers make this notion compatible with our definition of matching pairs, where bag buffers behave like if they were composed by a FIFO buffer per message it contains. Commuting two send actions or two receptions of the same message on the same bag buffer could change which send action matches each reception, leading to causally equivalent executions where the matching pairs are not the same.

Example 2.3.2 – Let $e = \iota!^p\mathsf{v} \cdot \iota!^q\mathsf{v} \cdot \iota?^r\mathsf{v}$ and $e' = \iota!^q\mathsf{v} \cdot \iota!^p\mathsf{v} \cdot \iota?^r\mathsf{v}$ be two executions, with ι being a bag buffer. If we did not prevent actions using the same message on the same buffer to commute, we would have $e \sim e'$ according to Definition 2.3.5. However, according to Definition 2.3.3, the send action matched by the final reception in both e and e' is the first one in each execution, so the only matching pair is not the same in these two executions.

Lemma 2.3.1. *Let $\mathfrak{S}$ be a system of communicating automata, e_1 and e_2 be two executions feasible in $\mathfrak{S}$ with $\gamma_0 \xRightarrow[\mathfrak{S}]{e_1} \gamma$ and $\gamma_0 \xRightarrow[\mathfrak{S}]{e_2} \gamma'$. If $e_1 \sim e_2$, then $\gamma = \gamma'$.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be a system of communicating automata, such that its product is $\mathsf{product}(\mathfrak{S}) = (L_\mathfrak{S}, \mathbb{V}_\mathfrak{S}, \mathbb{I}_\mathfrak{S}, \mathsf{Act}_\mathfrak{S}, \delta_\mathfrak{S}, \mathbf{l}^0_\mathfrak{S})$. Let e_1 and e_2 be two executions feasible in $\mathfrak{S}$ with $\gamma_0 \xRightarrow[\mathfrak{S}]{e_1} (\mathbf{l}, \mathbf{b})$ and $\gamma_0 \xRightarrow[\mathfrak{S}]{e_2} (\mathbf{l}', \mathbf{b}')$.

Observe that, in two causally equivalent executions, the order of the actions of each process happen in the same order, so for all $p \in \mathbb{P}$, the sequence of communicating automaton actions executed on $\mathcal{A}_p$ is the same, so $l_p = l'_p$.

We reason similarly for the buffers, beginning with FIFO ones. For a buffer $\iota \in \mathbb{I}^F$, b_ι is equal to the sequence of sent messages to which the sequence of received messages is removed. These two sequences are equal for two causally equivalent executions, so $b_\iota = b'_\iota$. For bag buffers now, the key argument is that because the actions are the same in e_1 and e_2, there are as many send actions and receptions of each message in each bag buffer. For a bag buffer ι, $b_\iota = b'_\iota$. □

In a synchronous setting, send and receive actions of the same message are indivisible. The whole exchange happens as one single step. In our setting, we say that an execution is *synchronous* if it is a sequence of matching pairs. If we consider the configurations reached after a matching pair as a *step* of a synchronous execution, each step is stable.

Well-formed executions are composed by exchanges of messages between participants: a message is either sent and received, or sent and remaining in a buffer. We consider the pair of actions constituting such exchanges, and call them *communications*. Formally, a communication (denoted c) is a set of actions: either $\{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$ or $\{\iota!^p\mathsf{v}\}$. We say that a communication of the first kind is a *matched* communication, and a communication of the second kind is an *unmatched* one.

Definition 2.3.6 (Communications of an execution)**.** Let $e = a_0 \cdot \ldots \cdot a_n$ be an execution over an alphabet of actions Λ. The multiset of all communications of e is

$$\begin{aligned}\mathsf{com}\,(e) = &\{\{a_j, a_{j'}\} \mid \{j, j'\} \text{ is a matching pair of } e\} \cup \\ &\{\{a_j\} \mid a_j \in S, \forall j' \in \{1, \ldots, n\}, \{j, j'\} \text{ is } \textit{not} \text{ a matching pair of } e\}\,.\end{aligned}$$

Observe that because unmatched communications can only contain send actions, if an execution e is ill-formed, some of its actions will not be part of a communication in $\mathsf{com}\,(e)$. We sometimes need to relate specific actions and their communication. For an execution $e = a_1 \cdot \ldots \cdot a_n$, we assume an arbitrary ordering $c_1, \ldots, c_m$ of the communications of e (that is for all $i \in \{1, \ldots, m\}$, $c_i \in \mathsf{com}\,(e)$). We sometimes need a function ϑ, which associates each action index to the index of the communication it belongs to: for all $i \in \{1, \ldots, n\}, \vartheta(i) = j$ if c_j contains action a_i specifically. This implies that for $j \in \{1, \ldots, m\}, \vartheta^{-1}(j) = \{i, i'\}$ if c_i is a matched communication and $\vartheta^{-1}(j) = i$ otherwise.

Example 2.3.3 – Consider the execution e from Example 2.3.1. Because there are no matching pairs in e, $\mathsf{com}\,(e) = \{\{\iota!\mathsf{v}_1\}, \{\iota'!\mathsf{v}_3\}\}$. The send actions form unmatched communications.

If we consider now a well-formed execution $e' = \iota!\mathsf{v}_1 \cdot \iota'!\mathsf{v}_2 \cdot \iota?\mathsf{v}_1$, we have $\mathsf{com}\,(e') = \{\{\iota!\mathsf{v}_1, \iota?\mathsf{v}_1\}, \{\iota'!\mathsf{v}_2\}\}$.

We define an alphabet of communications:

$$\Omega_{\mathbb{P},\mathbb{I},\mathbb{V}} = \left\{\iota!^p?^q\mathsf{v} \mid \iota \in \mathbb{I}, (p, q) \in \mathbb{P}^2, \mathsf{v} \in \mathbb{V}\right\} \cup \{\iota!^p\mathsf{v} \mid \iota \in \mathbb{I}, p \in \mathbb{P}, \mathsf{v} \in \mathbb{V}\}\,.$$

Similarly to Λ, we omit the subscript and write Ω when $\mathbb{P}$, $\mathbb{V}$, and $\mathbb{I}$ are obvious or not necessary. Notation $\iota!^p?^q\mathsf{v}$ stands for $\{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$, and $\iota!^p\mathsf{v}$ stands for the communication $\{\iota!^p\mathsf{v}\}$. To refer to an alphabet of matched communications only, we define $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}} = \{\iota!^p?^q\mathsf{v} \mid \iota \in \mathbb{I}, (p, q) \in \mathbb{P}^2, \mathsf{v} \in \mathbb{V}\}$. Often, we need to refer to the alphabets of communications that are possible in a given system. To do so, we use $\Upsilon_{\mathfrak{S}} = \{\iota!^p?^q\mathsf{v} \mid \iota!^p\mathsf{v} \in \mathsf{Act}_{\mathfrak{S}}, \iota?^q\mathsf{v} \in \mathsf{Act}_{\mathfrak{S}}\}$ and $\Omega_{\mathfrak{S}} = \Upsilon_{\mathfrak{S}} \cup \{\iota!^p\mathsf{v} \mid \iota!^p\mathsf{v} \in \mathsf{Act}_{\mathfrak{S}}\}$.

2.4 Graphical representations

We present several graphical representations of executions, and more specifically of their causal dependencies. We survey *Message Sequence Charts*, and motivate the need for *action graph*s in our setting.

2.4.1 Messages Sequence Charts

Message Sequence Charts have been introduced as a standard in [ITU-TS 1993], and are a visual representation of communications of a distributed system. Each participant of the communication is represented by a vertical line, and a message is represented by an arrow from its sender to the receiver. An *unmatched* message, that is sent, but not received, is represented by a partially dotted arrow. We say that the intersections between arrows and vertical lines are *events*, either *send* or *receive* ones. Time flows top to to bottom on each vertical line, meaning that an event that is below another happens later.

Example 2.4.1 – An MSC is depicted in Figure 2.2. It represents an interaction between three participants: p, q and r. First, a message v_1 is exchanged from p to q, and then a message v_2 is exchanged between r and q. After that, an unmatched message v_3 is sent to p by q.

We give the formal definition of an MSC:

Definition 2.4.1 (Message Sequence Chart)**.** Let $\mathbb{P}$ be a set of processes, $\mathbb{V}$ be a set of messages and $\mathbb{I}$ be a set of buffers identifiers. An MSC μ over action alphabet $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ is a tuple $(E, \lambda, m, \preceq_{proc})$ where

- E is a finite set of events;
- λ is a function from E to $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$, such that for $\mathsf{e} \in E$, $a = \lambda(\mathsf{e})$ is the action corresponding to event e;
- m is a binary relation on E, matching the send action and the reception of the same message:
  - for all $(\mathsf{e}, \mathsf{e}') \in m$, there exists $(p, q) \in \mathbb{P}^2, \iota \in \mathbb{I}, \mathsf{v} \in \mathbb{V}$ such that $\lambda(\mathsf{e}) = \iota!^p\mathsf{v}$ and $\lambda(\mathsf{e}') = \iota?^q\mathsf{v}$;
  - for all $\mathsf{e} \in E$ such that $\lambda(\mathsf{e}) \in R_{\mathbb{P},\mathbb{I},\mathbb{V}}$, there exists a unique $\mathsf{e}' \in E$ such that $(\mathsf{e}', \mathsf{e}) \in m$; and
  - for all $\mathsf{e} \in E$ such that $\lambda(\mathsf{e}) \in S_{\mathbb{P},\mathbb{I},\mathbb{V}}$, there exists at most one $\mathsf{e}' \in E$ such that $(\mathsf{e}, \mathsf{e}') \in m$, and $\lambda(\mathsf{e}') \in R_{\mathbb{P},\mathbb{I},\mathbb{V}}$;
- $\preceq_{proc}$ is a partial order over E, such that for all $p \in \mathbb{P}$, $\preceq_{proc}$ induces a total order on the events of p, that is on $\lambda^{-1}\left(S_{\{p\},\mathbb{I},\mathbb{V}} \cup R_{\{p\},\mathbb{I},\mathbb{V}}\right)$.
- $(m \cup \preceq_{proc})^+$, the transitive closure of m and $\preceq_{proc}$, is irreflexive.

We write $\preceq_\mu$ for the transitive closure of $\preceq_{proc}$ and m, and call this partial order the *visual order* of μ.

A linearisation of an MSC $\mu = (E, \lambda, m, \preceq_{proc})$, with $E = \{1, \ldots, n\}$, is an execution $e = a_1 \cdot \ldots \cdot a_n$ such that there exists a permutation σ of E with $\lambda(\mathsf{e}) = a_{\sigma(\mathsf{e})}$, and for all $\{\mathsf{e}, \mathsf{e}'\} \subseteq E$, $\mathsf{e} \preceq_\mu \mathsf{e}'$ implies $\sigma(\mathsf{e}) \leq \sigma(\mathsf{e}')$. We denote with $\mathsf{lin}\,(\mu)$ the set of linearisations of μ.

MSCs can be used to reason about executions of systems of communicating automata. If $\mathfrak{S}$ is a system, and $e \in \mathsf{executions}\,(\mathfrak{S})$, the MSC $\mu = (E, \lambda, m, \preceq_{proc})$ associated with $e = a_1, \ldots, a_n$ is defined such that $E = \{1, \ldots, n\}$, for all $\mathsf{e} \in E, \lambda(\mathsf{e}) = a_\mathsf{e}$, for all $\{\mathsf{e}, \mathsf{e}'\} \subseteq E, (\mathsf{e}, \mathsf{e}') \in m$ if and only if $\{\mathsf{e}, \mathsf{e}'\}$ forms a matching pair in e.

2.4.2 Action Graphs

A shortcoming of MSCs is that not all their linearisations are valid executions. While they are very well suited to reason about executions in a peer-to-peer setting, they lack information to reason about executions in a more generic communication architecture, and even for structured ones like mailbox.

Example 2.4.2 – Consider as an example the MSC μ from Figure 2.2, with FIFO mailbox communication: both v_1 and v_2 are sent in the same buffer. The execution

$$e = \iota!^p\mathsf{v}_1 \cdot \iota!^r\mathsf{v}_2 \cdot \iota?^q\mathsf{v}_1 \cdot \iota?^q\mathsf{v}_2 \cdot \iota'!^q\mathsf{v}_3$$

is a valid linearisation of μ, but

$$e' = \iota!^r\mathsf{v}_2 \cdot \iota!^p\mathsf{v}_1 \cdot \iota?^q\mathsf{v}_1 \cdot \iota?^q\mathsf{v}_2 \cdot \iota'!^q\mathsf{v}_3$$

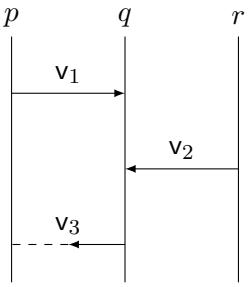


Figure 2.2: Example of an MSC

is not: message v_2 is sent first in ι but is received second, which is not compatible with the FIFO semantic.

As we consider a generic setting with no restriction on the communication architecture in general, we might have to deal with trickier situations: knowing the sender and the receiver of a message could be insufficient to know which buffer is involved in this communication. Even though this could be fixed by adding the information about the buffer in the MSC, a more suitable option is to use another graphical representation of execution: action graphs.

Definition 2.4.2 (Action Graph)**.** Let $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ be an alphabet of actions, and $e = a_1, \ldots, a_n$ be an execution over this alphabet. The action graph of e, denoted $\mathsf{agraph}(e)$, is the vertex-labelled directed graph $(\{1, \ldots, n\}, \prec_e, \lambda)$, where for all $i \in \{1, \ldots, n\}, \lambda(i) = a_i$, and arcs follow from causal dependency: there is an arc between vertices i and j if $i \prec_e j$.

Action graphs are acyclic, by definition of $\prec_e$: remember that $a \prec_e a'$ if a' occurs in e after a. Action graphs embed by definition all the information about causal dependency. The set of linearisations of an action graph $\mathsf{agraph}(e) = (\{1, \ldots, n\}, \prec_e, \lambda)$ is

$$\{a_{i_1} \cdot \ldots \cdot a_{i_n} \mid \forall j \in \{1, \ldots, n\}, a_{i_j} = \lambda(i_j), \\ \text{and } i_1, \ldots, i_n \text{ is a topological ordering of } \mathsf{agraph}(e)\}.$$

Because arcs in action graphs correspond to causal ordering of actions of the executions they represent, two executions e and e' are causally equivalent if their action graphs $\mathsf{agraph}(e)$ and $\mathsf{agraph}(e')$ are isomorphic. This implies that $\mathsf{lin}\,(\mathsf{agraph}(e)) = [\![e]\!]_\sim$.

The graphical representation of action graphs can be close to the one of MSCs: we chose to represent actions of the same process vertically, in the order they have to be from top to bottom.

Example 2.4.3 – Figure 2.3 is the representation of the action graph of the execution e from Example 2.4.2. Contrary to what we saw in its MSC in Figure 2.2, there is an arc between the two send actions, as they act on the same FIFO buffer. This arc prevent the wrong linearisation from earlier.

As all the causal dependencies are represented in an action graph, the visual representation can be cumbersome. Because there is a causal dependency between actions of the same process, the number of arcs is quadratic in the number of actions, and such a graph is not easily understandable. To circumvent this issue, we opted for a visual representation of action graphs omitting the arcs between actions of the same participant, except from the arc between two actions immediately following each other. Figure 2.4 is an example of this drawback, and presents the same action graph as Figure 2.3, with all its arcs.

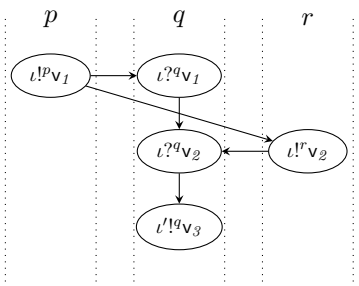


Figure 2.3: Action graph of execution e from Example 2.4.2

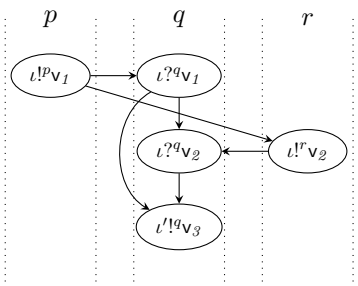


Figure 2.4: Complete representation of action graph from Figure 2.3

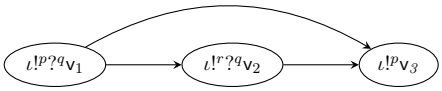


Figure 2.5: Representation of a conflict graph

2.4.3 Conflict Graphs

Another graphical tool that we will use to characterise executions is the *conflict graph*, which is intuitively obtained from the action graph by merging matching pairs of vertices.

Definition 2.4.3 (Conflict graph)**.** Given an execution $e = a_1 \cdot \ldots \cdot a_n$ with $\mathsf{com}\,(e) = \{c_1, \ldots, c_m\}$, the conflict graph $\mathsf{cgraph}\,(e)$ of the execution e is the vertex-labelled directed graph $(\{1, \ldots, m\}, \rightarrow_e, \kappa_e)$ where for all $i \in \{1, \ldots, m\}, \kappa_e(i) = c_i$, and for all $\{k, k'\} \subseteq \{1, \ldots, m\}$, $k \rightarrow_e k'$ if there is $i \in \vartheta^{-1}(k)$ and $j \in \vartheta^{-1}(k')$ such that $i \prec_e j$.

Example 2.4.4 – Let e be the execution from Example 2.4.2. Figure 2.5 shows the visual representation of $\mathsf{cgraph}\,(e)$. We used the compact representation of communications defined in the end of Section 2.3.

Lemma 2.4.1. *Let e and e' be two executions, $e \sim e'$ if and only if* $\mathsf{cgraph}\,(e)$ *is isomorphic to* $\mathsf{cgraph}\,(e')$.

Proof.

Let e and e' be two executions such that $e \sim e'$. By Definition 2.3.5, we know that e is a permutation of e', and because order of actions of the same kind on the same buffer is preserved between e and e', the k^{th} send, respectively receive, action on a buffer ι in e is also the k^{th} send, respectively receive, action ι in e'. We can deduce that actions forming a matching pair in e form a matching pair in e' as well. From this consideration we have that $\mathsf{com}\,(e) = \mathsf{com}\,(e')$, and from item (2) Definition 2.3.5, and Definition 2.4.3 of the conflict graphs, $k_1 \rightarrow_e k_2$ if and only if there exists $a_i \in \kappa_{k_1}$ and $a_j \in \mathsf{cgraph}\,(k_2)$ such that $a_i \prec_e a_j$, so $a_i \prec_{e'} a_j$, so there exists $\{k'_1, k'_2\} \subseteq \{1, \ldots, |\mathsf{com}\,(e')|\}$ such that $a_i \in \kappa_{e'}(k'_1)$ and $a_j \in \kappa_{e'}(k'_2)$, with $k'_1 \rightarrow_{e'} k'_2$.

Now, we show that two executions with isomorphic conflict graphs are causally equivalent. Let $e = a_1 \cdot \ldots \cdot a_n$ and $e' = a'_1 \cdot \ldots \cdot a'_n$ two executions such that $\mathsf{cgraph}\,(e)$ is isomorphic to $\mathsf{cgraph}\,(e')$. Because their conflict graphs are isomorphic, we know that $\mathsf{com}\,(e) = \mathsf{com}\,(e')$.

Let $\{i, j\} \subseteq \{1, \ldots, n\}$ such that a_i and a_j do not commute in e. We know that a_i and a_j do not commute in e' either. Assume that $a_i \prec_e a_j$, then there exists $\{k_1, k_2\} \subseteq \{1, \ldots, |\mathsf{com}\,(e)\,|\}$ such that $a_i \in \kappa_e(k_1)$, $a_j \in \kappa_e(k_2)$, and $k_1 \rightarrow_e k_2$. Because $\mathsf{cgraph}\,(e)$ is isomorphic to $\mathsf{cgraph}\,(e')$, there exists $\{k'_1, k'_2\} \subseteq \{1, \ldots, |\mathsf{com}\,(e)\,|\}$ such that $a_i \in \kappa_{e'}(k'_1)$, $a_j \in \kappa_{e'}(k'_2)$, and $k'_1 \rightarrow_{e'} k'_2$; therefore $a_i \prec_{e'} a_j$. □

2.5 Discussion

Now that we have introduced the formal framework we will be working with in the next chapters, we will discuss how the definitions we have chosen for its various elements are related to previous work.

We begin with an overview of the different definitions of communicating automata, and how they take into account the possible buffer semantics and communication architectures. Then we discuss the graphical representation of executions: we discuss how definitions of MSCs from the literature are related to action graphs. Finally, we will discuss how our conflict graphs differs from those that have been introduced in the literature so far.

2.5.1 Communicating automata

The definition we provided for communicating automata is as generic as possible. In particular, most of the works around this formalism consider a limited amount of communication architectures, typically peer-to-peer (like in [Brand and Zafiropulo 1983a; Genest, Kuske et al. 2007; Heußner, Leroux et al. 2012; Kuske and Muscholl 2021] to mention a few), or sometimes mailbox (e.g. in [Basu, Bultan and Ouederni 2012b; Bouajjani, Enea et al. 2018a; Finkel and É. Lozes 2017])*. In the present thesis, we reason on a generic setting that subsumes any communication architecture from these works.

We only considered two semantics for our buffers: FIFO or bag, and we allow buffers using different ones coexisting in the same system. This is a similar setting as the one studied in [Clemente et al. 2014], where topologies composed by both bag and FIFO buffers are studied. Asynchronous communications relying on bag buffers are referred to as *fully asynchronous communication* in [Chevrou et al. 2016]. This term gives the intuition of a communication where the only order between actions comes from the fact they are done by the same participant (or between a send action and its corresponding reception). While most of the literature focused on FIFO communication, [Basu and Bultan 2016] and [Akroun and Salaün 2018] are instances of works studying bag buffers. Contrary to what we defined here, both these works did not define a notion of matching pair. The semantic of [Akroun and Salaün 2018] is defined as if for each buffer, there was a counter per possible message. Each time a message is sent to the buffer, its counter is incremented, and each time a message is received from the buffer, its counter is decremented. In [Basu and Bultan 2016], each message is treated as if it had been tagged with its sender and recipient. This forbids a shared bag buffer to contain a message that could be received by multiple participants (to mimic a token for example).

*Buffers are sometimes called channels in the literature.

2.5.2 Fully-bag causal equivalence

The definition we chose for matching pairs (Definition 2.3.3) is not the only possible one. This is because in general, it is not possible to know which actions form matching pairs in an execution using bag buffers with our semantics. This is illustrated in the example below.

Example 2.5.1 – Let $e = \iota!^p\mathsf{v} \cdot \iota!^q\mathsf{v} \cdot \iota?^r\mathsf{v}$ be an execution such that ι is a bag buffer. According to Definition 2.3.3, $\iota?^r\mathsf{v}$ is matched with $\iota!^p\mathsf{v}$. However, from the operational semantics point of view (as hinted in Definition 2.2.8), we could also consider that $\iota?^r\mathsf{v}$ is matched with $\iota!^q\mathsf{v}$.

We could have considered a more relaxed definition of matching pairs, which we call *fully-bag matching pairs*, in which receptions are arbitrarily (but injectively) matched to any send action that happened before.

This reasoning extends to *fully-bag causal equivalence*: two actions *can* commute if they are actions of the same type on the same bag buffer implying the same message. Two executions e and e' are fully-bag causally equivalent if they are causally equivalent according to Definition 2.3.5 with the modified condition to commute, and both executions match the same actions together (because fully-bag matching pairs are arbitrary and cannot be determined with only the information contained in the execution itself). Once again, if two executions are causally equivalent, they are fully-bag causally equivalent as well.

The issue with fully-bag matching pairs is that an execution in itself does not embed enough information to determine which send action matches which reception, which is why we rely on the arbitrary association of matching pairs from Definition 2.3.3. Furthermore, in practice, there is little difference between the matching pairs from Definition 2.3.3 and fully-bag matching pairs. Causal equivalence using matching pairs from our definition still allow to reorder actions of the same type in bag buffers, and receptions from the same buffer do not have to be in the same order as their send actions (as long as they do not involve the same message).

We did not however consider 'lossy' buffers. This is a semantic where messages may randomly disappear from a buffer. The idea is to mimic errors in the communication. Systems using them were studied in [Abdulla and Jonsson 1996a; b]. Some combinations of FIFO and lossy buffers were also studied in [Chambart and Schnoebelen 2008].

2.5.3 MSCs and action graphs

MSCs were studied in numerous papers, and Definition 2.4.1 is similar to the definition of MSCs from [Lohrey and Muscholl 2002] or [Genest, Muscholl and Peled 2003] for instance. A difference is that in both these definitions, the visual order $\preceq_\mu$ is defined as a component of the MSC itself while in our proposal it is deduced from the other components. This approach is identical to the definition of MSCs from [Laversa 2021]. In the remainder of this work, we will not formally rely on MSCs. We will discuss them in Chapter 6 when we discuss global descriptions of communication protocols. We defined them to motivate the use of action graphs, which are more suited to reason about executions when no specific communication architecture is assumed.

The intuition of action graphs is present in [Bollig, Katoen et al. 2010], where MSCs are represented as graphs, allowing to easily visualise the partial order between the actions of an execution. However, Bollig, Katoen et al. did not consider other communication architecture than peer-to-peer, so their graph representation of MSCs was just a different visualisation, which did not add information to them.

Action graphs are closely related to partially ordered multisets (pomsets). If we consider the multiset of the actions of an execution rather than their indexes, and we consider the relation of causal dependencies as a partial order on the actions themselves, we obtain a pomset that embeds the same information as our action graphs. A reason why we relied on the graph formalism is that it made it easier to

2.5.4 Conflict graphs

The notion of conflict graph appeared in [Bouajjani, Enea et al. 2018a], inspired by [Papadimitriou 1979]. Contrary to our Definition 2.4.3, Bouajjani, Enea et al. label the arcs of their conflict graph rather that the nodes. They lose the information about which communication is in conflict with another one, but the label on an arc between two communications allow to know which action of each communication created the conflict. Arc labels are of the kind XY, for $(X, Y) \in \{S, R\}^2$. There is an arc between two communications c, c' if an action of c' *depends on* an action of c. The value of Y (respectively X) indicates which action of c' (respectively c) depends on c (respectively is depended on by c'). An S label means that the send action is involved in the dependency, and an R means it is the reception. Note that there might be multiple arcs between two vertices in a conflict graph from [Bouajjani, Enea et al. 2018a].

By labelling the nodes with the communications themselves, rather than the arcs with information about the nature of the conflict, we do not loose any information. Indeed, given two communications, it is easy in our setting to know whether it is the send action or the possible reception of one communication that conflicts with the other one. For us, the main interest of using such a labelling is that, provided with such a conflict graph, it is easy to rebuild another execution with the same conflict graph. We will use that feature in our proofs.

In both [Bouajjani, Enea et al. 2018a; Di Giusto, Laversa et al. 2020], conflict graphs have been defined as inferred from the MSCs. Therefore, when using another communication architecture than peer-to-peer, they suffer the same drawbacks as the ones illustrated in Example 2.4.2: some of the arcs are there implicitly. This is one of the reasons that led Di Giusto, Laversa et al. to extend the definition from [Bouajjani, Enea et al. 2018a] with new arcs. These arcs represent the causal dependency of send actions from different participants happening in the same buffer. These extra arcs are naturally included in our conflict graphs since we build them from the partial order of our action graphs rather than the one of MSCs.

CHAPTER 3

RSC systems

In this chapter, we study the notion of Realisability with Synchronous Communication, after the definition from [Charron-Bost et al. 1996]. We state how this notion applies to our definition of executions, and we define RSC systems of communicating automata. An interesting result is that the set of RSC executions of a system is regular, we show that by detailing how an automaton recognising them can be built. We conclude this chapter with a discussion of certain aspects of the notion of RSC, and a comparison with related notions from the literature.

3.1 RSC executions

We begin this chapter by defining RSC executions in our setting.

Definition 3.1.1 (RSC execution)**.** Let Λ be an alphabet of actions. A well-formed execution $e = a_1 \cdot \ldots \cdot a_n$ over Λ is *RSC* if all its matching pairs are of the form $\{j, j+1\}$ with $j \in \{1, \ldots, n-1\}$.

Intuitively, an execution is RSC if it mimics a synchronous execution, where the send action and reception of the same message happen at the same time. Note that contrary to actual synchronous executions, here we do not impose that all send actions are matched. We say that an execution e is *causally RSC* if there exists $e' \in [\![e]\!]_\sim$ such that e' is RSC.

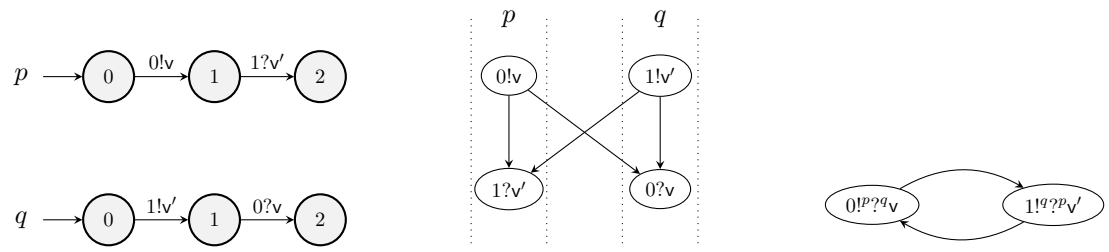
Example 3.1.1 – Let $e_r = s!^c req \cdot s?^s req \cdot c!^s res \cdot c?^c res \cdot s!^c ack_s \cdot s?^s ack_s \cdot d!^c log_c \cdot d?^d log_c \cdot c!^d ack_d \cdot c?^c ack_d \cdot d!^s log_s \cdot d?^d log_s \cdot s!^c req$, this execution is RSC. All the receptions happen right after their corresponding send actions. However, a send actions is not matched: the second $s!^c req$.

Example 3.1.2 – $e_n = 0!^p \mathsf{v} \cdot 1!^q \mathsf{v}' \cdot 1?^p \mathsf{v}' \cdot 0?^q \mathsf{v}$ is not RSC. Actions $1!^q \mathsf{v}'$ and $1?^p \mathsf{v}'$ happen between the send action and the reception of v.

Another example, using a different alphabet of actions, is:

$$e_{eq} = s!^c req \cdot s?^s req \cdot c!^s res \cdot c?^c res \cdot s!^c ack_s \cdot d!^c log_c \cdot d?^d log_c \cdot s?^s ack_s \cdot$$
$$c!^d ack_d \cdot d!^s log_s \cdot c?^c ack_d \cdot s!^c req \cdot d?^d log_s$$

where we linked by an arrow the actions belonging to matching pairs separated by other actions. However, e_{eq} is causally equivalent to e_{er} from Example 3.1.1, so e_{eq} is causally RSC.



(a) Non-RSC system $\mathfrak{S}_n$ (b) $\mathsf{agraph}(e_n)$ (c) $\mathsf{cgraph}\,(e_n)$

Figure 3.1: Non-RSC system, action and conflict graphs of one of its executions

3.1.1 RSC executions and sequences of communications

In an RSC execution, the actions composing a matching pair happen immediately one after the other. This means that for an RSC execution e, there exists an ordering $k_1, \ldots, k_m$ of $\mathsf{com}\,(e)$ such that $e = \mathsf{cte}\,(c_{k_1} \cdot \ldots \cdot c_{k_m})$, where $\mathsf{cte}\,(\{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}) = \iota!^p\mathsf{v} \cdot \iota?^q\mathsf{v}$, and $\mathsf{cte}\,(\{\iota!^p\mathsf{v}\}) = \iota!^p\mathsf{v}$. For a word $w = c_1 \cdot \ldots \cdot c_n$ of communications, we write $\mathsf{cte}\,(w)$ for the execution $\mathsf{cte}\,(c_1) \cdot \ldots \cdot \mathsf{cte}\,(c_n)$.

We denote the inverse of cte with etc: for e, $\mathsf{etc}\,(e) = w$ if $\mathsf{cte}\,(w) = e$. Note that cte is injective, so cte is not always defined, but it is defined for RSC executions.

Intuitively, for e an RSC execution, $\mathsf{etc}\,(e)$ is a sequence of communications equivalent to e: $e = \iota!^p\mathsf{v} \cdot \iota?^q\mathsf{v} \cdot \iota'!^p\mathsf{v}'$, $\mathsf{etc}\,(e) = \{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\} \cdot \{\iota'!^p\mathsf{v}'\}$. This function is not defined for non-RSC executions. This is because in a non-RSC execution, there is at least one action that is scheduled between two actions of the same communication.

If an RSC execution is equivalent to a sequence of communications, the reverse is not true: any sequence of communications is not necessarily equivalent to an RSC execution. Example 3.1.3 below illustrates this.

Example 3.1.3 – Let $w = \iota!^p\mathsf{v} \cdot \iota!^p?^q\mathsf{v}$, and let $e = \iota!^p\mathsf{v} \cdot \iota!^p\mathsf{v} \cdot \iota?^q\mathsf{v}$. We have that $e = \mathsf{cte}\,(w)$. Observe that e is not RSC: the reception of v matches the first send action.

For a sequence of communications $w = c_1 \cdot \ldots \cdot c_m$ over an alphabet $\Omega_{\mathbb{P},\mathbb{I},\mathbb{V}}$, the execution $e = \mathsf{cte}\,(w)$ is RSC if:

- it is well-formed, and
- for all $k \in \{1, \ldots, m\}$, if c_k is a matched communication, with $c_k = \iota!^p?^q\mathsf{v}$ for $(p, q) \in \mathbb{P}^2$, $\iota \in \mathbb{I}$ and $\mathsf{v} \in \mathbb{V}$, then for all $k' \in \{1, \ldots, k-1\}$, $c_{k'} \neq \iota!^r\mathsf{v}$ with $r \in \mathbb{P}$.

A well-formed execution corresponding to a sequence of communications is RSC if no message v is sent on a buffer ι before a matching pair involving this same message on this same buffer. Note that if another message was unmatched before the matching pair, the execution would not be well-formed, as an action not in a matching pair are never matched, preventing the matching pair to happen.

3.1.2 Characterisation

In this section, we show how RSC executions can be graphically characterised. The first observation we make is that RSC executions have acyclic conflict graphs.

Lemma 3.1.1. *Let e be a well-formed execution. If e is RSC, then* $\mathsf{cgraph}\,(e)$ *is acyclic.*

Proof.

Let $e = a_1 \cdot \ldots \cdot a_n$ be a well-formed RSC execution. Let $\mathsf{com}\,(e) = \{c_1, \ldots, c_m\}$ be the multiset of communications of e, and let $\mathsf{cgraph}\,(e) = (\{1, \ldots, m\}, \rightarrow_e, \kappa_e)$. For $j \in \{1, \ldots, m\}$, let $\max_{com}(j) = \max\left(\vartheta^{-1}(j)\right)$ and $\min_{com}(j) = \min\left(\vartheta^{-1}(j)\right)$. In an RSC execution, $k \rightarrow_e k'$ induces $\max_{com}(k) < \min_{com}(k')$, because $c_{k'}$ happens after c_k in the execution, and the two actions of c_k (if there are two) are not separated by any action.

By contradiction, assume $\mathsf{cgraph}\,(e)$ admits a cycle $k_1 \rightarrow_e k_2 \rightarrow_e \cdots \rightarrow_e k_n \rightarrow_e k_1$. This implies $\max_{com}(k_1) < \min_{com}(k_2) \leq \max_{com}(k_2) < \cdots < \min_{com}(k_n) \leq \max_{com}(k_n) < \min_{com}(k_1) \leq \max_{com}(k_1)$. This would imply $\max_{com}(k_1) < \max_{com}(k_1)$, which is a contradiction. □

Intuitively, this is because actions of a communication in a cycle in the conflict graph of an execution have to happen both before and after something else. As the send action has to happen before the reception, this means that there are actions in the execution that have to happen after the send action, and before the reception.

The second observation is that executions with acyclic conflict graphs are causally equivalent to an RSC execution.

Lemma 3.1.2. *Let e be a well-formed execution. If* $\mathsf{cgraph}\,(e)$ *is acyclic, then there exists an RSC execution e' such that $e \sim e'$.*

Proof.

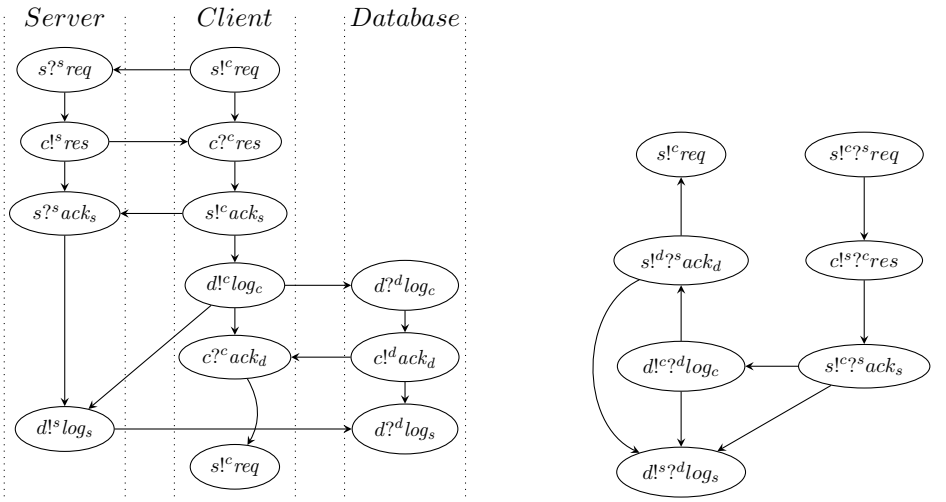
Let e be a well-formed execution, with $\mathsf{com}\,(e) = \{c_1, \ldots, c_m\}$, and assume $\mathsf{cgraph}\,(e) = (\{1, \ldots, m\}, \rightarrow_e, \kappa_e)$ is acyclic. Let $e' = \mathsf{cte}\,(c_{k_1} \cdot \ldots \cdot c_{k_m})$ such that $k_1, \ldots, k_m$ is a topological sort of $\mathsf{cgraph}\,(e)$. By Definition 3.1.1, e' is RSC: it is obtained by concatenation of communications, so actions of matching pairs happen right one after the other. Finally, by Lemma 2.4.1, because by construction, $\mathsf{cgraph}\,(e')$ is isomorphic to $\mathsf{cgraph}\,(e)$, $e \sim e'$. □

An execution whose conflict graph is acyclic can be reordered such that the actions forming a communication are paired, hence it allows an RSC ordering of its actions.

From the previous observations, we can characterise the executions that are causally equivalent to RSC executions: an execution is causally equivalent to an RSC execution if and only if its conflict graph is acyclic. As a corollary, we have that executions that are not RSC are characterised by their cyclic conflict graph.

Example 3.1.4 – Consider execution e_{eq} of Example 3.1.2. It is not RSC, but it is causally equivalent to execution e_r from Example 3.1.1. Their conflict graph can be seen in Figure 3.2b. As expected, because e_{eq} is causally equivalent to an RSC execution, its conflict graph is acyclic.

Example 3.1.5 – Consider execution e_n from Example 3.1.2. As stated in this example, it is not RSC. Its action and conflict graphs can be seen in Figure 3.1b and Figure 3.1c respectively. We can see that there is a cycle in the conflict graph, preventing this execution to be reordered to be RSC.



(a) The action graph $\mathsf{agraph}(e_{eq})$ (b) The conflict graph $\mathsf{cgraph}\,(e_{eq})$

Figure 3.2: Causal dependencies of an execution of Client/Server/Database protocol

3.2 RSC Systems

We begin with the formal definition of an RSC system.

Definition 3.2.1 (RSC system)**.** A system $\mathfrak{S}$ is *RSC* if for all execution $e \in \mathsf{executions}\,(\mathfrak{S})$, there exists an RSC execution e' such that $e \sim e'$.

We do not impose all the executions of an RSC system to be RSC, this allows some slack in the synchronisation. A sent message could be received arbitrarily late, but it is ensured that from a causal point of view, nothing prevents the send and receive actions to happen virtually simultaneously. The system from Figure 2.1 is an example of RSC system, but being as simple as it is, it allows few non-RSC scheduling of its executions. To make for a more interesting example, we extended the protocol by giving to the client the possibility to log information in the database.

Example 3.2.1 – The system in Figure 3.3 is RSC. For instance, it is capable of producing execution e_{eq} from Example 3.1.2 which is not RSC. However, execution e_{eq} is causally equivalent to e_r from Example 3.1.1 , which is RSC. Notice that, as expected, e_{eq} and e_r have the same action graph, given in Figure 3.2a.

Example 3.2.2 – The system in Figure 3.1a is not RSC. It admits execution e_n from Example 3.1.2, which is not causally equivalent to an RSC execution. In facts, as it can be observed in the action graph of e_n in Figure 3.1b, any execution causally equivalent to e_n will begin with the two send actions. One of the receptions will necessarily be delayed from its matching send action.

3.3 Automaton $\mathcal{A}_{rsc}$

An RSC execution can be written as a sequence of communications. Either the communication is a matched one and the reception happens right after the send action, or the communication is an

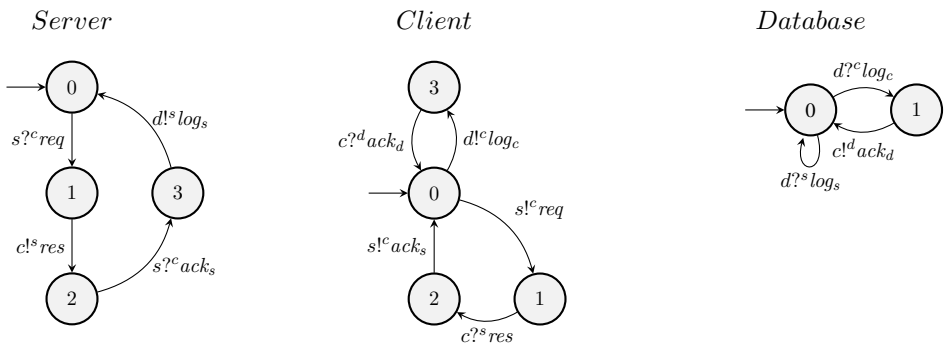


Figure 3.3: Client/Server/Database protocol

unmatched send. Therefore, the language of all RSC executions over a set of participant $\mathbb{P}$, a set of buffers $\mathbb{I}$ and a set of messages $\mathbb{V}$ is a subset of $(\Omega_{\mathbb{P},\mathbb{I},\mathbb{V}})^*$.

We will now show that the set of RSC executions of a system is regular. For that, we will define a way to build, for any system of communicating automata $\mathcal{A}_{rsc}$, a finite state automaton recognising RSC executions feasible in this system.

Definition 3.3.1 ($\mathcal{A}_{rsc}$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$.

Let $\mathcal{A}_{rsc}(\mathfrak{S}) = (\mathsf{Q}_{rsc}, \delta_{rsc}, \mathfrak{q}^0_{rsc}, \mathsf{F}_{rsc})$ be the deterministic finite state automaton over $\Omega_{\mathfrak{S}}$ with:

(1) $\mathsf{Q}_{rsc} = L_{\mathfrak{S}} \times 2^{\mathbb{I}^F_{\mathfrak{S}}} \times 2^{\left(\mathbb{I}^B_{\mathfrak{S}} \times \mathbb{V}_{\mathfrak{S}}\right)}$ its set of control states;

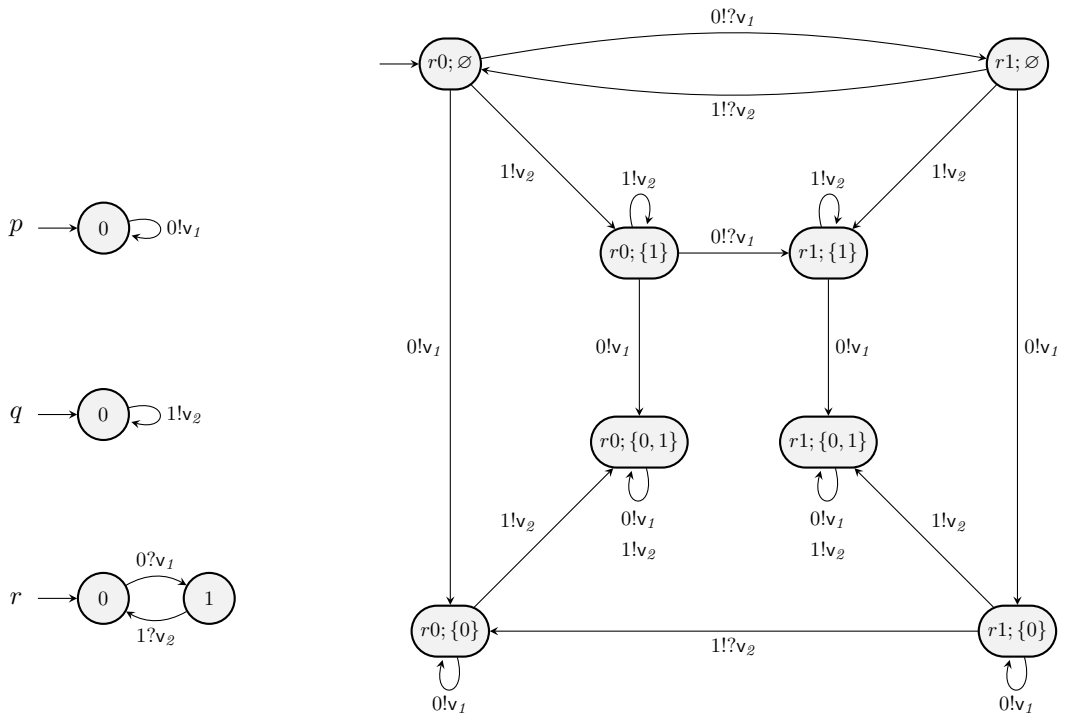
(2) $\mathfrak{q}^0_{rsc} = (\mathfrak{l}_0, \varnothing, \varnothing)$ its initial state;

(3) $\mathsf{F}_{rsc} = \mathsf{Q}_{rsc}$ its set of accepting states (all states are accepting);

(4) for $c \in \Omega_{\mathfrak{S}}$, $(\mathfrak{l}, \mathfrak{f}, \mathfrak{b}) \in \mathsf{Q}_{rsc}$, $(\mathfrak{l}', \mathfrak{f}', \mathfrak{b}') \in \mathsf{Q}_{rsc}$, $((\mathfrak{l}, \mathfrak{f}, \mathfrak{b}), c, (\mathfrak{l}', \mathfrak{f}', \mathfrak{b}')) \in \delta_{rsc}$ if:

- $(\mathfrak{l}, \mathbf{b}) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c)} (\mathfrak{l}', \mathbf{b}')$ for some $\mathbf{b}, \mathbf{b}'$, such that:
  - for all $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, $b_\iota \neq \varepsilon$ if and only if $\iota \in \mathfrak{f}$, and $b'_\iota \neq \varepsilon$ if and only if $\iota \in \mathfrak{f}'$ and
  - for all $\iota \in \mathbb{I}^B_{\mathfrak{S}}$, $\mathsf{v} \in b_\iota$ if and only if $(\iota, \mathsf{v}) \in \mathfrak{b}$ and $\mathsf{v} \in b'_\iota$ if and only if $(\iota, \mathsf{v}) \in \mathfrak{b}'$; and
- if $c = \iota!?\mathsf{v}$, $\iota \notin \mathfrak{f}$ and $(\iota, \mathsf{v}) \notin \mathfrak{b}$.

A control state of $\mathcal{A}_{rsc}$ encodes a control state of the system, the set of blocked FIFO buffers, and the set of blocked messages on a given bag buffers. Remembering the blocked FIFO buffers allows us to prevent occurrences of matched communications after an unmatched send in an execution: for a word $w = \iota!\mathsf{v} \cdot \iota!?\mathsf{v}$, the corresponding execution is $e = \iota!\mathsf{v} \cdot \iota!\mathsf{v} \cdot \iota?\mathsf{v}$ where the reception matches the first send action. As the second send action happens in between the send action and reception of this matching pair, execution e is not RSC. The second condition of point (4) in Definition 3.3.1 ensures that w is not recognised by $\mathcal{A}_{rsc}$: after reading the first $\iota!\mathsf{v}$, $\mathcal{A}_{rsc}$ reaches a state where buffer ι is marked as blocked, from which $\iota!?\mathsf{v}$ cannot be read. The reasoning is the same for the pairs bag buffer/message, $\mathcal{A}_{rsc}$ prevents a matched communication of message v



(a) Representation of system $\mathfrak{S}_{rsc}$

(b) Representation of $\mathcal{A}_{rsc}(\mathfrak{S})$

Figure 3.4: Representation of an RSC system and its automaton $\mathcal{A}_{rsc}$

in a bag buffer ι if an unmatched send of this message happened previously in ι. This is to ensure that the reception of a matched communication matches the send action it is paired with, and not an earlier one, according to Definition 2.3.3.

Example 3.3.1 – Let $\mathfrak{S}_{rsc}$ be a system of three communicating automata. Two of them send messages v_1 and v_2 respectively, to different FIFO buffers, and the third one alternates between receptions of v_1 and v_2. It is depicted in Figure 3.4a, and Figure 3.4b is the visual representation of $\mathcal{A}_{rsc}(\mathfrak{S}_{rsc})$. As the communicating automata of participants p and q have only one control state, representing the state of r is enough to encode the global control state of the system. Similarly, as no buffer is a bag, there is no need for remembering the blocked pairs of messages in bag buffers. We can visually confirm that from the moment an unmatched send action happens in an execution, no further matched communication (hence reception) is possible.

The next example illustrates the fact that systems that are not RSC can have RSC executions. Automaton $\mathcal{A}_{rsc}$ can be built for any system, and may accept some executions for non-RSC systems. However, the causal closure of the language of $\mathcal{A}_{rsc}$ will not be the language of all executions of a non-RSC system.

Example 3.3.2 – Let $\mathfrak{S}_n$ be the system of communicating automata depicted in Figure 3.1a. Figure 3.5 represents $\mathcal{A}_{rsc}(\mathfrak{S}_n)$. Even though $\mathfrak{S}_n$ is not RSC, it admits some RSC executions. For instance, using the upmost transition available at each time, we see that the word $w = 0!\mathsf{v} \cdot 1!?\mathsf{v}'$ is recognised by $\mathcal{A}_{rsc}(\mathfrak{S}_n)$, meaning that $e = 0!\mathsf{v} \cdot 1!\mathsf{v}' \cdot 1?\mathsf{v}'$ is an RSC execution feasible in $\mathfrak{S}_n$.

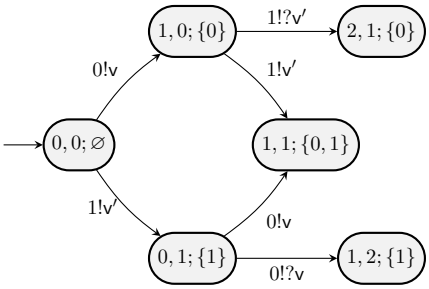


Figure 3.5: Representation of $\mathcal{A}_{rsc}$ for a non-RSC system

However, for $e_n = 0!^p\mathsf{v} \cdot 1!^q\mathsf{v}' \cdot 1?^p\mathsf{v}' \cdot 0?^q\mathsf{v}$, no execution $e' \in [\![e_n]\!]_\sim$ is recognised by $\mathcal{A}_{rsc}(\mathfrak{S}_n)$. This is because e_n is not causally equivalent to any RSC execution.

We defined $\mathcal{A}_{rsc}$ so that it recognises all sequences of communications that the transitions of a system allow. The only constraint we added was to prevent a matched communication to happen after an unmatched send in a FIFO buffer, of after an unmatched send implying the same message in a bag buffer. Such situations never occur in a feasible RSC execution, so this constraint does not prevent any RSC execution of the studied system to be recognised. We state formally that $\mathcal{A}_{rsc}$ is complete: it recognises all RSC executions of a system.

Lemma 3.3.1. *Let $\mathfrak{S}$ be a system of communicating automata, and let $e \in \mathsf{executions}(\mathfrak{S})$. If e is RSC, then $\mathsf{etc}(e) \in \mathcal{A}_{rsc}(\mathfrak{S})$.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that its product is $\mathsf{product}(\mathfrak{S}) = (L_\mathfrak{S}, \mathbb{V}_\mathfrak{S}, \mathbb{I}_\mathfrak{S}, \mathsf{Act}_\mathfrak{S}, \delta_\mathfrak{S}, \mathbf{l}^0_\mathfrak{S})$. Let $\mathcal{A}_{rsc}(\mathfrak{S}) = (\mathsf{Q}_{rsc}, \delta_{rsc}, \mathsf{q}^0_{rsc}, \mathsf{F}_{rsc})$ be the finite state automaton as in Definition 3.3.1. Let $e \in \mathsf{executions}(\mathfrak{S})$ be an RSC execution, and $w = \mathsf{etc}(e)$: we know w is defined because e is RSC. We will show that $w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$.

Let $w = c_1 \cdot \ldots \cdot c_m$, we have:

$$\left(\mathbf{l}^0, \mathbf{b}^\varnothing\right) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c_1)} (\mathbf{l}_1, \mathbf{b}_1) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c_2)} \cdots \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c_m)} (\mathbf{l}_m, \mathbf{b}_m).$$

For all $j \in \{1, \ldots, m\}$, let $\mathfrak{f}_j = \left\{\iota \mid \iota \in \mathbb{I}^F_\mathfrak{S}, b_\iota \neq \varepsilon, b_\iota \text{ the } \iota^{th} \text{ component of } \mathbf{b}_j\right\}$, and let $\mathfrak{b}_j = \left\{(\iota, \mathsf{v}) \mid \iota \in \mathbb{I}^B_\mathfrak{S}, \mathsf{v} \in \mathbb{V}_\mathfrak{S}, \mathsf{v} \in b_\iota, b_\iota \text{ the } \iota^{th} \text{ component of } \mathbf{b}_j\right\}$. Because e is RSC, for all $j \in \{1, \ldots, m\}$, if $c_j = \iota!?\mathsf{v}$ then the two actions of $\mathsf{cte}(c_j)$ form a matching pair in e, so we know that b_ι the ι^{th} component of $\mathbf{b}_{j-1}$ is empty if it is FIFO, and does not contain message v if it is bag (otherwise, there would be one more send than receive action on this buffer, contradicting the condition for actions to be in the same matching pair given in Definition 2.3.3). Therefore, we can associate to each configuration $(\mathbf{l}_j, \mathbf{b}_j)$, for $j \in \{1, \ldots, m\}$, a control state $(\mathbf{l}_j, \mathfrak{f}_j, \mathfrak{b}_j) \in \mathsf{Q}_{rsc}$, and the conditions of (4) in Definition 3.3.1 are satisfied such that $((\mathbf{l}_{j-1}, \mathfrak{f}_{j-1}, \mathfrak{b}_{j-1}), c_j, (\mathbf{l}_j, \mathfrak{f}_j, \mathfrak{b}_j)) \in \delta_{rsc}$, proving our point. □

In the next lemma, we state that all executions accepted by $\mathcal{A}_{rsc}(\mathfrak{S})$ are indeed RSC executions feasible in $\mathfrak{S}$.

Lemma 3.3.2. *Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata. For all $w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$, $\mathsf{cte}(w) \in \mathsf{executions}(\mathfrak{S})$.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. Let $\mathcal{A}_{rsc}(\mathfrak{S}) = (\mathsf{Q}_{rsc}, \delta_{rsc}, \mathsf{q}^0_{rsc}, \mathsf{F}_{rsc})$ be the finite state automaton as in Definition 3.3.1. We proceed by induction on the length of w.

Let $w = c_1 \cdot \ldots \cdot c_n$ in $\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$, and $\{(\mathbf{l}_0, \mathfrak{f}_0, \mathfrak{b}_0), \ldots, (\mathbf{l}_n, \mathfrak{f}_n, \mathfrak{b}_,)\} \subseteq \mathsf{Q}_{rsc}$ the control states of $\mathcal{A}_{rsc}(\mathfrak{S})$ such that for all $j \in \{1, \ldots, n\}$, $((\mathbf{l}_{j-1}, \mathfrak{f}_{j-1}, \mathfrak{b}_{j-1}), c_j, (\mathbf{l}_j, \mathfrak{f}_j, \mathfrak{b}_j)) \in \delta_{rsc}$. Let $e = \mathsf{cte}(w)$, and assume by induction that e is RSC and that there exists $\mathbf{b}$ such that $(\mathbf{l}^0, \mathbf{b}^\varnothing) \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} (\mathbf{l}_n, \mathbf{b})$ with for all $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, $b_\iota = \varepsilon$ if and only if $\iota \notin \mathfrak{f}_n$, and for all $\iota \in \mathbb{I}^B_{\mathfrak{S}}$, for all $\mathsf{v} \in \mathbb{V}_{\mathfrak{S}}$, $\mathsf{v} \in b_\iota$ if and only if $(\iota, \mathsf{v}) \in \mathfrak{b}_n$.

We will show that if there exists $c \in \Omega_{\mathbb{P},\mathbb{I},\mathbb{V}}$ such that $((\mathbf{l}_n, \mathfrak{f}_n, \mathfrak{b}_n), c, (\mathbf{l}_{n+1}, \mathfrak{f}_{n+1}, \mathfrak{b}_{n+1}))$, there exists $\mathbf{b}'$ such that $(\mathbf{l}^0, \mathbf{b}^\varnothing) \xRightarrow[\mathfrak{S}]{e\cdot\mathsf{cte}(c)} (\mathbf{l}_{n+1}, \mathbf{b}')$, with for all $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, $b'_\iota = \varepsilon$ if and only if $\iota \notin \mathfrak{f}_{n+1}$, and for all $\iota \in \mathbb{I}^B_{\mathfrak{S}}$, $\mathsf{v} \in b'_\iota$ if and only if $(\iota, \mathsf{v}) \in \mathfrak{b}_{n+1}$.

We know by Definition 3.3.1 that there exists $\mathbf{b}^{pre}, \mathbf{b}^{post}$ such that $(\mathbf{l}_n, \mathbf{b}^{pre}) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c)} (\mathbf{l}_{n+1}, \mathbf{b}^{post})$.

- If $c = \iota!\mathsf{v}$, then we know from Definition 2.2.8 that $\mathsf{v} \in b^{post}_\iota$, so $b^{post}_\iota \neq \varepsilon$. By the same definition, since there are no restrictions on the content of the buffers before sending a message, we know that $(\mathbf{l}_n, \mathbf{b}) \xrightarrow[\mathfrak{S}]{\iota!\mathsf{v}} (\mathbf{l}_{n+1}, \mathbf{b}')$ for some $\mathbf{b}'$, and if $\iota \in \mathbb{I}^F_{\mathfrak{S}}, \iota \in \mathfrak{f}_{n+1}$ and $b'_\iota \neq \varepsilon$, else $(\iota, \mathsf{v}) \in \mathfrak{b}_{n+1}$, with $\mathsf{v} \in b'_\iota$. Because e was RSC, $e \cdot \iota!\mathsf{v}$ is still RSC: there are no additional matching pairs, and we added the action at the end of the execution so no matching pair is split.
- If $c = \iota!?\mathsf{v}$, we know that $\iota \notin \mathfrak{f}_n$ and that $(\iota, \mathsf{v}) \notin \mathfrak{b}_n$, so either
  - $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ and $b_\iota = \varepsilon$, or
  - $\iota \in \mathbb{I}^B_{\mathfrak{S}}$ and $\mathsf{v} \notin b_\iota$.

  We can deduce from Definition 2.2.8 that if $b^{pre}_\iota = b_\iota$, $(\mathbf{l}, \mathbf{b}^{pre}) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(\iota!?\mathsf{v})} (\mathbf{l}', \mathbf{b}^{post})$ implies $(\mathbf{l}, \mathbf{b}) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(\iota!?\mathsf{v})} (\mathbf{l}', \mathbf{b}')$ for some $\mathbf{b}'$. Moreover, as either $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ and $b_\iota = \varepsilon$ or $\iota \in \mathbb{I}^B_{\mathfrak{S}}$ and $\mathsf{v} \notin b_\iota$, the reception $\iota?\mathsf{v}$ matches $\iota!\mathsf{v}$, so $e \cdot \iota!\mathsf{v} \cdot \iota?\mathsf{v}$ is RSC.

In both cases above, as other buffers than ι are unchanged by a transition, for all $j \in \mathbb{I}_{\mathfrak{S}} \setminus \{\iota\}$, $b_j = \varepsilon$ if and only if $b'_j = \varepsilon$, so $j \in \mathfrak{f}_{n+1}$ if and only if $b'_j = \varepsilon$. We showed that $e \cdot \mathsf{cte}(c) \in \mathsf{executions}(\mathfrak{S})$, and that for $\iota \in \mathbb{I}_{\mathfrak{S}}$, $b'_\iota = \varepsilon$ if and only if $\iota \in \mathfrak{f}_{n+1}$. Because a trivial base case is both a word $w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$ and an execution $e \in \mathsf{executions}(\mathfrak{S})$ of length 0, this shows the claim. □

Because we have shown that the way $\mathcal{A}_{rsc}$ is defined for a system $\mathfrak{S}$, it recognises all and only the RSC executions of $\mathfrak{S}$, we can show the following theorem. It states that the set of RSC executions of a system is regular.

Theorem 3.3.3. *Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. The language of RSC executions feasible in $\mathfrak{S}$*

Figure 3.6: RSC system with non regular execution set

is regular, and a deterministic finite state automaton recognising it is computable in time $\mathcal{O}\left(|L_{\mathfrak{S}}| \cdot 2^{|\mathbb{I}^F_{\mathfrak{S}}|+|\mathbb{I}^B_{\mathfrak{S}}\times\mathbb{V}_{\mathfrak{S}}|} \cdot |\mathbb{V}_{\mathfrak{S}}| \cdot |\mathbb{P}|^2 \cdot |\mathbb{I}_{\mathfrak{S}}|\right)$.

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$. By Lemma 3.3.2, we know that $\mathsf{cte}(\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})))$ is the language of RSC executions of $\mathfrak{S}$.

The complexity results comes from the fact that each transition can be built in constant time, so we show that the result we stated is an upper bound to the number of transitions in $\mathcal{A}_{rsc}(\mathfrak{S})$. The number of states of $\mathcal{A}_{rsc}(\mathfrak{S})$ is $|\mathsf{Q}_{rsc}| = |L_{\mathfrak{S}}| \cdot 2^{|\mathbb{I}^F_{\mathfrak{S}}|} \cdot 2^{|\mathbb{I}^B_{\mathfrak{S}}\times\mathbb{V}_{\mathfrak{S}}|}$. From any state $(\mathfrak{l}, \mathfrak{f}, \mathfrak{b})$, a bound to the number of outgoing transitions is the number of communications, that is $|\mathbb{V}_{\mathfrak{S}}| \cdot |\mathbb{P}| \cdot |\mathbb{I}_{\mathfrak{S}}|$ unmatched communications and $|\mathbb{V}_{\mathfrak{S}}| \cdot |\mathbb{P}|^2 \cdot |\mathbb{I}_{\mathfrak{S}}|$ matched ones, multiplied by the number of destination states: in the worst case, $|\mathsf{Q}_{rsc}|$. Therefore, $|\delta_{rsc}| \leq \left(|\mathsf{Q}_{rsc}|^2 \cdot 2\,|\mathbb{V}_{\mathfrak{S}}| \cdot \left(|\mathbb{P}|^2 + |\mathbb{P}|\right) \cdot 2\,|\mathbb{I}_{\mathfrak{S}}|\right)$. □

For a system $\mathfrak{S}$, we will write $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S})$ for $\mathsf{cte}(\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})))$, that is to say the set of all RSC executions feasible in the system. Because in RSC systems, all executions are causally equivalent to an RSC execution, for such a system $\mathfrak{S}$ the causal closure of $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S})$ is equal to $\mathsf{executions}(\mathfrak{S})$. It is important to note that the set of all executions of an RSC system is not necessarily regular. For instance, if we take the system $\mathfrak{S}$ as depicted in Figure 3.6, the following is not regular:

$$\mathsf{executions}(\mathfrak{S}) \cap ((1!^p\mathsf{v})^* \cdot (1?^q\mathsf{v})^*) = \{(1!^p\mathsf{v})^n \cdot (1?^q\mathsf{v})^m \,|n \geq m\}\,.$$

3.4 Discussion

We begin with a comparison between our definition of the class of RSC systems and other notions from the literature. We focus on the similarities and differences between these notions and RSC systems, leaving the discussion on decidability results for the next chapter.

As far as we know, the definition of the class of RSC systems of communicating automata was introduced in [Di Giusto, Germerie Guizouarn and É. Lozes 2021]*, as well as the proof of the regularity of the set of RSC executions of a system. However, RSC *computations* were introduced in [Charron-Bost et al. 1996], which is the first paper to explore the notion of *realisability with synchronous communication*. A 'computation' is the equivalent of our action graphs, that is a partial order on actions of an execution. RSC computations are defined as the partial orders admitting a linear extension where all matching actions are not separated by another action. In this thesis, we defined RSC executions, that is total orders on actions, as the ones satisfying the same

*In this work this notion was named 'greedy', it was renamed RSC in [Di Giusto, Germerie Guizouarn and E. Lozes 2023])

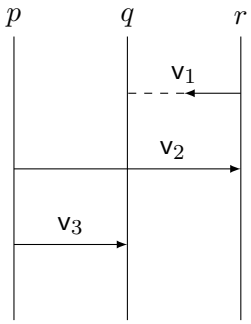


Figure 3.7: MSC with straight lines that is not RSC with a mailbox architecture

condition: matching actions must be paired together. An execution corresponds to one linearisation of a computation, so if RSC computations are more accurately comparable with causally RSC executions, the executions that are causally equivalent to an RSC execution.

Charron-Bost et al. proposed two alternative characterisations of RSC computations. The first one is called the 'crown criterion', and it says that a computation is RSC if and only if it has no crown. A *crown* is defined on computations, so on a partial order of the actions of an execution. We transcript this definition to our setting: an execution $e = a_1 \cdot \ldots \cdot a_n$ contains a crown if there exists $\{s_1, r_1, \ldots, s_m, r_m\} \subseteq \{1, \ldots, n\}$ such that for all $i \in \{1, \ldots, m\}$, $\{s_i, r_i\}$ is a matching pair of e, and $s_1 \prec_e r_2, s_2 \prec_e r_3, \ldots, s_m \prec_e r_1$. Because they consider a peer-to-peer architecture, for $s_i \prec_e r_{i+1}$ to hold, it means that a_{s_i} and $a_{r_{i+1}}$ are done by the same participant. A crown imposes that there are two matching pairs in the computation such that both the receptions require both send actions to be executed before happening. The way Charron-Bost et al. prove that the absence of crowns characterises causally RSC executions is interesting, as it relies on a construct very similar to conflict graphs. They show that a crown is present in a computation if and only if there is a cycle in this graph. This characterisation is the same as the one we give from Section 3.1.2.

The last RSC characterisation from [Charron-Bost et al. 1996] is a graphical one. This work defines 'space time diagrams', which are essentially MSCs. They are graphical representations of the computations, which are themselves partial orders of actions. If we express this characterisation in the terms we defined in the present work, it says that an execution is RSC if its MSC can be drawn without crossing arrows. Note that this condition applies to peer-to-peer architecture only. With a mailbox architecture, it is not a sufficient condition (although it is a necessary one). Figure 3.7 gives an example of such an MSC with straight lines which does not have an RSC linearisation with mailbox architecture. Indeed, the message v_1 is sent in the same buffer as message v_3, so in order for the latter to be received, it has to be sent before v_1, meaning v_2 cannot be received right after it was sent.

Half-duplex systems [Cécé and Finkel 2005] are a class of binary systems of communicating automata. If we say that *binary RSC systems* are the binary systems that are RSC, half-duplex and binary RSC are really close notions. We will discuss their relationship in more detail in Chapter 5.

Another class of communicating automata which is close to RSC is *eager* systems. In [Heußner, Leroux et al. 2012], eager 'runs' (executions in our setting) are defined in the same way as our RSC executions. The authors defined a class of *eager systems*, but their characterisation differs from ours: indeed, for a system to be eager, all its reachable configurations must be reachable through an eager run (for clarity, we will refer to eager runs as RSC executions from now on as they are the same, and we keep the term 'eager' to refer to systems). The key difference between eager and RSC

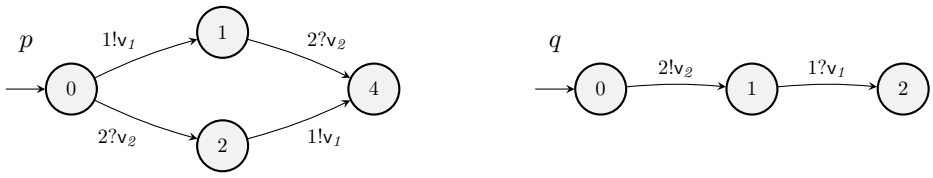


Figure 3.8: Example of eager system that is not RSC

systems is that there might be executions that are not causally equivalent to an RSC execution in an eager system. An RSC system is clearly eager: as all its executions are equivalent to an RSC one, every reachable configuration is reachable through an RSC execution. However, an eager system is not necessarily RSC, because two executions that are not causally equivalent may lead to the same configuration. Example 3.4.1 below illustrates this difference.

Example 3.4.1 – Let $\mathfrak{S}$ be the system of communicating automata depicted in Figure 3.8. Observe that any of its reachable configurations can be reached through an RSC execution. Consider the following execution: $e = 1!^{p}\mathsf{v}_1 \cdot 2!^{q}\mathsf{v}_2 \cdot 1?^{q}\mathsf{v}_1 \cdot 2?^{p}\mathsf{v}_2$. It is not causally RSC, implying that $\mathfrak{S}$ is not RSC. This system is eager, because the configuration γ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Longrightarrow}} \gamma$ is reachable through execution $e' = 2!^{q}\mathsf{v}_2 \cdot 2?^{p}\mathsf{v}_2 \cdot 1!^{p}\mathsf{v}_1 \cdot 1?^{q}\mathsf{v}_1$, which is RSC. However, $e \not\sim e'$, which is easy to see as participant p does not execute the same transitions in these executions.

Some classes are not defined as closely to RSC as the one we just discussed, but come from the same idea of characterising behaviours that are almost synchronous. One of them is the *synchronisable systems* from [Basu, Bultan and Ouederni 2012b]. A system is synchronisable if its synchronous behaviour is the same as its asynchronous behaviour. In this paper, the equivalence between behaviours of the systems is defined as the equivalence between traces of send actions. This notion of synchronisability was later extended to *stability* [Akroun and Salaün 2018; Akroun, Salaün and Ye 2016]. A system is k stable if its behaviour with its buffers bounded to k is the same as its behaviour with any bound $k' > k$.

Toward the same goal of characterising the synchronisability of systems, another class close to RSC is the k-synchronisability [Bouajjani, Enea et al. 2018a]. A system is k-synchronisable if its executions can be organised as sequences of k-*exchanges*. A k-exchange is a sequence of actions in which at most k messages are sent, and then some receptions may happen for these messages, without overlap between the sending and receiving phases. RSC systems are 1-synchronisable, but the converse is not true [Di Giusto, Laversa et al. 2020, Example 1.2]. However a refinement of the notion of k-synchronisability, named *strong k-synchronisability* [Bollig, Giusto et al. 2021], excludes this example. The class of strong 1-synchronisable systems is equivalent to the class of RSC systems.

Several notions of buffer boundedness were studied. In [Lohrey and Muscholl 2004], boundedness was studied for MSCs and languages of MSCs, and in [Genest, Kuske et al. 2007] these results were applied to communicating automata. Boundedness may be *local*, that is applied to each buffer individually, or *global*, that is considering the total amount of messages in all the buffers combined. An execution is k-*bounded* if at any step (that is, in any configuration reached by a prefix of the execution), there are at most k messages in either any buffer (local boundedness), or in all the buffers (global boundedness). For systems of communicating automata, or languages of executions, the two main notions of boundedness are *universal* and *existential* boundedness. A system is universally k-bounded if all its executions are k-bounded. A system is existentially k-bounded if

all its executions can be rearranged to be k-bounded. There are two differences between RSC and existentially 1-bounded systems:

1. in an RSC execution, we allow buffers to grow indefinitely, provided that no message is received from them.

2. in a 1-bounded execution, a single message may have to wait in a buffer before being received. The system from Example 3.4.1 is not RSC but is existentially 1-bounded.

Note that an RSC system without unmatched messages is existentially 1-bounded.

Bounded context-switching reachability was introduced in [Torre et al. 2008]. Intuitively, a *context* is a sequence during which a single participant is active, and it can only receive messages from one buffer (but can send messages to any buffer). A bounded context-switching execution is one that is a sequence of a bounded amount of contexts. If the idea of bounded interactions makes this class similar to the ones we discussed here, the intuition of bounded-context switching reachability is the opposite of the one of RSC reachability. For an execution to be RSC, the matching send and reception must be grouped together, while to minimise the number of context switches in an execution, all the actions of a participant should be grouped together if possible.

An interesting aspect of the characterisation of RSC executions is how it relies on matching pairs. This makes the notion versatile: as long as what actions form a matching pair in an execution is well defined in the semantics used, the RSC notion is defined as well. This is what allowed us to adapt the results we had for FIFO systems to systems with bag buffers.

With fully-bag matching pairs, the definition of an RSC system is less clear. In fact, let us consider the following execution: $e = \iota!^p\mathsf{v} \cdot \iota!^p\mathsf{v} \cdot \iota?^p\mathsf{v}$ (the only purpose of participant p receiving messages in the same buffer in this example is to shorten the example). If we follow the fully-bag definition of matching pairs and causal equivalence, the reception could match either send actions. Execution e is RSC only if the second send action is matched by the reception. So there is one action graph of e for which this execution is not RSC, and one for which it is, meaning that if the system is capable of producing both interpretations of the executions, it is not RSC according to Definition 3.2.1. Intuitively, a system is RSC with fully-bag matching pairs if all the conflict graphs of all the distributions of matching pairs of all its executions are acyclic. We leave membership of RSC with fully-bag buffers as an open problem.

CHAPTER 4

Model-checking

In general, model-checking consists in verifying whether a given model satisfies some properties. In this chapter, we will tackle two model-checking problems for systems of communicating automata. The first one is the *membership problem* for the class of RSC systems, and the second one is the *reachability problem* of a regular set of configurations through an RSC execution.

4.1 Membership

We call *membership problem* the decision problem consisting in checking whether a system is RSC or not. We say that an RSC system is *member of the class* of RSC systems. We will rely on a technique originally used in [Bouajjani, Enea et al. 2018a] to show that this problem is decidable. The idea is to characterise a regular subset of non-RSC executions, such that a system admits at least one of them if it is not RSC, and none if it is. We call these executions *borderline violations*.

4.1.1 Borderline violations

A borderline violation is an execution that is RSC until its last action, which prevents it from being causally equivalent to any RSC execution.

Definition 4.1.1 (Borderline violation)**.** Let $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ be an alphabet of communications. A borderline violation is an execution $e = e' \cdot \iota?^{p}\mathsf{v}$ over $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ such that e' is RSC, and e is not causally equivalent to an RSC execution.

In a borderline violation, the last reception $\iota?\mathsf{v}$ matches a send action $\iota!\mathsf{v}$ in e', with $\iota!\mathsf{v}$ not being the last action of e' (because otherwise the last reception would follow immediately its matching send action, and as e' is RSC, e would be RSC as well). We say that such an execution is minimally non-RSC because removing its last action yields an RSC execution.

Example 4.1.1 – Execution $e_n = 0!^{p}\mathsf{v} \cdot 1!^{q}\mathsf{v}' \cdot 1?^{p}\mathsf{v}' \cdot 0?^{q}\mathsf{v}$, already studied in Example 3.1.2, is a borderline violation. Indeed, $0!^{p}\mathsf{v} \cdot 1!^{q}\mathsf{v}' \cdot 1?^{p}\mathsf{v}'$ is RSC: the only matching pair is exchanging v', and the reception of this message immediately follows its send action. The final reception makes e_n not causally equivalent to an RSC execution.

Note that, by Lemma 3.1.2, the conflict graph of a borderline violation is cyclic. Moreover, we know that removing the last reception of a borderline violation makes it RSC, breaking all cycles in the conflict graph. So the communication containing the last reception is part of any cycle of the conflict graph of a borderline violation.

The claim of the following lemma is that if a system is not RSC, there is at least one borderline violation in its executions.

Lemma 4.1.1. *A system $\mathfrak{S}$ is RSC if and only if* $\mathsf{executions}(\mathfrak{S})$ *contains no borderline violation.*

Proof.

Let $\mathfrak{S}$ be a system such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. Clearly, if $\mathsf{executions}(\mathfrak{S})$ contains a borderline violation, $\mathfrak{S}$ is not RSC. Conversely, assume that $\mathfrak{S}$ is not RSC, and let us show that $\mathsf{executions}(\mathfrak{S})$ contains a borderline violation.

Let $e \in \mathsf{executions}(\mathfrak{S})$ be an execution that is not causally equivalent to any RSC execution and of minimal length among all such executions. Then $e = e_1 \cdot a$ with e_1 causally equivalent to an RSC execution, otherwise e_1 would contradict the minimality of the length of e. Let e'_1 be an RSC execution causally equivalent to e_1. Then $e' = e'_1 \cdot a \in \mathsf{executions}(\mathfrak{S})$, because by Lemma 2.3.1, e_1 and e'_1 lead to the same configuration. Moreover,we know that a is a reception, otherwise e' is RSC, contradicting the fact that e is not causally equivalent to an RSC execution. Therefore, e' is a borderline violation. □

We say that the borderline violations of a system are all the borderline violations feasible in the system. We will show that this set is regular for any system of communicating automata. To do so, we will rely on the regularity of the set of RSC execution of a system, and on a relaxation of the definition of borderline violations: *candidate borderline violations*.

Definition 4.1.2 (Candidate borderline violation)**.** Let $\Omega_{\mathbb{P},\mathbb{I},\mathbb{V}}$ be an alphabet of communications, and $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ an alphabet of actions. An execution e over $\Lambda_{\mathbb{P},\mathbb{I},\mathbb{V}}$ is a *candidate borderline violation* if there exists $w \in (\Omega_{\mathbb{P},\mathbb{I},\mathbb{V}})^*, c \in \Omega_{\mathbb{P},\mathbb{I},\mathbb{V}}, p \in \mathbb{P}, \iota \in \mathbb{I}, \mathsf{v} \in \mathbb{V}$ such that

(1) $e = \mathsf{cte}(w) \cdot \mathsf{cte}(c) \cdot \iota?^p\mathsf{v}$,

(2) $\iota!^p\mathsf{v} \in \mathsf{cte}(w)$, and

(3) if $\mathsf{cte}(w \cdot c)$ is RSC, then e is a borderline violation.

Intuitively, a candidate borderline violation is an execution that is built like a borderline violation, but might not be one. It is an execution obtained by concatenation of communications, ending with a single reception, whose buffer and message match the ones of at least one earlier unmatched communication. A candidate borderline violation might not be a borderline violation in the case where the concatenation of communications is not RSC: this happens when these communications do not correspond to the matching pairs of the execution (see Example 3.1.3). Example 4.1.2 below illustrates this situation.

Example 4.1.2 – Let $e = \mathsf{cte}(\iota!^p\mathsf{v} \cdot \iota!^p?^q\mathsf{v}) \cdot \iota?^q\mathsf{v}$, e is a candidate borderline violation: let $w = \iota!^p\mathsf{v}$, and $c = \iota!^p?^q\mathsf{v}$, then $e = \mathsf{cte}(w) \cdot \mathsf{cte}(c) \cdot \iota?^q\mathsf{v}$; $\mathsf{cte}(w) \cdot \mathsf{cte}(c)$ is not RSC, so the implication in condition (3) of Definition 4.1.2 trivially holds. However, e is not a borderline violation: its prefix $\mathsf{cte}(w \cdot c)$ is not RSC, so e does not satisfy the first condition in Definition 4.1.1. In fact, observe that e is actually causally equivalent to an RSC execution: $e' = \iota!^p\mathsf{v} \cdot \iota?^q\mathsf{v} \cdot \iota!^p\mathsf{v} \cdot \iota?^q\mathsf{v}$.

Condition (3) is there to ensure that the only situation where a candidate borderline violation is not actually a borderline violation is when its matching pairs are not formed by actions belonging to the same communication in the sequence it was obtained from. An example where this condition is not met is given below.

Example 4.1.3 – Let $e = \iota!^p\mathsf{v}_1 \cdot \iota'!^q\mathsf{v}_2 \cdot \iota'?^r\mathsf{v}_2 \cdot \iota?^r\mathsf{v}_1$ be an execution satisfying items (1) and (2) of Definition 4.1.2. Because of the condition in item (3) of Definition 4.1.2, this execution is *not* a candidate borderline violation: its prefix $\iota!^p\mathsf{v}_1 \cdot \iota'!^q\mathsf{v}_2 \cdot \iota'?^r\mathsf{v}_2$ is RSC, but e is not a borderline violation. Indeed, e is causally equivalent to $e' = \iota'!^q\mathsf{v}_2 \cdot \iota'?^r\mathsf{v}_2 \cdot \iota!^p\mathsf{v}_1 \cdot \iota?^r\mathsf{v}_1$, which is RSC.

4.1.2 Automaton $\mathcal{A}_{bv}$

We define a non deterministic finite state automaton $\mathcal{A}_{bv}$ that will be able to recognise some candidate borderline violations, among which all the actual borderline violations over an alphabet of actions. It does so by recognising executions formed by concatenation of communications, among which it selects non-deterministically an unmatched send. The words it accepts end with a reception $\iota?\mathsf{v}$ matching the selected send action.

If the communications of a recognised word w are the matching pairs of $\mathsf{cte}\,(w)$, there is a cycle in $\mathsf{cgraph}\,(\mathsf{cte}\,(w))$. This is because $\mathcal{A}_{bv}$ ensures there is a sequence of conflicting communications from the selected send action and the final reception. As illustrated in Figure 4.1, the communication formed by the final reception and its matching send action close a cycle in $\mathsf{cgraph}\,(e)$. Notice however that if the matching pairs of e are not formed by actions of the same communication in the recognised word, there might not be a cycle in the conflict graph.

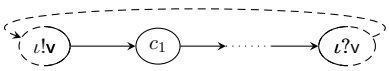


Figure 4.1: Illustration of a cycle in a conflict graph ensured by $\mathcal{A}_{bv}$.

In what follows, by a slight abuse of notation, for a word $w = w' \cdot \iota?^p\mathsf{v}$ over an alphabet $\Lambda \cup R_{\mathbb{P},\mathbb{I},\mathbb{V}}$, we write $\mathsf{cte}\,(w)$ for $\mathsf{cte}\,(w') \cdot \iota?^p\mathsf{v}$.

Definition 4.1.3 ($\mathcal{A}_{bv}$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system such that its product is $\mathsf{product}\,(\mathfrak{S}) = \left(L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}}\right)$, $\mathcal{A}_{bv}\,(\mathfrak{S}) = \left(Q_{bv}, \delta_{bv}, (\varnothing, \varnothing), \left\{\mathfrak{q}^f_{bv}\right\}\right)$ is the non-deterministic finite state automaton over $\Omega_{\mathfrak{S}} \cup R_{\mathbb{P},\mathbb{I},\mathbb{V}}$ such that

$$\mathsf{Q}_{bv} = \left(2^{\mathbb{I}^F} \times 2^{\mathbb{I}^B \times \mathbb{V}}\right) \cup (\mathbb{I}_{\mathfrak{S}} \times \mathbb{V}_{\mathfrak{S}} \times \mathbb{P}) \cup (\mathbb{I}_{\mathfrak{S}} \times \mathbb{V}_{\mathfrak{S}} \times \Omega_{\mathfrak{S}}) \cup \left\{\mathfrak{q}^f_{bv}\right\},$$

and for all $(c, c') \in (\Omega_{\mathfrak{S}})^2$, for all $\iota \in \mathbb{I}_{\mathfrak{S}}, \mathsf{v} \in \mathbb{V}_{\mathfrak{S}}$:

(1) $((\mathfrak{f}, \mathfrak{b}), \iota!?\mathsf{v}, (\mathfrak{f}, \mathfrak{b})) \in \delta_{bv}$ if $\iota!?\mathsf{v} \in \Omega_{\mathfrak{S}}$

(2) $((\mathfrak{f}, \mathfrak{b}), \iota!\mathsf{v}, (\mathfrak{f}', \mathfrak{b}')) \in \delta_{bv}$ if $\iota!\mathsf{v} \in \Omega_{\mathfrak{S}}$, and either

- $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ and $\mathfrak{f}' = \mathfrak{f} \cup \{\iota\}$, $\mathfrak{b} = \mathfrak{b}'$; or
- $\iota \in \mathbb{I}^B_{\mathfrak{S}}$ and $\mathfrak{b}' = \mathfrak{b} \cup \{(\iota, \mathsf{v})\}$, and $\mathfrak{f} = \mathfrak{f}'$

(3) $((\mathfrak{f}, \mathfrak{b}), \iota!^p\mathsf{v}, (\iota, \mathsf{v}, p)) \in \delta_{bv}$ if $\iota \notin \mathfrak{f}$ and $(\iota, \mathsf{v}) \notin \mathfrak{b}$

(4) $((\iota, \mathsf{v}, p), c, (\iota, \mathsf{v}, p)) \in \delta_{bv}$

(5) $((\iota, \mathsf{v}, p), c, (\iota, \mathsf{v}, c)) \in \delta_{bv}$ if either

- $p \in \mathsf{process}\,(c)$, or

- $\mathsf{buffer}(c) = \iota$ and either $\mathsf{buffer}(c) \in \mathbb{I}^F_{\mathfrak{S}}$ or $\mathsf{message}(c) = \mathsf{v}$

(6) $((\iota, \mathsf{v}, c), c', (\iota, \mathsf{v}, c)) \in \delta_{bv}$

(7) $((\iota, \mathsf{v}, c), c', (\iota, \mathsf{v}, c')) \in \delta_{bv}$ if either

- $\mathsf{process}(c') \cap \mathsf{process}(c) \neq \varnothing$, or
- $\mathsf{buffer}(c) = \mathsf{buffer}(c')$ and either $\mathsf{buffer}(c) \in \mathbb{I}^F_{\mathfrak{S}}$ or $\mathsf{message}(c) = \mathsf{message}(c')$

(8) $\left((\iota, \mathsf{v}, c), \iota?^p\mathsf{v}, \mathsf{q}^f_{bv}\right) \in \delta_{bv}$ if $\mathsf{process}(c) = p$.

The states of this automaton can be divided in three stages. The first one gathers control states of the form $(\mathfrak{f}, \mathfrak{b})$ where $\mathfrak{f}$ is a set of blocked FIFO buffers, and $\mathfrak{b}$ is a set of blocked combinations of buffer and messages. Within this stage, any sequence of communication can be recognised, but we remember the FIFO buffers in which an unmatched send occurred, and the messages and bag buffers of unmatched communications. This is why transitions defined in (1) prevent looping on unmatched send actions in a buffer, unless the buffer is already marked as blocked. Transitions from (2) allow to recognise an unmatched send from a first stage state, by marking the buffer as blocked if it is FIFO, or the pair buffer/message if the buffer is a bag.

Transitions defined in (3) go from the first to the second stage, selecting the send action $\iota!^p\mathsf{v}$ to be matched by the final reception. We make sure that the selected action happens in a buffer that was not blocked yet, otherwise the final reception would not match this send action.

States from the second stage are of the form (ι, v, p) where ι, v and p are respectively the buffer identifier, the message, and the actor of the send action that was non-deterministically selected to be matched by the final reception. From each state of the second stage, we can read any communication through transitions from (4). This time, unmatched send actions are included in the loop.

A state (ι, v, c) from the third stage keeps the information about the selected send action, except from the actor which is replaced by the last communication taking part in the cycle we are building in the conflict graph. To get to this stage, a communication that is causally dependant on the selected unmatched send must be read. Therefore, a transition from (5) going from the second to the third stage is possible only if it is labelled by a communication c that either has p as one of its actors, or acts on buffer ι, with either ι being a FIFO buffer, or the message of c is the same as the selected one. These conditions ensure that $\iota!^p\mathsf{v}$ and $\mathsf{cte}(c)$ do not commute. There is always a send action in a communication, and remember that actions of the same type on the same FIFO buffer do not commute, and that actions implying the same message on the same bag buffer do not commute either. The communication c is remembered in the control state, allowing transitions from (7) to recognise a new step in the potential cycle in the conflict graph. Such a transition recognises a communication that is conflicting with the one that was previously remembered. This ensures that for any reachable third stage state (ι, v, c), there is a sequence of conflicting communications between the chosen send action and c. In this stage again, from each state we can read any communication and stay in the same state (transitions defined in (6)), to go through any communication not implied in the cycle in the conflict graph.

Finally, from the third stage, it is possible to read a final reception, moving to the only accepting state (transitions defined in (8)). The transitions to this accepting state must be labelled with a reception of message v in buffer ι corresponding to the send action remembered when moving to the second stage. In addition, this reception must be causally dependent on the communication remembered in the control state from which it happens. This means that its actor must be one of the participants involved in the last remembered communication.

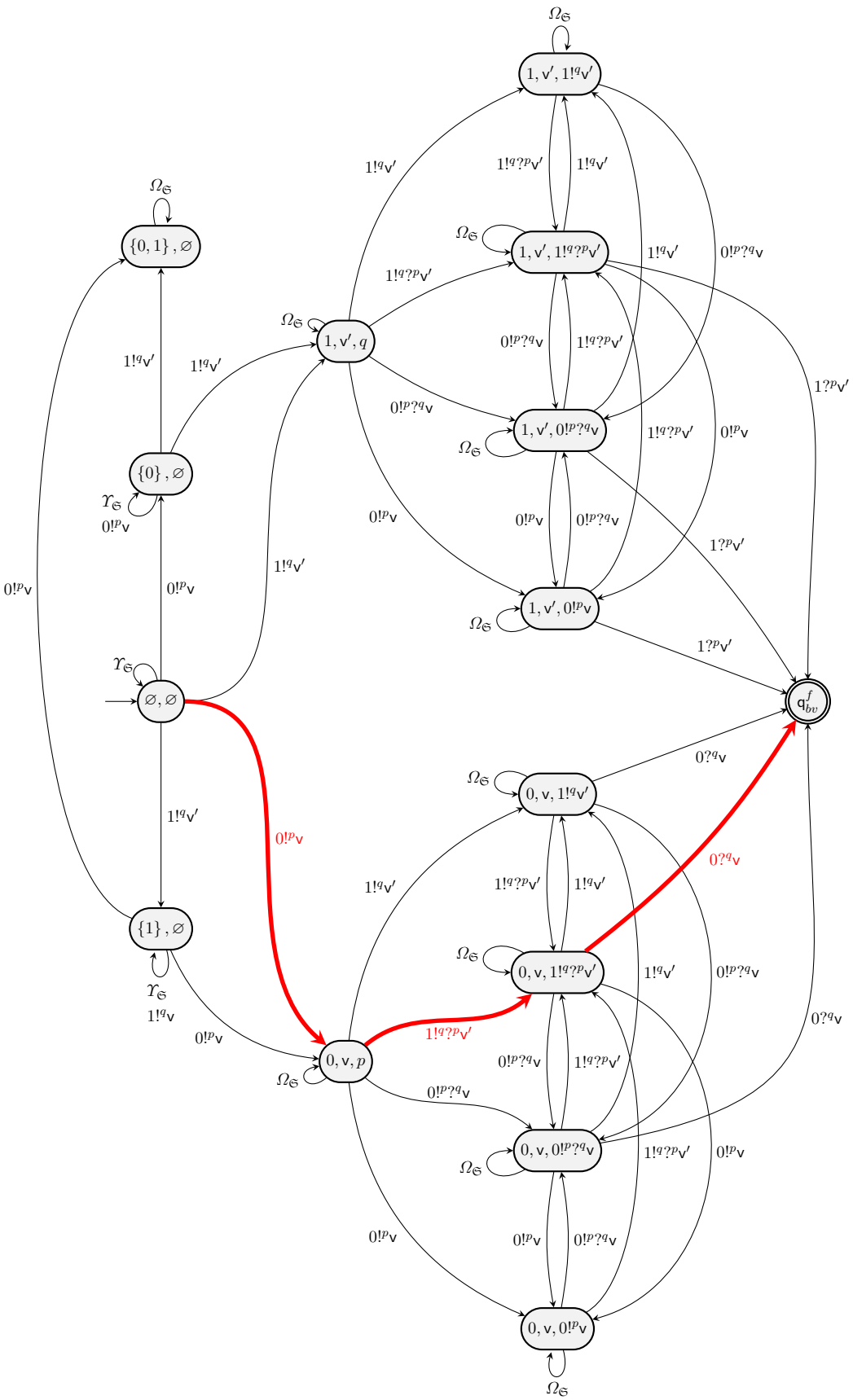


Figure 4.2: Illustration of $\mathcal{A}_{bv}(\mathfrak{S}_n)$, for $\mathfrak{S}_n$ from Example 3.2.2.

Example 4.1.4 – Let $\mathfrak{S}_n$ be the non-RSC system presented in Example 3.2.2. Figure 4.2 represents $\mathcal{A}_{bv}(\mathfrak{S}_n)$. To improve clarity, we gathered the loops reading all possible communications in a single loop labelled by the alphabet $\Omega_{\mathfrak{S}}$. The states are vertically partitioned according to the three stages, from left to right.

Notice that we did not represent all the states described in Definition 4.1.3. We chose to omit the states that are unreachable from the initial state. For instance, the second stage should include, in addition to the two states that are present, states like $(1, \mathsf{v}, p)$, or $(1, \mathsf{v}, q)$, because it is defined as all the combinations of buffer identifiers, messages, and participants. However, as there are no actions $1!^p\mathsf{v}$ or $1!^q\mathsf{v}$ in $\mathsf{Act}_{\mathfrak{S}}$, there are no transitions going from the first stage to them.

The highlighted path in the automaton represents the transitions used when recognising the sequence of communications corresponding to execution $e_n = 0!^p\mathsf{v} \cdot 1!^q\mathsf{v}' \cdot 1?^p\mathsf{v}' \cdot 0?^q\mathsf{v}$ from Example 4.1.1.

Notice that if, from second to third stage, we chose the transition $0!^p?^q\mathsf{v}$ instead of $1!^q?^p\mathsf{v}'$, we could recognise a word of communications corresponding to the execution $e' = 0!^p\mathsf{v} \cdot 0!^p\mathsf{v} \cdot 0?^q\mathsf{v} \cdot 0?^q\mathsf{v}$, which is similar to execution e from Example 4.1.2. It is a candidate borderline violation but not a borderline violation.

We now show that all *confirmed* borderline violations of a system are recognised by its automaton $\mathcal{A}_{bv}$.

Lemma 4.1.2. *Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$ its product. For all $e \in \mathsf{executions}(\mathfrak{S})$, if e is a borderline violation then there exists $w \in \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$ such that $\mathsf{cte}(w) = e$.*

Proof.

Let $\mathfrak{S}$ be a system such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. Let $e = a_1 \cdot \ldots \cdot a_n \cdot \iota?^q\mathsf{v} \in \mathsf{executions}(\mathfrak{S})$ be a borderline violation of $\mathfrak{S}$. We know that $e' = a_1 \cdot \ldots \cdot a_n$ is RSC, and that there exists $i \in \{1, \ldots, n-1\}$ such that $a_i = \iota!^p\mathsf{v}$ and $\{i, n+1\}$ is a matching pair. Because $\{i, n+1\}$ is a matching pair, we know that for all $j \in \{1, \ldots, i\}$, either $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ and $a_j \neq \iota!^r\mathsf{v}'$ with $r \in \mathbb{P}$ and $\mathsf{v}' \in \mathbb{V}$, or $a_j \neq \iota!^r\mathsf{v}$ with $r \in \mathbb{P}$. Let $w = \mathsf{etc}(e')$, we will show that $w \cdot \iota?^q\mathsf{v} \in \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$.

Let $w = w_1 \cdot c_s \cdot w_2$ with $c_s = \{\iota!^p\mathsf{v}\}$ the communication containing a_i. As $w_1 \in (\Omega_{\mathfrak{S}})^*$, $((\varnothing, \varnothing), w_1, (\mathfrak{f}, \mathfrak{b})) \in (\delta_{bv})^*$. If $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, then c_s is the first unmatched send on ι, so $\iota \notin \mathfrak{f}$. Otherwise, if ι is a bag buffer, c_s is the first unmatched send of v on ι, so $(\iota, \mathsf{v}) \notin \mathfrak{b}$. Therefore, $((\mathfrak{f}, \mathfrak{b}), c_s, (\iota, \mathsf{v}, c_s)) \in \delta_{bv}$.

Because e is a borderline violation, and because e' is RSC, there is a cycle in the conflict graph of e that is broken when removing its last reception. Let $\mathsf{cgraph}(e) = (\{1, \ldots, m\}, \rightarrow_e, \kappa_e)$, and let $i_1, \ldots, i'_m, i_f$ be a cycle in $\mathsf{cgraph}(e)$ with $\kappa_e(i_f) = \{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$ be the communication matched by the last reception. There exists $c_1 \cdot \ldots \cdot c_{m'} \sqsubseteq w_2$ with for all $j \in \{1, \ldots, m'\}$, $\kappa_e(i_j) = c_j$.

Because $i_f \rightarrow_e i_1$, we know that the send action $\iota!^p\mathsf{v}$ do not commute with an action of c_1:

- $\mathsf{buffer}(c_1) = \iota$, and either
  - $\iota \in \mathbb{I}^F_{\mathfrak{S}}$ or
  - $\mathsf{message}(c) = \mathsf{v}$, or
- $p \in \mathsf{process}(c_1)$,

so $((\iota, \mathsf{v}, p), c_1, (\iota, \mathsf{v}, c_1)) \in \delta_{\mathcal{A}_{bv}}$ (transition defined in (5) of Definition 4.1.3).

We have that $w_2 = w'_0 \cdot c_1 \cdot w'_1 \cdot c_2 \cdot \ldots \cdot c_m \cdot w'_m$. For all $j \in \{1, \ldots, m\}, w'_m \in (\Omega_{\mathfrak{S}})^*$ so because of Definition 4.1.3, (6), $\Big((i, \mathsf{v}, c_j), w'_j, (i, \mathsf{v}, c_j)\Big) \in (\delta_{\mathcal{A}_{bv}})^*$. For all $j \in \{2, \ldots, m\}$, we have that $c_{j-1} \rightarrow_e c_j$ so either

- $\mathsf{buffer}(c_j) = \mathsf{buffer}(c_{j-1})$ and either
  - $\mathsf{buffer}(c_j) \in \mathbb{I}^F_{\mathfrak{S}}$, or
  - $\mathsf{message}(c_j) = \mathsf{message}(c_{j-1})$; or
- $\mathsf{process}(c_{j-1}) \cap \mathsf{process}(c_j)$,

meaning that, according to Definition 4.1.3, (7), $((\iota, \mathsf{v}, c_{j-1}), c_j, (\iota, \mathsf{v}, c_j)) \in \delta_{\mathcal{A}_{bv}}$.

Finally, because $c_m \rightarrow_e \{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$, again we know that $q \in \mathsf{process}(c_m)$, because the send action $\iota!^p\mathsf{v}$ happens earlier in the execution, and if the reception $\iota?^q\mathsf{v}$ depended causally on the reception of c_m, it would mean that the last reception would not complete the unmatched communication c_s. By Definition 4.1.3, (8), this ensures that $\Big((\iota, \mathsf{v}, c_m), \iota?^q\mathsf{v}, \mathsf{q}^f_{bv}\Big)$. This shows that w is recognised by $\mathcal{A}_{bv}(\mathfrak{S})$, proving that the sequences of communications corresponding to all borderline violations of $\mathfrak{S}$ are recognised by $\mathcal{A}_{bv}(\mathfrak{S})$. □

All borderline violations of a system are recognised by $\mathcal{A}_{bv}$. However the language of this automaton contains more than just borderline violations. Execution e' from Example 4.1.4 illustrates this. Example 4.1.5 below is even more striking: the language recognised by $\mathcal{A}_{bv}$ might not be empty even for an RSC system, which, by definition, does not admit any borderline violation.

Example 4.1.5 – Let $\mathfrak{S}_{rsc}$ be the system from Example 3.3.1, depicted in Figure 3.4b, page 32 (and recalled in Figure 4.4a, page 49. Figure 4.3 represents $\mathcal{A}_{bv}(\mathfrak{S}_{rsc})$. We can see that even though $\mathfrak{S}_{rsc}$ is RSC, $\mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}_{rsc}))$ is not empty. Indeed, the execution $e = 1!^q\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2$ is recognised by $\mathcal{A}_{bv}(\mathfrak{S}_{rsc})$, as illustrated with the transitions used to read the communications used to compose it. However it is easy to see that this execution is causally equivalent to $e' = 1!^q\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2$, which is RSC. Execution e is therefore not a borderline violation, but it is still a candidate borderline violation. Indeed, the prefix of e, $1!^q\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2 \cdot 1!^q\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2$ was not RSC: the reception in the end of this execution matched the first send action.

What we know about the words recognised by $\mathcal{A}_{bv}$ is that they are all candidate borderline violations.

Lemma 4.1.3. *Let $\mathfrak{S}$ be a system with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$ its product. If word $w \in \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$, then $\mathsf{cte}(w)$ is a candidate borderline violation.*

Proof.

Let $\mathfrak{S}$ be a system with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$ its product, and let $\mathcal{A}_{bv}(\mathfrak{S}) = \Big(Q_{bv}, \delta_{bv}, (\varnothing, \varnothing), \Big\{\mathsf{q}^f_{bv}\Big\}\Big)$. Let $w \in \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$, we show that $\mathsf{cte}(w)$ is a candidate borderline violation.

We know that $w = w' \cdot \iota?^q\mathsf{v}_1$ for $\iota?^q\mathsf{v} \in \mathsf{Act}_{\mathfrak{S}}$. Let $e = \mathsf{cte}(w)$, and let $e' = \mathsf{cte}(w')$. We know that e' is obtained by concatenation of communications. Because of the division of the states

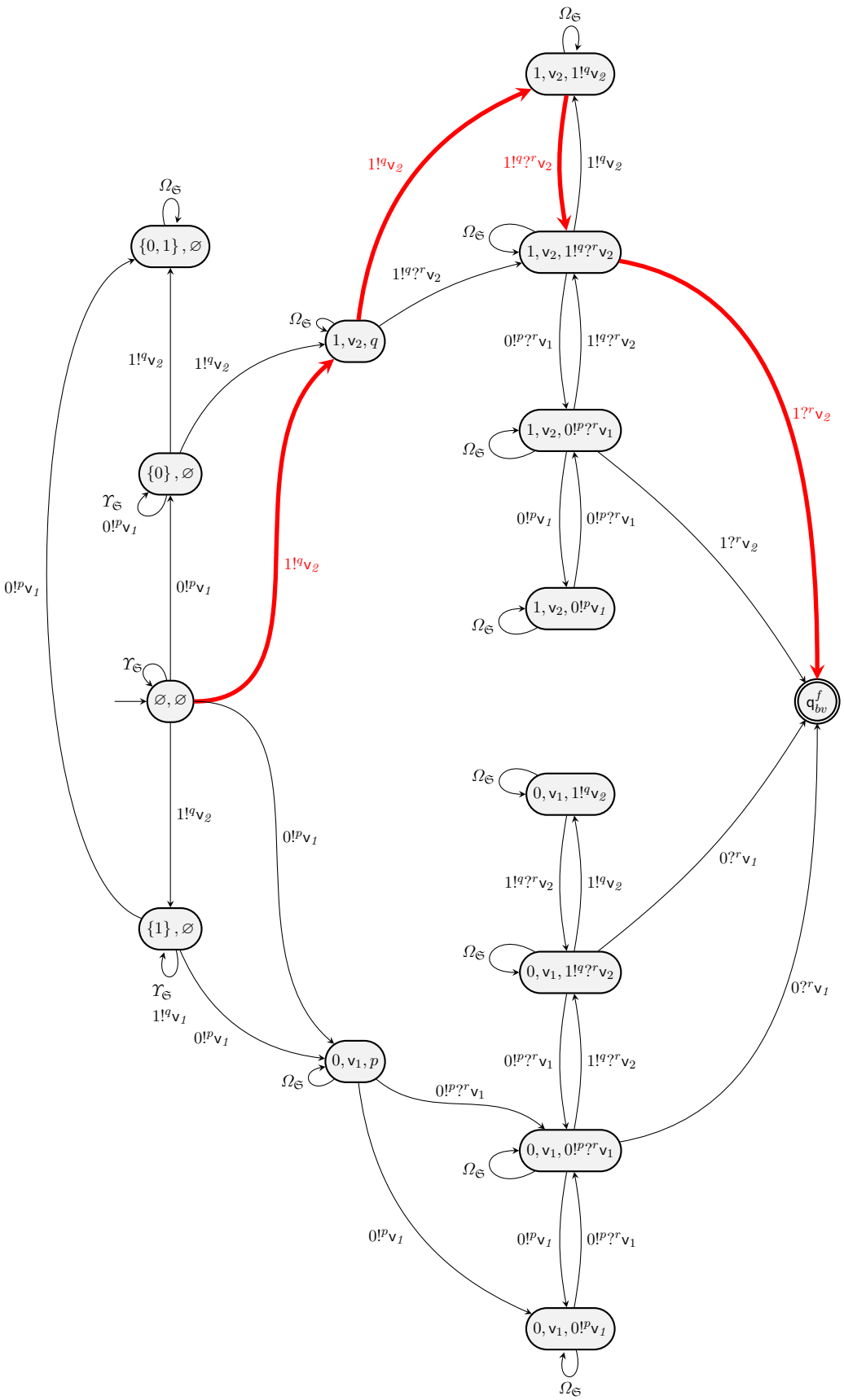


Figure 4.3: $\mathcal{A}_{bv}\left(\mathfrak{S}_{rsc}\right)$.

of $\mathcal{A}_{bv}(\mathfrak{S})$ in three stages, we know by Definition 4.1.3 that to recognise w, a transition from items (3), (5), and (8) must have been used. This implies that $\iota!^p\mathsf{v} \in e'$, and that there is at least a communication in w' between $\iota!^p\mathsf{v}$ and $\iota?^q\mathsf{v}$. These considerations ensure that e satisfies points (1) and (2) of Definition 4.1.2.

To show that the last condition of Definition 4.1.2 is satisfied as well, we will assume that e' is RSC (if it was not, this condition is trivially satisfied). As e' is composed by the concatenation of communications w', and because it is RSC, $\mathsf{com}(e') = \mathsf{letters}(w')$. Let $\mathsf{cgraph}(e) = (\{1,\dots,m\}, \rightarrow_e, \kappa_e)$, such that for all $i \in \{1,\dots,m\}, \kappa_i = c_i$. Let c_f be the communication $\{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$ containing the last reception. Let $w' = w_1 \cdot \iota!^p\mathsf{v} \cdot w_2$ with $\iota!^p\mathsf{v}$ the first unmatched communication using buffer ι if $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, or the first unmatched communication implying message v on buffer ι if $\iota \in \mathbb{I}^B_{\mathfrak{S}}$.

To recognise the last reception, with a transition (8), the automaton $\mathcal{A}_{bv}$ must have reached one of its third stage states. The reception must be causally dependent on the communication remembered in this control state. Let c_l be this communication, we know that $c_l \rightarrow_e c_f$. To reach a third stage state, with a transition (5), the communication c read by $\mathcal{A}_{bv}$ was necessarily causally dependent on the send action $\iota!^p\mathsf{v}$, so $c_f \rightarrow_e c$. Finally, to get from the first third stage state visited to the last before the final reception, each communication recognised by a transition (7) must be conflicting with the previous one, meaning there is a path $c_f \rightarrow_e c \rightarrow_e \cdots \rightarrow_e c_s \rightarrow_e c_f$ in $\mathsf{cgraph}(e)$, so $\mathsf{cgraph}(e)$ is cyclic, therefore e is a borderline violation. □

Notice that the previous lemma does not imply that all candidate borderline violations are recognised by $\mathcal{A}_{bv}$. For instance, execution $e = 0!^p\mathsf{v}_1 \cdot 1!^q?^r\mathsf{v}_2 \cdot 0!^p?^r\mathsf{v}_1 \cdot 0?^r\mathsf{v}_1$ is a candidate borderline violation $\mathfrak{S}$ from Example 4.1.5 that is not recognised by $\mathcal{A}_{bv}(\mathfrak{S}_{rsc})$ from Figure 4.3. This is not an issue because what matters is that all the confirmed borderline violations are recognised, and that nothing else than candidates borderline violations are recognised.

4.1.3 Decidability of the membership problem

Using automata $\mathcal{A}_{bv}$ and $\mathcal{A}_{rsc}$, we show that the set of borderline violations of a system is regular. The idea is to isolate, among the candidate borderline violations recognised by $\mathcal{A}_{bv}$, the ones that are confirmed borderline violations. Automaton $\mathcal{A}_{rsc}$ can recognise the RSC executions of a system, so we can use it to identify the candidate borderline violations that have an RSC prefix, therefore the ones that are confirmed borderline violations.

The idea behind the following theorem is that we can build a finite state automaton recognising borderline violations of a system by computing the intersection between $\mathcal{A}_{bv}$, and a slight modification of $\mathcal{A}_{rsc}$ such that it accepts executions ending with any reception, not necessarily following immediately its send action.

Theorem 4.1.4. *Whether a system of communicating automata is RSC is decidable.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. We denote with $\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})) \cdot R_{\mathfrak{S}}$ the set of words $\{w \cdot a \mid w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})), a \in R_{\mathfrak{S}}\}$. Let $\mathcal{L} = \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S})) \cap (\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})) \cdot R_{\mathfrak{S}})$.

We show that $e \in \mathsf{executions}(\mathfrak{S})$ is a borderline violation if and only if there exists $w \in \mathcal{L}$ with $\mathsf{cte}(w) = e$.

Let $e \in \mathsf{executions}(\mathfrak{S})$ a borderline violation, then by Lemma 4.1.2, there exists a word $w \in \mathcal{L}(\mathcal{A}_{bv}(\mathfrak{S}))$ such that $\mathsf{cte}(w) = e$. By Lemma 4.1.3 w is a candidate borderline violation, and by Definition 4.1.2, there exist $\iota \in \mathbb{I}_{\mathfrak{S}}$, $\mathsf{v} \in \mathbb{V}_{\mathfrak{S}}$, and $q \in \mathbb{P}$ such that $e = e' \cdot \iota?^q\mathsf{v}$ with e' RSC. Let $w = w' \cdot \iota?^q\mathsf{v}$, by Lemma 3.3.1, $w' \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$, so $w \in (\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})) \cdot R_{\mathfrak{S}})$, so $w \in \mathcal{L}$.

Let $w \in \mathcal{L}$, and let $e = \mathsf{cte}(w)$. Let us prove that $e \in \mathsf{executions}(\mathfrak{S})$ and that it is a borderline violation. We know that w is of the form $w' \cdot \iota?^q\mathsf{v}$, and let $e' = \mathsf{cte}(w')$. Because $w \in (\mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S})) \cdot R_{\mathfrak{S}})$, $w' \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$, so by Lemma 3.3.2 e' is RSC. By Lemma 4.1.3, w is a candidate borderline violation, and by Definition 4.1.2, because $\mathsf{cte}(w')$ is RSC, $\mathsf{cte}(w)$ is a borderline violation. □

Once a set of participants and their communication architecture is set, variations in the size of the system impact only polynomially the verification time of membership of the class of RSC systems. However, the number of buffers has an exponential impact on this time. As the number of global states is exponential with respect to the number of participants, the verification time is exponential with the number of participants as well.

4.2 Reachability

In general, the reachability problem consists in checking whether some configuration from a given set is reachable by the system. We will see in this section that this problem is decidable for RSC systems.

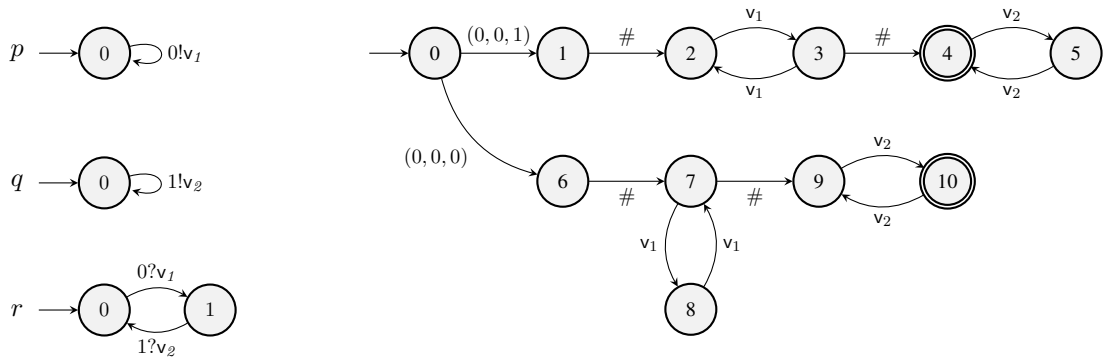
We are interested in the reachability problem to ensure safety of protocols. For a property P, we denote with $P(\mathfrak{S})$ the set of configurations of $\mathfrak{S}$ satisfying P. We call the problem consisting in checking whether a system $\mathfrak{S}$ is such that $RS(\mathfrak{S}) \cap P(\mathfrak{S}) = \varnothing$ the P safety problem. We address this problem for the properties that are *regular*. A property is regular if for all $\mathfrak{S}$, $P(\mathfrak{S})$ is a regular set.

To tackle this problem, we need a way to represent the sets of configurations as languages. As we restrict ourself to regular sets of configurations, we use regular expressions to describe them. Words matching such a regular expression encode a configuration in the set. The encoding consists in the concatenation of the different elements of a configuration: the global control state, encoded as a single letter, and the content of each FIFO buffers, separated by a specific symbol. We do not consider the content of bag buffers here, as they are not entirely relevant to the safety properties we will study. Note however that the systems may still rely on them to perform their executions. This encoding is close to the Queue-content Decision Diagram from [Boigelot and Godefroid 1996].

Definition 4.2.1 (Encoding $[\gamma]$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$, and let $\gamma = (\mathbf{l}, b_1, \ldots, b_{|\mathbb{I}_{\mathfrak{S}}|})$ be a configuration of $\mathfrak{S}$. The encoding $[\gamma]$ of γ is $\mathbf{l} \cdot \# \cdot b_{\sigma(1)} \cdot \# \cdot \ldots \cdot \# \cdot b_{\sigma\left(|\mathbb{I}^F_{\mathfrak{S}}|\right)}$, where:

- $\# \notin L_{\mathfrak{S}} \cup \mathbb{V}_{\mathfrak{S}}$, and
- $\sigma : \left|\mathbb{I}^F_{\mathfrak{S}}\right| \to |\mathbb{I}_{\mathfrak{S}}|$ is a function such that $\sigma(i)$ is the buffer index of the i-th FIFO buffer.

The $\#$ symbol is used to separate the encoding of each buffer and the encoding of the control state from that of the buffers. This symbol is chosen such that it is different from any control state,



(a) System $\mathfrak{S}_{rsc}$

(b) $\mathcal{A}_{P(\mathfrak{S}_{rsc})}$

Figure 4.4: Example of $\mathcal{A}_{P(\mathfrak{S}_{rsc})}$

any buffer identifier, and any message. Notice that we encode the content of the buffers from first message sent to last, contrary to the semantics from Definition 2.2.8.

Example 4.2.1 – Let $(\mathbf{l}, \mathbf{b})$ be a configuration of system $\mathfrak{S}_{rsc}$ from Example 3.3.1 (recalled in Figure 4.4a), such that $\mathbf{l}$ is the initial global control state, $b_0 = \mathsf{v}_1 \cdot \mathsf{v}_1$ and $b_1 = \mathsf{v}_2 \cdot \mathsf{v}_2 \cdot \mathsf{v}_2$. The encoding $[(\mathbf{l}, \mathbf{b})]$ is $\mathbf{l} \cdot \# \cdot \mathsf{v}_1 \cdot \mathsf{v}_1 \cdot \# \cdot \mathsf{v}_2 \cdot \mathsf{v}_2 \cdot \mathsf{v}_2$.

Because for a regular safety property P, and a system $\mathfrak{S}$, $P(\mathfrak{S})$ is regular, we can build an automaton $\mathcal{A}_{P(\mathfrak{S})}$ such that $\mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right) = P(\mathfrak{S})$. We use for that the encoding from Definition 4.2.1.

Example 4.2.2 – Let $\mathfrak{S}_{rsc}$ be the system recalled in Figure 4.4a. Let P be a regular safety property, such that the informal description of $P(\mathfrak{S}_{rsc})$ is: 'all the configurations where participant r is in state 1 with an odd number of messages in buffer 0 and an even number of messages in buffer 1; and all the configurations where participant r is in state 0 with an even number of messages in buffer 0 and an odd number of messages in buffer 1'. The automaton $\mathcal{A}_{P(\mathfrak{S}_{rsc})}$ recognising the encodings of the configurations satisfying this property is represented in Figure 4.4b.

4.2.1 Recognising executions leading to a given configuration

In general, the set of all reachable configurations of an RSC system is not regular. Example 4.2.3 below illustrates this, with a system whose reachability space is even context-sensitive. However, we can reason on the RSC executions of a system to check whether one of them can lead to a configuration we are looking for.

Example 4.2.3 – Let $\mathfrak{S}$ be the system containing only one communicating automaton depicted in Figure 4.5. It is trivially RSC, as there are no receptions, so no matching pair, in any of its executions. Its reachability space is not regular:

$$[RS(\mathfrak{S})] \cap \left(l_0 \# \left(\mathbb{V}_{\mathfrak{S}}\right)^* \# \left(\mathbb{V}_{\mathfrak{S}}\right)^* \# \left(\mathbb{V}_{\mathfrak{S}}\right)^*\right) = \left\{l_0 \# \mathsf{v}_1^n \# \mathsf{v}_2^n \# \mathsf{v}_3^n | n \geq 0\right\}.$$

We need to be able to check whether an RSC execution reaches a configuration with specific buffer contents. This is not trivial: the first difficulty is that the buffers can be filled in any order. Two very different executions may lead to the same configuration, as long as the order of the messages to each buffer is the same.

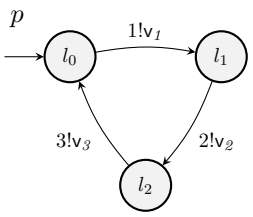
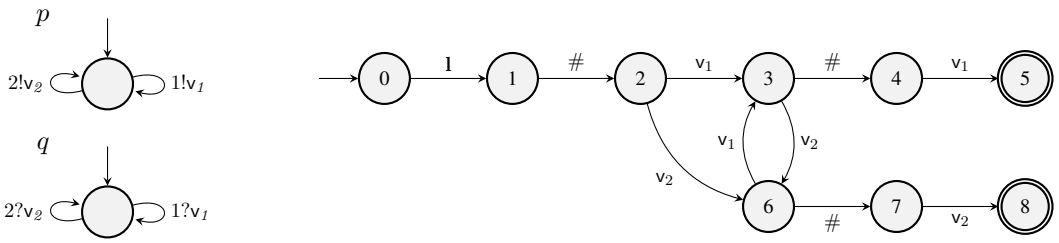


Figure 4.5: Example of system with non regular reachability space



(a) The system $\mathfrak{S}$ (b) $\mathcal{A}_{P(\mathfrak{S})}$

Figure 4.6: An example of $\mathcal{A}_{P(\mathfrak{S})}$ such that acceptance of a buffer depends on the content of an other one

Example 4.2.4 – Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system, and let execution $e_1 = \iota!^p\mathsf{v}_1 \cdot \iota!^p\mathsf{v}_1 \cdot \iota'!^q\mathsf{v}_2$ be such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$. Executions $e_2 = \iota!^p\mathsf{v}_1 \cdot \iota'!^q\mathsf{v}_2 \cdot \iota!^p\mathsf{v}_1$ and $e_3 = \iota'!^q\mathsf{v}_2 \cdot \iota!^p\mathsf{v}_1 \cdot \iota!^p\mathsf{v}_1$ both lead to γ as well, where $\gamma = (\mathbf{l}, \mathbf{b})$ with $b_\iota = \mathsf{v}_1 \cdot \mathsf{v}_1$ and $b_{\iota'} = \mathsf{v}_2$.

The second challenge is that the specification of a buffer may depend on the content of another buffer.

Example 4.2.5 – Let us consider the system $\mathfrak{S}$ depicted in Figure 4.6a, and a property P such that $P(\mathfrak{S})$ contains all the configurations where buffer 2 consists in a single message: the last message contained by buffer 1. Figure 4.6b is the representation of $\mathcal{A}_{P(\mathfrak{S})}$. Note that here it is not relevant to mention the control state, as the product of the system has only one. Execution $e = 1!\mathsf{v}_2 \cdot 2!\mathsf{v}_1 \cdot 1!\mathsf{v}_1$ leads to $\gamma \in P(\mathfrak{S})$.

However, for a finite state automaton to recognise an encoding of e, ensuring it satisfies the specification for the buffer contents, it has to accepts parts of $[\gamma]$ out of order: any sequence of actions in buffer 1 will satisfy the required content, but to accept an action adding a message in buffer 2, it has to make sure this message is the same as the last message sent to buffer 1. The difficulty is that the last action adding a message to buffer 1 could happen after a message is sent to buffer 2.

To recognise executions able to produce configurations whose encoding is recognised by an automaton $\mathcal{A}_{P(\mathfrak{S})}$, the content of each buffer must be recognised separately, as actions filling them could be interleaved, but consistency between each buffer must be ensured. We present in the next section automaton $\mathcal{A}_{ep}$ which is our answer to these challenges.

4.2.2 Automaton $\mathcal{A}_{ep}$

We explain how to build an automaton which, for a given system and an automaton $\mathcal{A}_{P(\mathfrak{S})}$, accepts all the sequences of communications leading to a configuration with a buffer content recognised by $\mathcal{A}_{P(\mathfrak{S})}$.

To circumvent the difficulties presented in the previous section, we built in the states of automaton $\mathcal{A}_{ep}$ what could be seen as 'pebbles' placed on the states of $\mathcal{A}_{P(\mathfrak{S})}$. They will allow to read the content of each buffer independently, and to accept an execution only if the parts of $\mathcal{A}_{P(\mathfrak{S})}$ used to recognise each buffer are compatible.

Definition 4.2.2 ($\mathcal{A}_{ep}(\mathfrak{S})$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$, and let P be a regular safety property. Let $\mathcal{A}_{P(\mathfrak{S})} = \left(\mathsf{Q}_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, \mathsf{q}^0_{P(\mathfrak{S})}, \mathsf{F}_{P(\mathfrak{S})}\right)$ a finite state automaton such that $\mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right) = P(\mathfrak{S})$.

$\mathcal{A}_{ep}(\mathfrak{S}) = \left(\mathsf{Q}_{ep}, \delta_{ep}, \mathsf{q}^0_{ep}, \mathsf{F}_{ep}\right)$ is the non-deterministic finite state automaton over the alphabet $\Omega_{\mathfrak{S}}$ where:

(1) $\mathsf{Q}_{ep} = \left(L_{\mathfrak{S}} \times \left(\mathsf{Q}_{P(\mathfrak{S})}\right)^{\left|\mathbb{I}^F_{\mathfrak{S}}\right|} \times \left(\mathsf{Q}_{P(\mathfrak{S})}\right)^{\left|\mathbb{I}^F_{\mathfrak{S}}\right|-1}\right) \cup \left\{\mathsf{q}^0_{ep}\right\}$;

(2) $\left(\mathsf{q}^0_{ep}, \varepsilon, (\mathbf{l}, \mathbf{p}, \mathbf{i})\right) \in \delta_{ep}$ if:

- $\left(\mathsf{q}^0_{P(\mathfrak{S})}, \mathbf{l}\cdot\#, p_1\right) \in \left(\delta_{\mathcal{A}_{P(\mathfrak{S})}}\right)^*$, and
- for all $\iota \in \left\{2, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right|\right\}, p_\iota = i_{\iota-1}$;

(3) $(\mathbf{l}, \mathbf{p}, \mathbf{i}) \in \mathsf{F}_{ep}$ if:

(a) for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}, (p_\iota, \#, i_\iota) \in \delta_{\mathcal{A}_{P(\mathfrak{S})}}$, and

(b) $p_{\left|\mathbb{I}^F_{\mathfrak{S}}\right|} \in \mathsf{F}_{\mathcal{A}_{P(\mathfrak{S})}}$;

(4) $((\mathbf{l}, \mathbf{p}, \mathbf{i}), c, (\mathbf{l}', \mathbf{p}', \mathbf{i}')) \in \delta_{ep}$ if:

- $\mathbf{l} = \mathbf{l}'$,
- $\mathbf{i} = \mathbf{i}'$,
- if $c = \iota!\mathsf{v}$ and $\iota \in \mathbb{I}^F_{\mathfrak{S}}$, then $\left(p_{\sigma(\iota)}, \mathsf{v}, p'_{\sigma(\iota)}\right) \in \delta_{\mathcal{A}}$ and for all $\iota' \in \mathbb{I}^F_{\mathfrak{S}}, \iota' \neq \iota$, $p_{\sigma(\iota')} = p'_{\sigma(\iota')}$; else, $\mathbf{p} = \mathbf{p}'$.

In a control state $(\mathbf{l}, \mathbf{p}, \mathbf{i})$, $\mathbf{p}$ represents the positions of the pebbles: for $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right|\right\}$, p_ι is the state of $\mathcal{A}_{P(\mathfrak{S})}$ on which the ι^{th} pebble is placed. To ensure consistency between the buffers recognition, we need to remember the initial position for all buffers but the first one: this is what $\mathbf{i}$ does. For $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}$, i_ι is the initial position of the $(\iota+1)^{th}$ pebble. Finally, $\mathbf{l}$ is the target control state of $\mathfrak{S}$. This is the control state of a configuration whose encoding is accepted by $\mathcal{A}_{P(\mathfrak{S})}$.

A control state $(\mathbf{l}, \mathbf{p}, \mathbf{i})$ is accepting if it is consistent, and if the last pebble is on an accepting state of $\mathcal{A}_{P(\mathfrak{S})}$. For a state to be consistent, the position of the ι^{th} pebble must be a control state of $\mathcal{A}_{P(\mathfrak{S})}$ allowing a transition to the initial position of the $(\iota+1)^{th}$ pebble, reading the separating

letter $\#$. This ensures that the execution accepted produced a buffer content that is accepted by $\mathcal{A}_{P(\mathfrak{S})}$ in its entirety, not only pieces of accepted buffer contents.

The goal of transitions from the initial state is to select the states that might be relevant to begin recognising an execution. A state $(\mathbf{l}, \mathbf{p}, \mathbf{i})$ is a relevant one if p_1 is a control state of $\mathcal{A}_{P(\mathfrak{S})}$ reachable by reading $\mathbf{l} \cdot \#$. For the other pebbles, all possible combinations are considered.

Observe that, in an execution composed only by sequences of communications, matched communications do not take part in the final buffer content. From each state of $\mathcal{A}_{ep}(\mathfrak{S})$, it is possible to read any matched communication, remaining in the same state. To read a matched communication $\{\iota!\mathsf{v}\}$ where ι is a FIFO buffer however, automaton $\mathcal{A}_{P(\mathfrak{S})}$ must be able to read v from the control state marked by the $(\sigma(\iota))^{th}$ pebble.

Example 4.2.6 – Let $\mathfrak{S}_{rsc}$ be the system recalled in Figure 4.4a and $\mathcal{A}_{P(\mathfrak{S}_{rsc})}$ be the automaton represented in Figure 4.4b. Figure 4.7 shows the graphical representation of $\mathcal{A}_{ep}(\mathfrak{S}_{rsc})$. In a state $(\mathbf{l}; \langle\mathbf{p}\rangle; \langle\mathbf{i}\rangle)$, $\mathbf{l}$ represents the targeted control state of $\mathfrak{S}_{rsc}$, $\mathbf{p}$ represents the position of each pebble, and $\mathbf{i}$ the initial position of all pebbles but the first one, here only the second one.

Similarly to what we did in Example 3.3.1, we only represent the control state of participant q as p has only one state.

The representation is partial, as illustrated by the partial dashed arrows. This is because according to Definition 4.2.2, item (2), all the combinations of pebbles positions are reachable from the initial state, as long as the first pebble is on a state reachable after reading only $\mathbf{l} \cdot \#$, where $\mathbf{l}$ is the target control state of $\mathfrak{S}$. However, only a few of these combinations allow to proceed to an accepting state. This is illustrated by the state reached through the third ε-transition from the top. The position for pebble 0 is in a different branch of $\mathcal{A}_{P(\mathfrak{S})}$ than the position of pebble 1. This implies that the condition (3)a will never be satisfied.

We show that, for a system $\mathfrak{S}$ and a safety property P, if a sequence of communications recognised by $\mathcal{A}_{ep}(\mathfrak{S})$ is RSC and feasible in $\mathfrak{S}$, then it leads to a configuration with buffers with the same content as in a configuration whose encoding is in $P(\mathfrak{S})$. In other words, $\mathcal{A}_{ep}$ is correct and recognises only executions leading to buffer contents described in the property.

Lemma 4.2.1. *Let $\mathfrak{S}$ be a system, and P a regular safety property. If $w \in \mathcal{L}(\mathcal{A}_{ep}(\mathfrak{S}))$, with $(\mathbf{l}', \mathbf{p}, \mathbf{i})$ an accepting state of w, such that $\gamma_0 \xRightarrow[\mathfrak{S}]{\mathsf{cte}(w)} (\mathbf{l}, \mathbf{b})$, then $[(\mathbf{l}', \mathbf{b})] \in \mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right)$.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. Let P be a regular safety property, and $\mathcal{A}_{P(\mathfrak{S})} = \left(\mathsf{Q}_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, \mathsf{q}^0_{P(\mathfrak{S})}, \mathsf{F}_{P(\mathfrak{S})}\right)$ the finite state automaton such that $\mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right) = P(\mathfrak{S})$. Let $\mathcal{A}_{ep}(\mathfrak{S}) = \left(\mathsf{Q}_{ep}, \delta_{ep}, \mathsf{q}^0_{ep}, \mathsf{F}_{ep}\right)$ the finite state automaton as defined in Definition 4.2.2. Let $w \in \mathcal{L}(\mathcal{A}_{ep}(\mathfrak{S}))$ be a word such that $\gamma_0 \xRightarrow[\mathfrak{S}]{\mathsf{cte}(w)} (\mathbf{l}, \mathbf{b})$. Let $(\mathbf{l}', \mathbf{p}, \mathbf{i})$ be the accepting control state of $\mathcal{A}_{ep}(\mathfrak{S})$ reached when reading w.

Because the only transitions from q^0_{ep} are the one described in item (2), we know that there exists $\mathbf{l}' \cdot \# \cdot p'_1$ prefix of an accepted word in $\mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right)$, and because of how transitions are defined in item (4) of Definition 4.2.2, we know that $\left(p'_1, \mathsf{message}\left(\mathsf{cte}\left(\mathsf{unmatched}(w)\right)\restriction_{\sigma(1)}\right), p\right) \in \left(\delta_{P(\mathfrak{S})}\right)^*$, where for $w \in \Omega$, $\mathsf{unmatched}(w)$

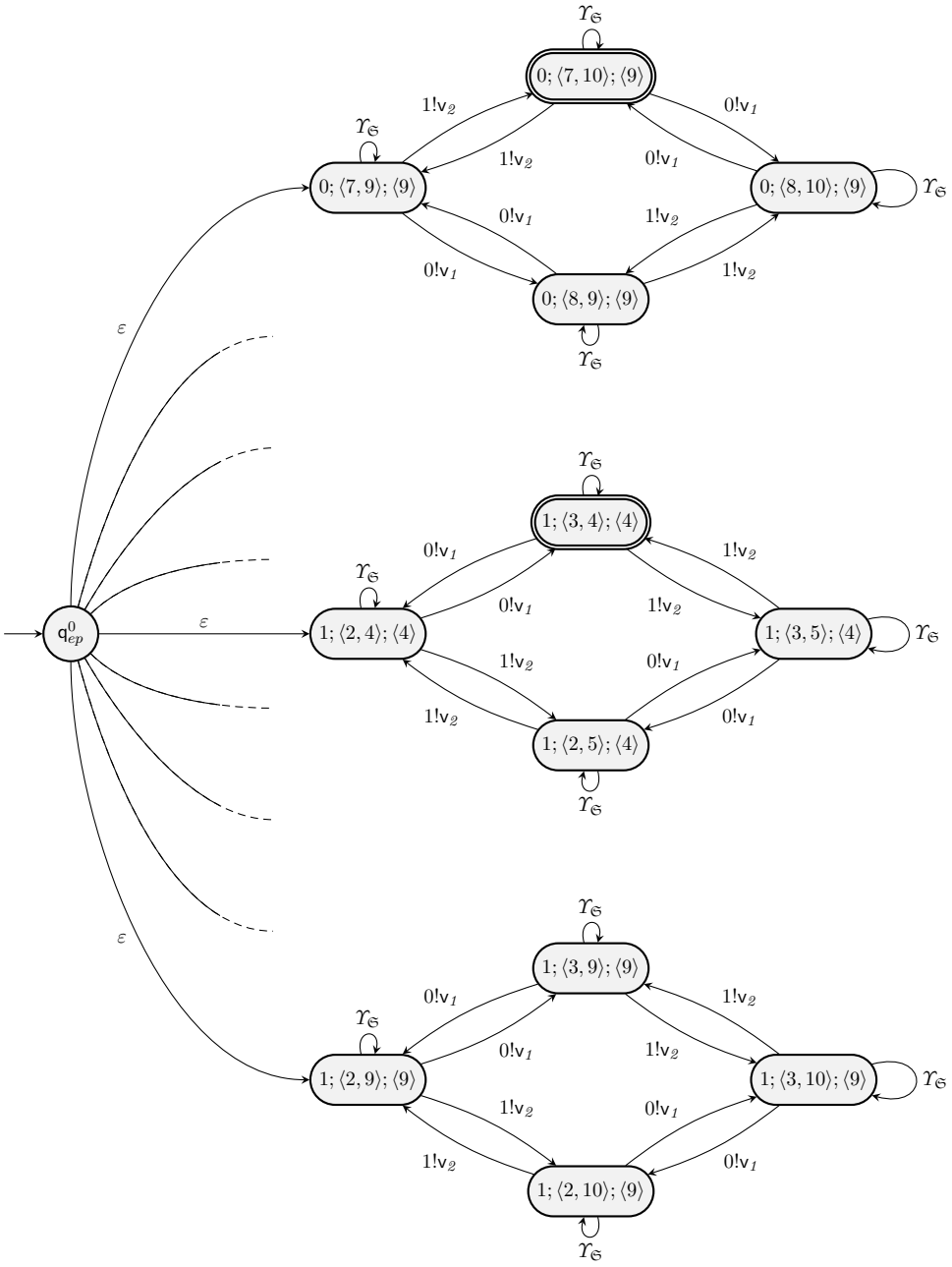


Figure 4.7: Representation of $\mathcal{A}_{ep}\,(\mathfrak{S}_{rsc})$

is the subword of w containing only unmatched communications. For each unmatched communication to be accepted, the first pebble must have been able to move, up to the position it reached at the end of the run.

Following the same logic, we know that for all buffer identifiers $\iota \in \left\{2, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right|\right\}$, $\left(i_{\iota-1}, \mathsf{message}\left(\mathsf{cte}\left(\mathsf{unmatched}\left(w\right)\right) \downarrow_{\sigma(j)}\right), p_\iota\right) \in \left(\delta_{P(\mathfrak{S})}\right)^*$.

Finally, because $(\mathbf{l}', \mathbf{p}, \mathbf{i})$ is accepting, by Definition 4.2.2 item (3), we know that for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}, (p_\iota, \#, i_\iota) \in \delta_{P(\mathfrak{S})}$, and that $p_{\left|\mathbb{I}^F_{\mathfrak{S}}\right|} \in \mathsf{F}_{P(\mathfrak{S})}$, so we have that $\mathbf{l}' \cdot \# \cdot p'_1 \cdot \mathsf{message}\left(\mathsf{cte}\left(\mathsf{unmatched}\left(w\right)\right) \downarrow_{\sigma(1)}\right) \cdot \# \cdot \ldots \cdot \# \cdot \mathsf{message}\left(\mathsf{cte}\left(\mathsf{unmatched}\left(w\right)\right) \downarrow_{\sigma\left(\left|\mathbb{I}^F_{\mathfrak{S}}\right|\right)}\right)$ is a word accepted by $\mathcal{A}_{P(\mathfrak{S})}$.

Observe that, by Definition 2.2.8, for $\iota \in \mathbb{I}^F_{\mathfrak{S}}, b_\iota = \mathsf{message}\left(\mathsf{cte}\left(\mathsf{unmatched}\left(w\right)\right)\right)$. Therefore $[(\mathbf{l}', \mathbf{b})] \in \mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right)$. □

We now show that $\mathcal{A}_{ep}$ is complete: all sequences of communications corresponding to executions $\mathfrak{S}$ leading to a configuration with buffer contents in $P\left(\mathfrak{S}\right)$ are accepted by $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$.

Lemma 4.2.2. *Let $\mathfrak{S}$ be a system of communicating automata, and let P be a regular property. Let $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$ be the finite state automaton defined in Definition 4.2.2 for $\mathfrak{S}$ and P. Let e be an RSC execution such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$. If $[\gamma] \in P\left(\mathfrak{S}\right)$, then $\mathsf{etc}\left(e\right) \in \mathcal{L}\left(\mathcal{A}_{ep}\left(\mathfrak{S}\right)\right)$.*

Proof.

Let $\mathfrak{S}$ be a system, let P be a regular safety property and $\mathcal{A}_{P(\mathfrak{S})} = \left(\mathsf{Q}_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, \mathsf{q}^0_{P(\mathfrak{S})}, \mathsf{F}_{P(\mathfrak{S})}\right)$ the finite state automaton such that $\mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right) = P\left(\mathfrak{S}\right)$.

Let e be an RSC execution such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} (\mathbf{l}, \mathbf{b})$, and assume that $[(\mathbf{l}, \mathbf{b})] \in \mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right)$.

Let $w = \mathsf{etc}\left(e\right)$: this is defined because e is RSC. We will show that w is accepted by $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$.

Because $[(\mathbf{l}, \mathbf{b})] \in \mathcal{L}\left(\mathcal{A}_{P(\mathfrak{S})}\right)$, we know that $\left(\mathsf{q}^0_{P(\mathfrak{S})}, \mathbf{l} \cdot \#, \mathsf{q}_1\right) \in \left(\delta_{P(\mathfrak{S})}\right)^*$, and that for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}$, $\left(\mathsf{q}_\iota, b_{\sigma(\iota)} \cdot \#, \mathsf{q}_{\iota+1}\right) \in \left(\delta_{P(\mathfrak{S})}\right)^*$, and that $\left(\mathsf{q}_{\left|\mathbb{I}^F_{\mathfrak{S}}\right|}, b_{\left|\mathbb{I}^F_{\mathfrak{S}}\right|}, \mathsf{q}_a\right) \in \left(\delta_{P(\mathfrak{S})}\right)^*$, with $\mathsf{q}_a \in \mathsf{F}_{P(\mathfrak{S})}$. Thus $\left(\mathsf{q}^0_{ep}, \varepsilon, (\mathbf{l}, \mathbf{p}, \mathbf{i})\right) \in \delta_{ep}$ with for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right|\right\}, p_\iota = \mathsf{q}_\iota$, and for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}, i_\iota = \mathsf{q}_{(\iota+1)}$. We can also deduce by Definition 4.2.2 item (4) that $\left((\mathbf{l}, \mathbf{p}, \mathbf{i}), w, (\mathbf{l}, \mathbf{p}', \mathbf{i})\right) \in \left(\delta_{ep}\right)^*$ such that for all $\iota \in \left\{1, \ldots, \left|\mathbb{I}^F_{\mathfrak{S}}\right| - 1\right\}, (p'_i, \#, \mathsf{q}_{\iota+1}) \in \delta_{P(\mathfrak{S})}$, and $p'_{\left|\mathbb{I}^F_{\mathfrak{S}}\right|} = \mathsf{q}_a$. By Definition 4.2.2, item (3), $(\mathbf{l}, \mathbf{p}', \mathbf{i})$ is accepting, therefore $w \in \mathcal{L}\left(\mathcal{A}_{ep}\left(\mathfrak{S}\right)\right)$. □

4.2.3 Decidability of the reachability problem for RSC systems

For a system $\mathfrak{S}$, and a property P, we can use the intersection of $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$ and $\mathcal{A}_{rsc}\left(\mathfrak{S}\right)$ to answer the P safety problem for RSC systems. However, using a simple product construction between $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$ and $\mathcal{A}_{rsc}\left(\mathfrak{S}\right)$ is not enough. Indeed, executions recognised by $\mathcal{A}_{ep}\left(\mathfrak{S}\right)$ are not required to lead in a configuration $(\mathbf{l}, \mathbf{b})$ such that $[(\mathbf{l}, \mathbf{b})]$ is accepted by $\mathcal{A}_{P(\mathfrak{S})}$, we only know that there exists $\mathbf{l}'$ such that $[(\mathbf{l}', \mathbf{b})]$ is accepted. To solve this issue, we define $\mathcal{A}_{prod}$, which is a slight variation around the usual Cartesian product between finite state automata.

Definition 4.2.3 ($\mathcal{A}_{prod}$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be an RSC system with as product $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$, $\mathcal{A}_{rsc}(\mathfrak{S}) = (\mathsf{Q}_{rsc}, \delta_{rsc}, \mathsf{q}^0_{rsc}, \mathsf{F}_{rsc})$ is the finite state automaton recognising all RSC executions of $\mathfrak{S}$ as defined in Definition 3.3.1. Let P be a safety property, and $\mathcal{A}_{P(\mathfrak{S})} = \left(\mathsf{Q}_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, \mathsf{q}^0_{P(\mathfrak{S})}, \mathsf{F}_{P(\mathfrak{S})}\right)$ the finite state automaton recognising encodings of the configurations satisfying $P(\mathfrak{S})$. Let $\mathcal{A}_{ep}(\mathfrak{S}) = \left(\mathsf{Q}_{ep}, \delta_{ep}, \mathsf{q}^0_{ep}, \mathsf{F}_{ep}\right)$ the finite state automaton as defined in Definition 4.2.2.

$\mathcal{A}_{prod}(\mathfrak{S}) = \left(\mathsf{Q}_{prod}, \delta_{prod}, \mathsf{q}^0_{prod}, \mathsf{F}_{prod}\right)$ is the non deterministic finite state automaton over alphabet $\Omega_{\mathfrak{S}}$ such that, for all $c \in \Omega_{\mathfrak{S}}$:

- $\mathsf{Q}_{prod} = \mathsf{Q}_{rsc} \times \mathsf{Q}_{ep}$,
- $\left((\mathsf{q}_{rsc}, \mathsf{q}_{ep}), c, \left(\mathsf{q}'_{rsc}, \mathsf{q}'_{ep}\right)\right) \in \delta_{prod}$ if $(\mathsf{q}_{rsc}, c, \mathsf{q}'_{rsc}) \in \delta_{rsc}$ and $\left(\mathsf{q}_{ep}, c, \mathsf{q}'_{ep}\right) \in \delta_{ep}$,
- $\mathsf{q}^0_{prod} = \left(\mathsf{q}^0_{rsc}, \mathsf{q}^0_{ep}\right)$, and
- $((\mathfrak{l}, \mathfrak{f}, \mathfrak{b}), (\mathfrak{l}', \mathbf{p}, \mathbf{i})) \in \mathsf{F}_{prod}$ if $(\mathfrak{l}', \mathbf{p}, \mathbf{i}) \in \mathsf{F}_{ep}$, and $\mathfrak{l} = \mathfrak{l}'$.

Intuitively, an execution is recognised by $\mathcal{A}_{prod}$ if:

- it is an RSC execution which produces buffer contents that are recognised by $\mathcal{A}_{P(\mathfrak{S})}$,
- is feasible in the system, and
- gets the system in the global state that was necessary for the buffer content to be recognised by $\mathcal{A}_{P(\mathfrak{S})}$.

We express that formally in the following lemma.

Lemma 4.2.3. *Let $\mathfrak{S}$ be a system, and P a regular property. Let $w \in (\Omega_{\mathfrak{S}})^*$, $w \in \mathcal{L}(\mathcal{A}_{prod})$ if and only if $\mathsf{cte}(w)$ is an RSC execution feasible in $\mathfrak{S}$ and leading to a configuration whose encoding is in $P(\mathfrak{S})$.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be an RSC system with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathfrak{l}^0_{\mathfrak{S}})$, $\mathcal{A}_{rsc}(\mathfrak{S}) = (\mathsf{Q}_{rsc}, \delta_{rsc}, \mathsf{q}^0_{rsc}, \mathsf{F}_{rsc})$ is the finite state automaton recognising all RSC executions of $\mathfrak{S}$ as defined in Definition 3.3.1. Let $\mathcal{A}_{P(\mathfrak{S})} = \left(\mathsf{Q}_{P(\mathfrak{S})}, \delta_{P(\mathfrak{S})}, \mathsf{q}^0_{P(\mathfrak{S})}, \mathsf{F}_{P(\mathfrak{S})}\right)$ the finite state automaton recognising encodings of the configurations satisfying $P(\mathfrak{S})$. Let $\mathcal{A}_{ep}(\mathfrak{S}) = \left(\mathsf{Q}_{ep}, \delta_{ep}, \mathsf{q}^0_{ep}, \mathsf{F}_{ep}\right)$ the finite state automaton as defined in Definition 4.2.2. Let $\mathcal{A}_{prod} = \left(\mathsf{Q}_{prod}, \delta_{prod}, \mathsf{q}^0_{prod}, \mathsf{F}_{prod}\right)$ be a finite state automaton as defined in Definition 4.2.3.

Let $w \in \mathcal{L}(\mathcal{A}_{prod})$, we show that $\mathsf{cte}(w)$ is an RSC execution such that $\gamma_0 \xRightarrow[\mathfrak{S}]{\mathsf{cte}(w)} (\mathfrak{l}, \mathbf{b})$, with $[(\mathfrak{l}, \mathbf{b})] \in P(\mathfrak{S})$. By Definition 4.2.3, $w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$ and $w \in \mathcal{L}(\mathcal{A}_{ep}(\mathfrak{S}))$. Let $\mathsf{q} = ((\mathfrak{l}, \mathfrak{f}, \mathfrak{b}), (\mathfrak{l}, \mathbf{p}, \mathbf{i})) \in \mathsf{Q}_{prod}$ be an accepting state of w. By Lemma 3.3.2, $\gamma_0 \xRightarrow[\mathfrak{S}]{\mathsf{cte}(w)} (\mathfrak{l}, \mathbf{b})$, and $(\mathfrak{l}, \mathbf{p}, \mathbf{i})$ is an accepting state of w in $\mathcal{A}_{ep}(\mathfrak{S})$, so by Lemma 4.2.1, $[(\mathfrak{l}, \mathbf{b})] \in P(\mathfrak{S})$.

Let $e \in \mathsf{executions}(\mathfrak{S})$ be an RSC execution such that $\gamma_0 \xRightarrow[\mathfrak{S}]{e} (\mathfrak{l}, \mathbf{b})$, and $[(\mathfrak{l}, \mathbf{b})] \in P(\mathfrak{S})$. We show that there exists w such that $\mathsf{cte}(w) = e$, and $w \in \mathcal{L}(\mathcal{A}_{prod})$. By Lemma 4.2.2,

$\mathsf{etc}(e) \in \mathcal{L}(\mathcal{A}_{ep}(\mathfrak{S}))$, with $(\mathbf{l}, \mathbf{p}, \mathbf{i})$ an accepting state of w for some $\mathbf{p}$ and $\mathbf{i}$. By Lemma 3.3.1, $w \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$, and $(\mathbf{l}, \mathfrak{f})$ is an accepting state of w in $\mathcal{A}_{rsc}(\mathfrak{S})$. By Definition 4.2.3, as both $(\mathbf{l}, \mathfrak{f}, \mathfrak{b})$ and $(\mathbf{l}, \mathbf{p}, \mathbf{i})$ are accepting states of w in $\mathcal{A}_{rsc}(\mathfrak{S})$ and $\mathcal{A}_{ep}(\mathfrak{S})$ respectively, $((\mathbf{l}, \mathfrak{f}, \mathfrak{b}), (\mathbf{l}, \mathbf{p}, \mathbf{i}))$ is an accepting state of w in $\mathcal{A}_{prod}$: $w \in \mathcal{L}(\mathcal{A}_{prod})$. □

Because the language of $\mathcal{A}_{prod}$ is exactly the feasible RSC executions of a system leading to configurations that satisfy a property, we can use it to show decidability of the P safety problem.

Theorem 4.2.4. *Let $\mathfrak{S}$ be an RSC system, and P a regular property, it is decidable whether $\mathfrak{S}$ is P safe.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be an RSC system.

Let $\mathcal{A}_{prod}(\mathfrak{S})$ be a finite state automaton as defined in Definition 4.2.3. By Lemma 4.2.3, $w \in \mathcal{L}(\mathcal{A}_{prod}(\mathfrak{S}))$ if and only if $\mathsf{cte}(w)$ is an RSC execution feasible in $\mathfrak{S}$ and leading to a configuration satisfying $P(\mathfrak{S})$. Therefore, $\mathfrak{S}$ is P safe if and only if $\mathcal{L}(\mathcal{A}_{prod}(\mathfrak{S})) = \varnothing$. □

4.2.4 Regular Safety Problems

We showcase, here, some applications of Theorem 4.2.4 by presenting few regular properties.

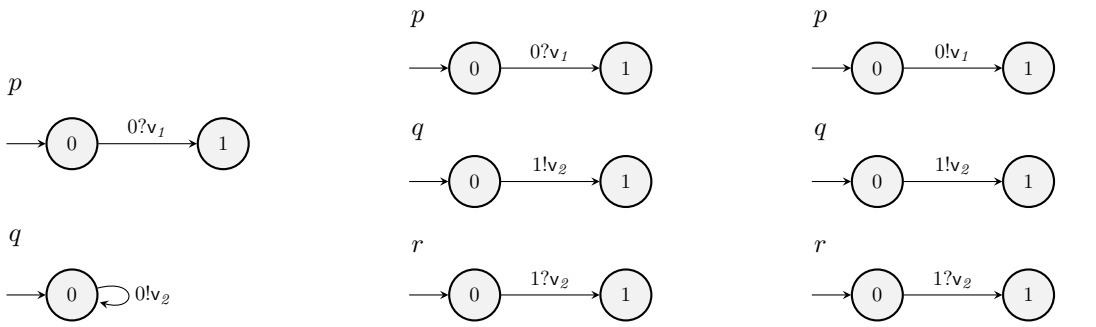
4.2.4.1 Unspecified reception

Unspecified receptions correspond to a situation where a participant awaits a message in a FIFO buffer, and receives something else, preventing it to progress any further due to the FIFO nature of its buffer. We say that a participant is in a *receiving state* if there are only (and at least one) receptions from its control state.

Definition 4.2.4 (Unspecified reception)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be a system, with its product $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. A configuration $(\mathbf{l}, \mathbf{b})$ of $\mathfrak{S}$ is an unspecified reception if there exists $p \in \mathbb{P}$ such that l_p is a receiving state, and for all $l_p \xrightarrow[\mathcal{A}_p]{\iota?\mathsf{v}} l'_p$, $\iota \in \mathbb{I}^F$, $b_\iota = \mathsf{v}' \cdot b'_\iota$ with $\mathsf{v}' \in \mathbb{V}$ and $b'_\iota \in \mathbb{V}^*$, $R_{\mathbb{P}\setminus\{p\},\{\iota\},\{\mathsf{v}'\}} \cap \mathsf{Act}_{\mathfrak{S}} = \varnothing$, and $\mathsf{v}' \neq \mathsf{v}$.

A configuration is an unspecified reception if one of the participants is in a receiving state, and none of its outgoing transitions can receive the first message in any of its buffers. A configuration is not an unspecified reception if the participant is in a receiving state and its buffers are empty: a message could arrive later, and in an unspecified reception, the participant is blocked forever: all the buffers it could receive from are blocked by a message that cannot be received. For the same reason, a configuration where a reception is possible from a bag buffer is not considered an unspecified reception, since the message could arrive later. The same reasoning applies to receptions from buffers from which other participants can receive messages as well, because the message 'blocking' the buffer could be received by another participant later.

A system $\mathfrak{S}$ is *unspecified reception free* if for all $\gamma \in RS(\mathfrak{S})$, γ is not an unspecified reception.



(a) System satisfying progress but not unspecified reception

(b) System that does not satisfy progress

(c) System with an orphan message

Figure 4.8: Systems to illustrate safety properties

Example 4.2.7 – Let $\mathfrak{S}_{csd}$ be the system from Example 2.2.2. The configurations where the $Server$ is in state 0 and buffer s starts with any message but req are unspecified receptions. Note that no such configuration is reachable in $\mathfrak{S}$, in fact this system is unspecified reception free.

The system depicted in Figure 4.8a is not however: as soon as q sent one v_2 in the FIFO buffer 0, an unspecified reception is reached, because participant p expects to receive v_1 from this buffer.

The set of unspecified receptions of a systems is regular: there is a finite number of one bounded configurations in a system, and unspecified receptions are configurations that can be obtained by adding any sequence of messages in the buffers of one bounded configurations.

4.2.4.2 Progress

Another important property is progress. Its purpose is to ensure a system does not get 'stuck' unless it has nothing more to do.

Definition 4.2.5 (Progress)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system such that its product is $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. A configuration $\gamma = (\mathbf{l}, \mathbf{b})$ of $\mathfrak{S}$ *satisfies progress* if either $\mathbf{l}$ is final, or there exists an action a and a configuration γ' such that $\gamma \xrightarrow[\mathfrak{S}]{a} \gamma'$. A system $\mathfrak{S}$ satisfies progress if for all $\gamma \in RS(\mathfrak{S})$, γ satisfies progress.

A system satisfies progress if, unless it reaches a configuration from which no transition exists (that is from which no action is supposed to happen), it can always perform an action.

Example 4.2.8 – Let us consider the system depicted in Figure 4.8b. This system does not satisfy progress: after the execution $e = 1!^q\mathsf{v}_2 \cdot 1?^r\mathsf{v}_2$, automata of both participants q and r reached a final state, but automaton of participant p did not, and not transition can be executed.

Conversely, the system from Figure 4.8a satisfies progress. Indeed, even though participant p will never be able to execute its action (because no message v_1 is ever sent), the transition of the automaton of participant q is always executable from any reachable configuration.

It can be observed that the set of configurations that do not satisfy progress is regular and polynomial time computable.

4.2.4.3 Orphan message

Intuitively, an orphan message is a message sent to a buffer, from which it will never be received. There are several ways to interpret this intuition. The following definition, from [Deniélou and Yoshida 2012a], is one of them.

Definition 4.2.6 (Orphan message configuration)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. A configuration $\gamma = (\mathbf{l}, \mathbf{b})$ is an *orphan message configuration* if $\mathbf{l}$ is final and there exists $\iota \in \mathbb{I}_{\mathfrak{S}}$ such that $b_\iota \neq \varepsilon$.

The above definition corresponds to what we could call 'terminal orphan messages', where the system has nothing more to do, and yet some messages are still there to be received. A system $\mathfrak{S}$ is *orphan message free* if no configuration $\gamma \in RS(\mathfrak{S})$ is an orphan message configuration.

Example 4.2.9 – The system in Figure 4.8c is not orphan message free. Indeed, once each participant executed its only action, a final state is reached, and buffer 0 contains message v_1 sent by p.

Both systems from Figure 4.8b and Figure 4.8a are trivially orphan message free, as no final state is reachable for them.

The set of orphan message configurations of a system is regular.

4.2.4.4 Reception-deadlock

Unspecified reception is not the only reception error that can occur. Indeed, we mentioned that a configuration in which the buffers are empty is not an unspecified reception because a message could arrive later, but if all the participant reach a reception state with all their buffers empty, the system is stuck.

Definition 4.2.7 (Reception-deadlock)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata, such that $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. A configuration $\gamma = (\mathbf{l}, \mathbf{b})$ is an *reception-deadlock configuration* if for all $p \in \mathbb{P}$, l_p is a receiving state, and for all $a_{\mathcal{A}}$ such that $l_p \xrightarrow[\mathcal{A}_p]{a_{\mathcal{A}}} l'_p$, $b_{\mathsf{buffer}(a_{\mathcal{A}})} = \varepsilon$.

A system $\mathfrak{S}$ is reception-deadlock free if for all $\gamma \in RS(\mathfrak{S})$, γ is not a reception-deadlock.

A system ends up in a reception-deadlock if all its participants reach at the same time a reception state, and all the buffers from which they may receive a message are empty. Notice that here, there is no difference between bag and FIFO buffers. The set of reception-deadlock configurations of a system is regular.

4.3 Discussion

We begin this discussion by comparing the decidability results we obtained for RSC systems to those of various classes from the literature. We recall that, in general, reachability is undecidable, due to the unbounded nature of the FIFO buffers.

The decidability results we obtained for RSC systems are similar to those for the class of k-synchronisable systems defined in [Bouajjani, Enea et al. 2018a]. Both membership of the class and reachability of a configuration are decidable, and the corrected proofs of these results can be found

in [Di Giusto, Laversa et al. 2020]. The techniques we used to prove membership are similar the one developed for k-synchronisability: the idea of characterising the borderline violations of a property for a system first appeared in [Bouajjani, Enea et al. 2018a]. It could be argued that the membership problem for the class of RSC systems can be treated as a special case of membership of the class of strong k-synchronisable systems, where $k = 1$. This problem was indeed shown decidable with both mailbox [Bollig, Giusto et al. 2021] and peer-to-peer [Laversa 2021] communication architectures. However, our proof offers two benefits compared to this approach. The first one is that, as mentioned earlier, we do not restrict ourselves to any communication architecture. Our proof holds even for architectures where participants are allowed to receive and send messages in the same buffer. The second benefit is that focusing on the instance where $k = 1$, we could devise an algorithm with a better complexity. This is important to us as we want to provide practical solutions to verification problems.

The results we obtained for the reachability problem are applicable to the class of eager systems, because all reachable configurations of such a system are reachable through an RSC execution. However, the difference in definition we mentioned in Section 3.4 has a big implication regarding membership: it is not decidable whether a system is eager [Heußner, Leroux et al. 2012].

Reachability for bounded systems, whether the bound is universal or existential, is trivially decidable. Indeed, as the alphabet of messages is also bounded, the set of configurations of a system is bounded for such a system. However, both universal and existential k-boundedness of a system is undecidable [Genest, Kuske et al. 2007]. In the same paper, this problem was shown to be decidable for deadlock-free systems, under the condition that k is known in advance. In [Genest, Kuske et al. 2007], a system is *deadlock-free* if all its states are accepting, and if from any reachable configuration, a stable configuration (with all the buffers empty, Definition 2.2.7) is reachable. Note that in general, checking whether a system of communicating automata satisfies these conditions is not decidable. As we did not consider accepting states in our setting, we can consider that all of our systems of communicating automata satisfy the first condition. The second one corresponds to the definition we give for deadlock-freedom in the Chapter 6 (Definition 6.3.2).

CHAPTER 5

Generalisation of half-duplex systems

The class of binary half-duplex systems was introduced in [Cécé and Finkel 2005]*, and offers some nice model-checking decidability results. However, this class is defined for binary systems only. We found interesting to try to generalise the definition of this class to multiparty systems: those with arbitrarily many participants. In this chapter, we propose the class of RSC systems as a natural generalisation of binary half-duplex systems.

Definition 5.0.1 (Binary half-duplex systems [Cécé and Finkel 2005])**.** A binary half duplex system is a system of communicating automata $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ with $|\mathbb{P}| = 2$, using a mailbox or peer-to-peer communication architecture, such that for all $(\mathbf{l}, \mathbf{b}) \in RS(\mathfrak{S})$, $b_0 = \varepsilon$ or $b_1 = \varepsilon$.

A binary system is binary half-duplex if in all its reachable configurations, there is no more than one buffer containing messages. One of the implication of this condition is that whenever a message is sent, either it will stay in its buffer forever and no more messages can be exchanged, or it can immediately be received.

Binary half-duplex systems benefit from decidability of a lot of verification problems. The decidability results, detailed in [Cécé and Finkel 2005], revolve around the regularity of the reachability set of those systems.

Theorem 5.0.1 ([Cécé and Finkel 2005])**.** *Regular safety problems are decidable for binary half-duplex systems.*

In addition to this decidability result, this class enjoys decidability of the membership problem. It also corresponds to a somewhat natural behaviour: the intuition behind its systems is that at a given time, a participant of a system is either 'talking' or 'listening' to the other. This makes it interesting, and generalising its properties to multiparty systems, that is with arbitrarily many participants, would yield a nice basis for verifying actual systems.

5.1 Unsuitable generalisations

In this section, we detail several attempts at generalising binary half-duplex systems. These attempts fail either at generalising enough, that is at effectively allowing more behaviours than binary half-duplex systems, or at keeping interesting decidability results. We begin with two propositions from [Cécé and Finkel 2005], and we follow with our attempt.

*The class was simply called 'half-duplex' in this paper

5.1.1 Propositions from [Cécé and Finkel 2005]

[Cécé and Finkel 2005] proposed two generalisations of the binary half-duplex systems to systems with multiple participants. They called them *natural generalisation* and *restricted generalisation*. In both these definitions, the authors considered peer-to-peer communication architectures.

Natural generalisation. The natural generalisation consists in restricting the communication between each pair of processes to half-duplex communications, as if each pair of processes formed a binary half-duplex subsystem, gathered with other subsystems to form a big half-duplex system.

Definition 5.1.1 (Natural half-duplex generalisation [Cécé and Finkel 2005])**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a peer-to-peer system of communicating automata, $\mathsf{product}\,(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$, and let r be the injection between pairs of processes and buffers (Definition 2.2.9), $\mathfrak{S}$ is *natural half-duplex* if for all $(\mathbf{l}, \mathbf{b}) \in RS(\mathfrak{S})$, for all $\{p, q\} \subseteq \mathbb{P}$, if both $\iota = r(p, q)$ and $\iota' = r(q, p)$ are defined, $b_\iota = \varepsilon$ or $b_{\iota'} = \varepsilon$.

One interesting aspect of this generalisation is that for systems with two processes, it coincides with the definition of binary half-duplex systems. However, the interest of generalising the definition to multiparty systems is to keep decidability for at least some of the verification problems that were decidable for binary systems. Unfortunately, a three participants system member of the class from Definition 5.1.1 can emulate a Turing machine, and therefore regular safety problems are not decidable in general for these systems.

Restricted generalisation. The restricted generalisation proposes the same definition as the one of binary systems: at most one buffer is not empty in any reachable configuration.

Definition 5.1.2 (Restricted half-duplex generalisation [Cécé and Finkel 2005])**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a peer-to-peer system of communicating automata with its product $\mathsf{product}\,(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$. System $\mathfrak{S}$ is *restricted half-duplex* if, for all $(\mathbf{l}, \mathbf{b}) \in RS(\mathfrak{S})$, for all $\iota \in \mathbb{I}_{\mathfrak{S}}$, if $b_\iota \neq \varepsilon$ then, for all $\iota' \in \mathbb{I}_{\mathfrak{S}} \setminus \{\iota\}$, $b_{\iota'} = \varepsilon$.

Here again, the generalised definition coincides with the binary definition for systems with two participants. The issue with this generalisation is that, as its name suggests, it is too restrictive. The point of generalising binary-half duplex systems to multiparty systems is to allow for behaviours that would not be possible in two participants systems. Here, allowing only one buffer to be used at any time, we obtain systems that are not more expressive than binary half-duplex systems.

5.1.2 Multiparty half-duplex systems

We propose another approach to generalise binary half-duplex systems, inspired by the definition of RSC systems. As we did for this notion, we begin by defining half-duplex executions, and we define multiparty half-duplex systems as the ones whose executions are all equivalent to an half-duplex one.

Definition 5.1.3 (Multiparty half-duplex system)**.** Given a system $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ with $\mathsf{product}\,(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}})$,

- a transition $(\mathbf{l}, \mathbf{b}) \xrightarrow[\mathfrak{S}]{\iota!p\mathsf{v}} (\mathbf{l}', \mathbf{b}')$ is a *half-duplex send* if for all $\iota \in \mathbb{I}^?_p$, $b_\iota = \varepsilon$;

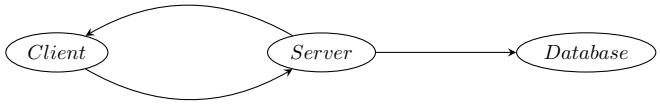


Figure 5.1: Topology of the system $\mathfrak{S}_{csd}$ in Figure 2.1

- an execution $e = \gamma_0 \xrightarrow[\mathfrak{S}]{a_1} \gamma_1 \to \cdots \xrightarrow[\mathfrak{S}]{a_n} \gamma_n$ is *half-duplex* if for all $j \in \{1, \ldots, n\}$, a_j is either a reception or a half-duplex send;
- $\mathfrak{S}$ is half-duplex if for all $e \in \mathsf{executions}(\mathfrak{S})$, there is a half-duplex execution e' such that $e \sim e'$.

Intuitively, an execution is half-duplex if all send actions are executed from a configuration where all the input buffers of the sender are empty. This means that at the time of sending a message, a communicating automaton does not have pending messages.

Example 5.1.1 – Let $\mathfrak{S}_{csd}$ be the system of communicating automata depicted in Figure 2.1, and let

$$e = s!^c req \cdot s?^s req \cdot c!^s res \cdot c?^c res \cdot s!^c ack \cdot s?^s ack \cdot s!^c req \cdot d!^s log$$

be one of its executions. Execution e is not half-duplex: when the last message log is sent, the sender ($server$) has message req in its single input buffer. However, this does not prevent $\mathfrak{S}_{csd}$ from being multiparty half-duplex:

$$e' = s!^c req \cdot s?^s req \cdot c!^s res \cdot c?^c res \cdot s!^c ack \cdot s?^s ack \cdot d!^s log \cdot s!^c req$$

is causally equivalent to e, and e' is an half-duplex execution.

This generalisation differs from the natural generalisation from [Cécé and Finkel 2005]. Indeed, their definition forbids simultaneous use of the two buffers linking each pair of participants, while ours prevents any send action to happen from a participant if *all* its input buffers are not empty. As a consequence, the Turing machine construction of Cécé and Finkel is not multiparty half-duplex following our definition. Moreover, restricted to two participants, multiparty peer-to-peer half-duplex systems are also half-duplex in the sense of the definition in [Cécé and Finkel 2005].

Unfortunately, for peer-to-peer systems, regular safety problems are still undecidable with this generalisation. To prove this, we begin by providing a graphical characterisation of multiparty half-duplex systems.

5.1.2.1 Graphical characterisation

Our definition of half-duplex system enjoys an interesting characterisation. We can define a class of communication topologies that ensures that a system (that uses such topology) is half-duplex, regardless of what the automata may do. For a peer-to-peer system $\mathfrak{S}$, let $\mathsf{topo}(\mathfrak{S})$ be the directed graph whose vertices are the participants of $\mathfrak{S}$ and whose edges $p \to q$ are the buffers that are used by the participants.

Example 5.1.2 – The topology of the system $\mathfrak{S}_{csd}$ from Figure 2.1 is depicted in Figure 5.1.

Definition 5.1.4 (Half-duplex topology)**.** A directed graph G is a *half-duplex topology* if all peer-to-peer systems $\mathfrak{S}$ such that $\mathsf{topo}(\mathfrak{S}) = G$ are half-duplex.

Lemma 5.1.1. *Let $G = (V, A)$ be a directed graph such that*

(1) G is acyclic, and

(2) for all $v \in V$, $\mathsf{d}^+(v) = 0$ or $\mathsf{d}^-(v) \leq 1$,

then G is a half-duplex topology.

Proof.

Let $G = (V, A)$ be a graph satisfying conditions (1) and (2), and $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a peer-to-peer system with $\mathsf{topo}(\mathfrak{S}) = G$, such that $\mathsf{product}\,(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{I}^0_{\mathfrak{S}})$. Thanks to the acyclicity of G, we can assign a priority to each participant of $\mathfrak{S}$ in such a way that, for all pairs of participants p, q, if $(p, q) \in A^*$, then p has lower priority on q ('priority on consumers'). Let $e \in \mathsf{executions}\,(\mathfrak{S})$ be some execution of $\mathfrak{S}$. We show that e can be rescheduled as a half-duplex execution. Without loss of generality, we assume that all send actions in e are matched. Consider the following scheduler: in order to select which machine will execute the next action, we pick the machine with highest priority whose next action in e can be immediately triggered (either because it is a send action, or because it is a receive action and the message that has to be received is already in the queue). Let e' be the execution obtained from e with this scheduler. We claim that e' is half-duplex. Indeed, assume by contradiction, that e' is not half-duplex. Then $e' = \ldots \cdot s_1 \cdot \ldots \cdot s_2 \cdot \ldots \cdot r_1 \cdot \ldots$ where s_1 is a send action from some participant p to some other participant q, s_2 is a send action from participant q, and r_1 is the reception matching s_1. Since p sends to q, q has a higher priority than p. So, when s_1 gets scheduled, q is not able to execute its next action, so this next action must be a reception. By hypothesis on G, q only receives from p, so at the time s_1 is performed, q is expecting a message v'_1 from p. Since q later performs s_2, p eventually provides v'_1 to q who receives it, that is

$$e = \ldots \cdot s_1 \cdot \ldots \cdot s'_1 \cdot \ldots \cdot r'_1 \cdot \ldots \cdot s_2 \cdot \ldots \cdot r_1 \cdot \ldots$$

with both v_1 and v'_1 transiting through the buffer between p and q, which violates the FIFO behavior of this buffer, and hence the contradiction. □

5.1.2.2 Reduction of reachability to PCP

We show that reachability in a peer-to-peer multiparty half-duplex system is undecidable by reducing to the Post correspondence problem [Post 1946] (PCP for short). We begin by providing formal definitions for instance and solutions of the PCP.

Definition 5.1.5 (PCP instance)**.** A PCP instance is a tuple $\mathcal{P} = (A, N, \alpha, \beta)$ where A is a finite alphabet, $N \geq 1$, and $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_N)$ and $\beta = (\beta_1, \beta_2, \ldots, \beta_N)$ are two sequences of N-words over A.

Definition 5.1.6 (PCP solution)**.** A solution to the PCP instance $\mathcal{P}$ is a finite, non-empty, sequence of indices $i_1, i_2, \ldots, i_m$ (with $i_j \in \{1, \ldots, N\}$) such that $\alpha_{i_1} \cdot \alpha_{i_2} \cdot \ldots \cdot \alpha_{i_m} = \beta_{i_1} \cdot \beta_{i_2} \cdot \ldots \cdot \beta_{i_m}$.

We aim at defining, for any given PCP instance $\mathcal{P}$, a special system of communicating automata $\mathfrak{S}_{\mathcal{P}}$ such that the reachability of a specific configuration in this system would depend on the existence of a solution to $\mathcal{P}$. Let $\mathcal{P} = (A, N, \alpha, \beta)$ be a PCP instance. System $\mathfrak{S}_{\mathcal{P}}$ consists in three participants. The first participant p_{g} (guesser), non-deterministically guesses a *non-empty* sequence of indices $i_1, \ldots, i_m$. For each index i, participant p_{g} sends letter by letter the word α_i to participant p_{c}. Automaton p_{g} also sends each index i to participant p_{f}. Upon reception of an index i, participant p_{f} (forwarder) sends letter by letter the word β_i to participant p_{c}. Automaton p_{c} (checker) therefore receives, from both participant p_{g} and participant p_{f}, in two distinct buffers, $\mathrm{g} \to \mathrm{c}$ and $\mathrm{f} \to \mathrm{c}$, the concatenation of words $\alpha_{i_1} \cdot \alpha_{i_2} \cdot \ldots \cdot \alpha_{i_m}$ and $\beta_{i_1} \cdot \beta_{i_2} \cdot \ldots \cdot \beta_{i_m}$, respectively. Note that since $\mathfrak{S}_{\mathcal{P}}$ is a peer-to-peer system, each buffer is associated with a pair of processes: the unique sender and receiver acting on it. We write $\mathrm{g} \to \mathrm{c}$ for the buffer between participants p_{g} and p_{c}. In order to check that the sequence of indices guessed by p_{g} is, indeed, a solution of $\mathcal{P}$, participant p_{f} repeatedly pops a pair of letters from the two buffers and checks whether, at each iteration, the two letters are the same. In order to detect the end of the sequence of indices, we introduce a special character $\#$. This system is depicted in Figure 5.2.

More formally, we denote with $|\alpha_i|$ the number of letters of α_i, and its j^{th} letter with $\alpha_{i,j}$.

The set of messages used in $\mathfrak{S}_{\mathcal{P}}$ is $\mathbb{V} = A \cup \{1, \ldots, N\} \cup \{\#\}$. The automata of $\mathfrak{S}_{\mathcal{P}}$ are defined as follows.

Definition 5.1.7 (Communicating automata encoding a PCP instance)**.** Let $\mathcal{P}$ be an instance of the Post correspondence problem.

- Automaton $\mathcal{A}_{\mathrm{g}} = (L_{\mathrm{g}}, \mathbb{V}, I_{\mathrm{g}}, \mathsf{Act}_{\mathrm{g}}, \delta_{\mathrm{g}}, l_0)$ where:
  - $L_{\mathrm{g}} = \left\{ l_{i,j},\, l'_{i,j} \,\middle|\, i \in \{1, \ldots, N\},\, j \in \{1, \ldots, |\alpha_i|\} \right\} \cup \{l_0,\, l'_0,\, l_{f_1},\, l_{f_2}\}$,
  - $\mathsf{Act}_{\mathrm{g}} = \{ {}_{\mathrm{g}\to\mathrm{f}}!^{\mathrm{g}} i,\ {}_{\mathrm{g}\to\mathrm{c}}!^{\mathrm{g}} \alpha_{i,j} \mid i \in \{1, \ldots, N\},\, j \in \{1, \ldots, |\alpha_i|\} \} \cup \left\{ {}_{\mathrm{g}\to\mathrm{c}}!^{\mathrm{g}}\#,\ {}_{\mathrm{g}\to\mathrm{f}}!^{\mathrm{g}}\# \right\}$,
  - $\delta_{\mathrm{g}} = \bigcup_{i=1}^{N} \left\{ l_{i,|\alpha_i|} \xrightarrow{\mathrm{g}\to\mathrm{f}!^{\mathrm{g}} i} l'_0,\ l'_{i,|\alpha_i|} \xrightarrow{\mathrm{g}\to\mathrm{f}!^{\mathrm{g}} i} l'_0,\ l'_0 \xrightarrow{\mathrm{g}\to\mathrm{c}!^{\mathrm{g}} \alpha_{i,1}} l_{i,1},\ l_0 \xrightarrow{\mathrm{g}\to\mathrm{c}!^{\mathrm{g}} \alpha_{i,1}} l'_{i,1} \right\} \cup$
    $\bigcup_{i=1}^{N} \bigcup_{j=2}^{|\alpha_i|} \left\{ l_{i,j-1} \xrightarrow{\mathrm{g}\to\mathrm{c}!^{\mathrm{g}} \alpha_{i,j}} l_{i,j},\ l'_{i,j-1} \xrightarrow{\mathrm{g}\to\mathrm{c}!^{\mathrm{g}} \alpha_{i,j}} l'_{i,j} \right\} \cup$
    $\left\{ l'_0 \xrightarrow{\mathrm{g}\to\mathrm{c}!^{\mathrm{g}}\#} l_{f_1},\ l_{f_1} \xrightarrow{\mathrm{g}\to\mathrm{f}!^{\mathrm{g}}\#} l_{f_2} \right\}$.
- Automaton $\mathcal{A}_{\mathrm{f}} = (L_{\mathrm{f}}, \mathbb{V}, I_{\mathrm{f}}, \mathsf{Act}_{\mathrm{f}}, \delta_{\mathrm{f}}, l_0)$ where:
  - $L_{\mathrm{f}} = \bigcup_{i=1}^{N} \{ l_{i,j} \mid j \in \{1, \ldots, |\beta_i|\} \} \cup \{l_0,\, l_{f_1},\, l_{f_2}\}$,
  - $\mathsf{Act}_{\mathrm{f}} = \bigcup_{i=1}^{N} \left\{ {}_{\mathrm{g}\to\mathrm{f}}?^{\mathrm{f}} i \right\} \cup \bigcup_{i=1}^{N} \left\{ {}_{\mathrm{f}\to\mathrm{c}}!^{\mathrm{f}} \beta_{i,j} \,\middle|\, j \in \{1, \ldots, |\alpha_i|\} \right\} \cup \left\{ {}_{\mathrm{g}\to\mathrm{f}}?^{\mathrm{f}}\#,\ {}_{\mathrm{f}\to\mathrm{c}}!^{\mathrm{f}}\# \right\}$,
  - $\delta_{\mathrm{f}} = \bigcup_{i=1}^{N} \left\{ l_0 \xrightarrow{\mathrm{g}\to\mathrm{f}?^{\mathrm{f}} i} l_{i,1},\ l_{i,|\beta_i|} \xrightarrow{\mathrm{f}\to\mathrm{c}!^{\mathrm{f}} \beta_{i,|\beta_i|}} l_0 \right\} \cup$
    $\bigcup_{i=1}^{N} \left\{ l_{i,j} \xrightarrow{\mathrm{f}\to\mathrm{c}!^{\mathrm{f}} \beta_{i,j}} l_{i,j+1} \,\middle|\, j \in \{1, \ldots, |\beta_i| - 1\} \right\} \cup$
    $\left\{ l_0 \xrightarrow{\mathrm{g}\to\mathrm{f}?^{\mathrm{f}}\#} l_{f_1},\ l_{f_1} \xrightarrow{\mathrm{f}\to\mathrm{c}!^{\mathrm{f}}\#} l_{f_2} \right\}$.

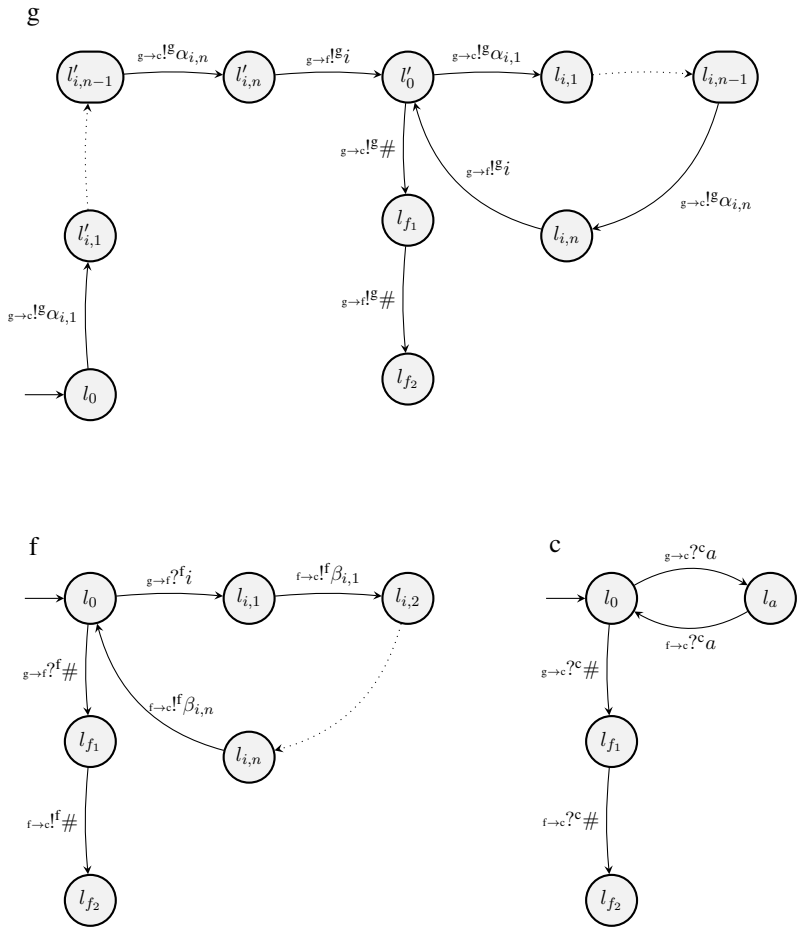


Figure 5.2: Automata p_{g}, p_{f}, and p_{c} used in the encoding of the Post correspondence problem

- Automaton $\mathcal{A}_{\mathrm{c}} = (L_{\mathrm{c}}, \mathbb{V}, I_{\mathrm{c}}, \mathsf{Act}_{\mathrm{c}}, \delta_{\mathrm{c}}, l_0)$ where:
  - $L_{\mathrm{c}} = \bigcup_{a \in A} \{l_a\} \cup \{l_0, l_{f_1}, l_{f_2}\}$,
  - $\mathsf{Act}_{\mathrm{c}} = \bigcup_{a \in A} \{{}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}a,\ {}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}a\} \cup \{{}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}\#,\ {}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}\#\}$,
  - $\delta_{\mathrm{c}} = \bigcup_{a \in A} \left\{ l_0 \xrightarrow{{}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}a} l_a,\ l_a \xrightarrow{{}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}a} l_0 \right\} \cup \left\{ l_0 \xrightarrow{{}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}\#} l_{f_1},\ l_{f_1} \xrightarrow{{}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}\#} l_{f_2} \right\}$.

Now that we have defined the system $\mathfrak{S}_{\mathcal{P}}$, we want to establish that it can be used to reduce the PCP problem to a reachability problem in a peer-to-peer half-duplex system. We first show the reduction to reachability in $\mathfrak{S}_{\mathcal{P}}$.

Lemma 5.1.2 (Reduction)**.** *Let $\mathcal{P} = (A, N, \alpha, \beta)$ be a PCP instance. Then $\mathcal{P}$ admits at least one solution if and only if the configuration $\big((l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon)\big)$ is reachable in $\mathfrak{S}_{\mathcal{P}}$.*

Proof.

Let $\mathcal{P} = (A, N, \alpha, \beta)$ be a PCP instance, and $\mathfrak{S}_{\mathcal{P}}$ the system of communicating automata encoding $\mathcal{P}$.

We begin by proving that if $\mathcal{P}$ has a solution, then $((l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$ is a reachable configuration in $\mathfrak{S}_\mathcal{P}$. Assume that $\mathcal{P}$ has a solution $i_1, i_2, \ldots, i_m$, and let $\sigma = \alpha_{i_1} \cdot \alpha_{i_2} \cdot \ldots \cdot \alpha_{i_m}$. Consider the execution

$$e = e_{i_1} \cdot e_{i_2} \cdot \ldots \cdot e_{i_m} \cdot e_v \cdot e_f$$

where, for each $j \in \{1, \cdots, N\}$, e_j denotes the execution

$$e_j = {}_{\mathrm{g}\to\mathrm{c}}!^{\mathrm{g}}\alpha_{j,1} \cdot \ldots \cdot {}_{\mathrm{g}\to\mathrm{c}}!^{\mathrm{g}}\alpha_{j,|\alpha_j|} \cdot {}_{\mathrm{g}\to\mathrm{f}}!^{\mathrm{g}}j \cdot {}_{\mathrm{g}\to\mathrm{f}}?^{\mathrm{f}}j \cdot {}_{\mathrm{f}\to\mathrm{c}}!^{\mathrm{f}}\beta_{j,1} \cdot \ldots \cdot {}_{\mathrm{f}\to\mathrm{c}}!^{\mathrm{f}}\beta_{j,|\beta_j|}$$

and

$$e_v = \bullet_{k=1}^{|\sigma|} ({}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}\sigma_k \cdot {}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}\sigma_k) \quad \text{and} \quad e_f = {}_{\mathrm{g}\to\mathrm{c}}!^{\mathrm{g}}\# \cdot {}_{\mathrm{g}\to\mathrm{f}}!^{\mathrm{g}}\# \cdot {}_{\mathrm{g}\to\mathrm{f}}?^{\mathrm{f}}\# \cdot {}_{\mathrm{f}\to\mathrm{c}}!^{\mathrm{f}}\# \cdot {}_{\mathrm{g}\to\mathrm{c}}?^{\mathrm{c}}\# \cdot {}_{\mathrm{f}\to\mathrm{c}}?^{\mathrm{c}}\#.$$

The processes executing actions in e_j are p_g and p_f. If they start with an empty buffer for g $\to$ f, they end with an buffer empty. So the sequence of actions $e_{i_1} \cdot e_{i_2} \cdot \ldots \cdot e_{i_m}$ respects the FIFO semantics, and ends with buffer g $\to$ c containing $\alpha_{i_1} \cdot \ldots \cdot \alpha_{i_m}$, whereas buffer f $\to$ c contains $\beta_{i_1} \cdot \ldots \cdot \beta_{i_m}$. Since, by hypothesis, $i_1, \ldots, i_m$ is a solution of $\mathcal{P}$, both buffers actually contain σ. So e_v respects the FIFO semantics, and we showed that $((l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$ is reachable in $\mathfrak{S}_\mathcal{P}$ through the execution e.

Conversely, if $((l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$ is reachable in $\mathfrak{S}_\mathcal{P}$, then we show that there exists a solution to $\mathcal{P}$. Let e be an execution of $\mathfrak{S}_\mathcal{P}$ such that $((l_0, l_0, l_0), (\varepsilon, \varepsilon, \varepsilon)) \stackrel{e}{\Rightarrow} (l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$. Let $e' \sim e$ be the execution obtained by rescheduling all receptions of p_c after all send actions of p_g and p_f, ie $e' = e_s \cdot e_\mathrm{c}$ where e_s contains actions of participants p_g and p_f and e_c contains actions of participant p_c. Let γ be the configuration such that $\gamma_0 \stackrel{e_s}{\Longrightarrow} \gamma$. We claim that γ is such that

(1) participants p_g and p_f are in their final state l_{f_2},

(2) participant p_c is in its initial state l_0, and buffer g $\to$ f is empty,

(3) there is a non-empty sequence of indices $i_1, \ldots, i_m$ such that g $\to$ c contains $\alpha_{i_1} \cdot \ldots \cdot \alpha_{i_m} \cdot \#$ and f $\to$ c contains $\beta_{i_1} \cdot \ldots \cdot \alpha_{i_m} \cdot \#$.

Claim 1 follows from the fact that $\gamma \stackrel{e_\mathrm{c}}{\Longrightarrow} (l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$ and participants p_g and p_f are idle during e_c. Claim 2 follows from the fact that $\gamma_0 \stackrel{e_s}{\Longrightarrow} \gamma$ and p_c remains idle during e_s. Claim 3 follows from claim 1 and claim 2 and the definition of $\mathfrak{S}_\mathcal{P}$; indeed, the only way for p_g to reach its final state is to guess a non-empty sequence $i_1, \ldots, i_m$, send it to p_f, and send $\alpha_{i_1} \cdot \ldots \cdot \alpha_{i_m} \cdot \#$ to p_c, whereas the only way for p_f to reach its final state l_{f_2} is to pop $i_1, \ldots, i_m$ and send $\beta_{i_1} \cdot \ldots \cdot \beta_{i_m} \cdot \#$ to p_c.

Now, from the fact that $\gamma \stackrel{e_\mathrm{c}}{\Longrightarrow} (l_{f_2}, l_{f_2}, l_{f_2}), (\varepsilon, \varepsilon, \varepsilon))$, we deduce that p_c receives on buffers g $\to$ c and f $\to$ c the same sequence of messages, therefore $\alpha_{i_1} \cdot \alpha_{1_2} \cdot \ldots \cdot \alpha_{i_m} = \beta_{i_1} \cdot \beta_{1_2} \cdot \ldots \cdot \beta_{i_m}$, and $i_1, i_2, \ldots, i_m$ is a solution of $\mathcal{P}$. □

Finally, observe that $\mathfrak{S}_\mathcal{P}$ is peer-to-peer half-duplex as $\mathsf{topo}(\mathfrak{S})$ is a half-duplex topology (see Figure 5.3).

So far, we have shown that $\mathfrak{S}_\mathcal{P}$ allows to reduce the existence of a solution to the PCP instance $\mathcal{P}$ to a reachability problem in $\mathfrak{S}_\mathcal{P}$ (Lemma 5.1.2). We also identified sufficient conditions on their topology for peer-to-peer systems to be half-duplex (Lemma 5.1.1). It is immediate to check that $\mathfrak{S}_\mathcal{P}$ satisfies these conditions. As a consequence, we have the following result.

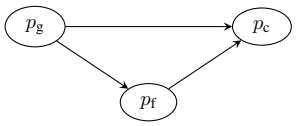


Figure 5.3: Topology of the system $\mathfrak{S}_{\mathcal{P}}$ from Figure 5.2

Theorem 5.1.3. *The configuration and control-state reachability problem are not decidable for peer-to-peer half-duplex systems.*

5.2 Mailbox multiparty half-duplex systems

We saw in the previous section that in general, regular safety problems are not decidable in multiparty half-duplex systems from Definition 5.1.3. However, our undecidability proof relied on a peer-to-peer system, and we will see in this section that the safety problems we are interested in are decidable for mailbox systems. The following theorem states that with a mailbox communication architecture, multiparty half-duplex systems are in fact RSC.

Theorem 5.2.1. *Mailbox multiparty half-duplex systems are RSC.*

Proof.

By contradiction, assume that a mailbox system $\mathfrak{S}$ is not RSC, we show that $\mathfrak{S}$ is not multiparty half-duplex. As $\mathfrak{S}$ is not RSC, there exists a borderline violation $e \in \mathsf{executions}(\mathfrak{S})$. We claim that for all e' such that $e \sim e'$, e' is not half-duplex.

Let $\mathsf{cgraph}(e') = (\{1,\dots,m\}, \to_{e'}, \kappa_{e'})$. Since e' is not causally RSC, we get, by Lemma 3.1.2, that $\mathsf{cgraph}(e')$ contains a cycle of communications $1 \to_{e'} 2 \to_{e'} \cdots \to_{e'} n \to_{e'} 1$ where for all $i \in \{1,\dots,n\}$, either $\vartheta^{-1}(i) = \{j_i, k_i\}$ is a matching pair, or $\vartheta^{-1}(i) = \{j_i\}$ is an unmatched send. We assume that $j_i < k_i$, ie j_i is the index of the send action and k_i the index of the receive action. Up to a circular permutation, we can also assume, without loss of generality, that j_1 is the first send in e' among the send actions of the communications forming the cycle, ie $j_1 < j_\ell$ for all $\ell \in \{2,\dots,n\}$. Now, let us reason by a case analysis on the nature of the edge in $\mathsf{agraph}(e')$ implying the edge $n \to_{e'} 1$ in $\mathsf{cgraph}(e')$.

- Case $j_n \prec_{e'} j_1$: then $j_n < j_1$, contradicts the minimality of j_1. Impossible.
- Case $k_n \prec_{e'} j_1$: then $j_n < k_n < j_1$, impossible.
- Case $k_n \prec_{e'} k_1$: then $k_n < k_1$ and either (1) $\mathsf{process}(a_{k_n}) = \mathsf{process}(a_{k_1})$ or (2) there is $i \in I, \mathsf{v}, \mathsf{v}' \in \mathbb{V}$ such that $a_{k_n} = \iota?\mathsf{v}$ and $a_{k_1} = \iota?\mathsf{v}'$. Because of the mailbox semantics, (1) and (2) are equivalent, so (2) is granted. But then $a_{j_n} = \iota!\mathsf{v}$ and $a_{j_1} = \iota!\mathsf{v}'$. Since e' is a FIFO execution, and $k_n < k_1$, we get that $j_n < j_1$, and again, we arrive at a contradiction.
- Case $j_n \prec_{e'} k_1$: then $j_n < k_1$, and $\mathsf{process}(a_{j_n}) = \mathsf{process}(a_{k_1})$. Moreover, $j_1 < j_n$ by the minimality of j_1.

To sum up, let $p, q, r, \mathsf{v}_1, \mathsf{v}_2$ be such that $a_{j_1} = q!^p\mathsf{v}_1$, $a_{k_1} = q?^p\mathsf{v}_1$, and $a_{j_n} = r!^q\mathsf{v}_2$. Then we have just shown that $e' = \dots \cdot q!^p\mathsf{v}_1 \cdot \dots \cdot r!^q\mathsf{v}_2 \cdot \dots \cdot q?^p\mathsf{v}_1 \cdot \dots$, so e' is not a half-duplex execution. □

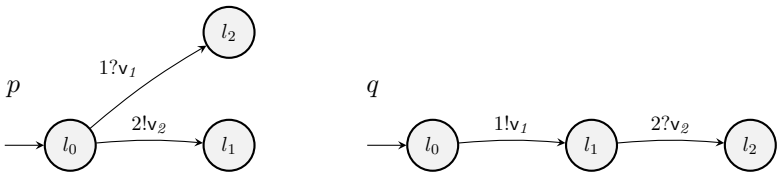


Figure 5.4: Example of RSC system that is not binary half-duplex

Notice that the converse of Theorem 5.2.1 does not hold: being RSC is not a sufficient condition to be half-duplex. Indeed, an unmatched send can fill the buffer of a process willing to send. More precisely, consider the system from Figure 5.4. It is RSC, but not binary half-duplex: the two send actions have to happen before the reception of v_2. At least one of the two messages is sent by a participant whose input buffer is not empty. However this can be seen as a pathological situation, and if a system is RSC and has no unmatched messages it is also half-duplex. In an RSC execution without unmatched messages, all messages sent are received right away, hence all send actions happen when all buffers are empty.

5.3 Why RSC is a good generalisation of binary half-duplex

As we saw, mailbox generalised half-duplex systems enjoy the same decidability results as binary half-duplex systems. Their reachability space is not regular, but as they are RSC, regular safety problems are decidable for them.

Binary half-duplex systems are RSC, and two participants RSC systems are close to binary half-duplex systems. The only difference between those two classes is the unmatched messages. Indeed, such RSC systems allow both buffers to be filled by unmatched messages, while binary half-duplex systems allow only one buffer to contain unmatched messages. As this was mentioned in the previous section, this is a pathological situation, as no system is supposed to be able to have its buffers filled with messages that cannot be matched.

As binary half-duplex systems, RSC systems enjoy decidability of regular safety problems. These decidability results hold whatever the number of participants is. As several buffers may contain messages simultaneously, they lack regularity of the reachability space, but we think this is an acceptable cost for extending the properties of binary half-duplex systems to multiparty.

To conclude this discussion, we propose in Figure 5.5 a graphical representation of the relation between the different generalisations we mentioned in this chapter.

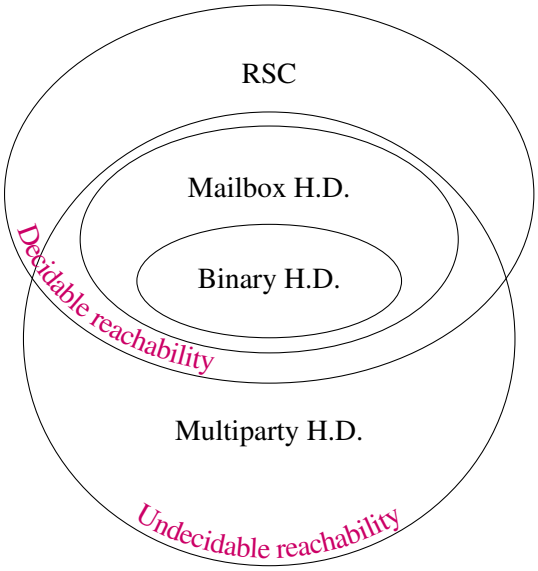


Figure 5.5: Relation between different half-duplex generalisations

CHAPTER 6

RSC characterisation of well-formed choreographies

Typically, the formal approaches reasoning on protocols of communications globally provide a way to describe synchronous behaviours: the atomic action is a message exchange between two participants. They also provide formal basis to reason on the distribution of global description on the individual participants. A global description is *well-formed* when its distribution is well-behaved.

In this chapter, we present a choreographic setting where the 'global description' is a language of synchronous execution, and the distribution of such a language is a system of communicating automata. We discuss how the notion of realisability with synchronous communication is well suited to reason on the distribution of such a language.

6.1 Choreographies

We will use the term *choreography* to refer to a global description of a protocol of communication. A choreography describes the interactions between all the participants: which message exchange can happen after which one, what can be done in parallel. We define formally choreographies as languages of synchronous executions, relying on the same generic framework we used throughout this thesis.

Definition 6.1.1 (Choreography)**.** Given a set of participants $\mathbb{P}$, a set of buffer $\mathbb{I}$ and a set of messages $\mathbb{V}$, a *choreography* (denoted $\mathcal{C}$) is a regular language, prefix closed, over $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$.

We restrict choreographies to regular languages to be able to describe them efficiently, using finite state automata. We denote with $\mathcal{A}_{\mathcal{C}}$ the finite state automaton recognising executions of $\mathcal{C}$. As $\mathcal{C}$ is prefixe closed, all control states of $\mathcal{A}_{\mathcal{C}}$ are accepting.

The intuition behind the prefix closure is that we want a global description that embeds all the possible behaviours of the system. This is similar to what we had with communicating automata, in which we did not define accepting states.

Example 6.1.1 – A choreography can be used to formalise the protocol we used to illustrate communicating automata in Example 2.2.2. Informally, this protocol is a sequence of four exchanges of messages: $request$ from the client to the server, $result$ from the server to the client, $acknowledgement$ from the client to the server, and log from the server to the database. This

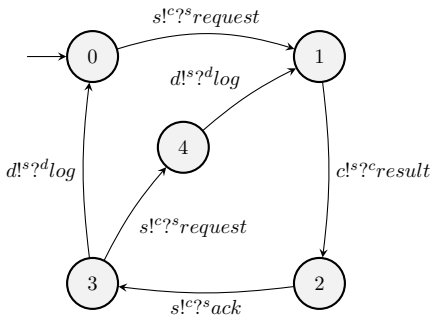


Figure 6.1: Example of a choreography: $\mathcal{C}_{csd}$

sequence of exchanges can be repeated at will. Let $\mathcal{C}_{csd}$ the choreography corresponding to this description, Figure 6.1 shows the automaton $\mathcal{A}_{\mathcal{C}}$*. Notice that to account for the fact that nothing prevents the client from sending a new $request$ before the server sent log to the database, the two orderings of these messages are composed in parallel.

We allow ourselves to extend the notation $\mathsf{cte}\,(w)$ for choreographies, defining it as such: $\mathsf{cte}\,(\mathcal{C}) = \{e | \exists w \in \mathcal{C}, e = \mathsf{cte}\,(w)\}$. To ease notation, given a choreography $\mathcal{C}$, when referring to $\mathsf{pre}\,(\mathsf{cte}\,(\mathcal{C}))$, or $\mathsf{cte}\,(\mathcal{C})^{\sim}$, we write $\mathsf{pre}\,(\mathcal{C})$, or $\mathcal{C}^{\sim}$.

A choreography is a description of a communication protocol from a global point view. We propose communicating automata to reason on their distributed description. We begin by defining the projection of a choreography onto a communicating automaton. This operation consists in extracting the behaviour of one participant from the global description.

Definition 6.1.2 (Projection)**.** Let $\mathcal{C}$ be a choreography, and $\mathcal{A}_{\mathcal{C}} = (\mathsf{Q}_{\mathcal{C}}, \delta_{\mathcal{C}}, \mathsf{q}^0_{\mathcal{C}}, \mathsf{F}_{\mathcal{C}})$ be the finite state automaton over alphabet $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$ recognising executions of $\mathcal{C}$. The *projection* of $\mathcal{C}$ on a participant $p \in \mathbb{P}$, denoted $\mathcal{C} \upharpoonright p$, is the non-deterministic communicating automaton $\mathcal{A}_p = \left(L_p, \mathbb{V}_p, \mathbb{I}_p, \mathsf{Act}_p, \delta_p, l^0_p\right)$ where:

- $L_p = \mathsf{Q}_{\mathcal{C}}$;
- $l^0_p = l^0_{\mathcal{C}}$;
- $\mathsf{Act}_p = \{\iota!\mathsf{v} \mid \exists (l, \iota!^p?^q\mathsf{v}, l') \in \delta_{\mathcal{C}}\} \cup \{\iota?\mathsf{v} \mid \exists (l, \iota!^q?^p\mathsf{v}, l') \in \delta_{\mathcal{C}}\}$;
- $\mathbb{V}_p = \{\mathsf{v} \mid \exists a_{\mathcal{A}} \in \mathsf{Act}_p, \mathsf{message}\,(a_{\mathcal{A}}) = \mathsf{v}\}$;
- $\mathbb{I}_p = \{\iota \mid \exists a_{\mathcal{A}} \in \mathsf{Act}_p, \mathsf{buffer}\,(a_{\mathcal{A}}) = \iota\}$;
- and the transition function is defined as follow:
  - $(l, \iota!\mathsf{v}, l') \in \delta_p$ if $(l, \iota!^p?^q\mathsf{v}, l') \in \delta_{\mathcal{C}}$,
  - $(l, \iota?\mathsf{v}, l') \in \delta_p$ if $(l, \iota!^q?^p\mathsf{v}, l') \in \delta_{\mathcal{C}}$, and
  - $(l, \varepsilon, l') \in \delta_p$ if $(l, c, l') \in \delta_{\mathcal{C}}$ with $p \notin \mathsf{process}\,(c)$.

*This figure corresponds to [Akroun, Salaün and Ye 2016, Fig. 4]

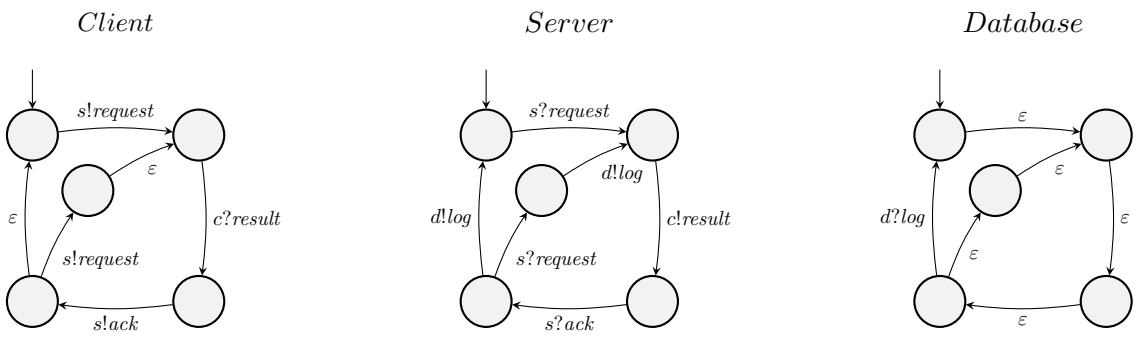


Figure 6.2: Intermediate implementation of the choreography $\mathcal{C}_{csd}$ from Figure 6.1

The projection of a choreography $\mathcal{C}$ is defined thanks to its automaton $\mathcal{A}_{\mathcal{C}}$: the obtained communicating automata have the same states and transitions as $\mathcal{A}_{\mathcal{C}}$. However, the labels of the transitions are translated from communications to communicating automaton actions. A communication is translated to a send action when projecting onto the sender, to a reception when projecting onto the receiver, and to an epsilon transition when projecting on neither of them.

While the projection allows to obtain a communicating automaton able to display all the behaviours of a participant in a choreography, what we are interested in is to obtain a system of communicating automata, able to interact with each other in a way that was described by the choreography. That is the purpose of implementation.

Definition 6.1.3 (Implementation of a choreography)**.** Let $\mathcal{C}$ be a choreography over $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$, the *intermediate implementation* of $\mathcal{C}$, denoted $\alpha_{nd}(\mathcal{C})$, is the system of non-deterministic communicating automata $(\mathcal{A}_p)_{p\in\mathbb{P}}$, where for all $p \in \mathbb{P}$, $\mathcal{A}_p = \mathcal{C} \upharpoonright p$.

The *implementation* of $\mathcal{C}$, denoted $\alpha(\mathcal{C})$, is obtained by removing the ε-transitions from $\alpha_{nd}(\mathcal{C})$[†].

The implementation of a choreography is a system of communicating automata capable of exhibiting all the behaviours of the choreography. When $\mathfrak{S} = \alpha(\mathcal{C})$, we say that $\mathfrak{S}$ *implements* $\mathcal{C}$.

Example 6.1.2 – The implementation of the choreography $\mathcal{C}_{csd}$ from Example 6.1.1 is displayed in Figure 6.3. As expected, this system behaves in a similar way to the one from Figure 2.1, page 14. The only difference is the two possible orderings of $d!log$ and $s?request$ for the server.

The intermediate implementation is shown in Figure 6.2. The use of ε-transitions simplifies the projection: providing as many copies of the automaton $\mathcal{A}_{\mathcal{C}}$ as there are participants with straightforward rewriting of the labels of the transitions is enough.

All executions described by a choreography are feasible synchronous executions in its implementation. We denote with $\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}) = \{e \mid e \in \mathsf{executions}(\mathfrak{S}), e \in \mathsf{cte}((\Upsilon_{\mathfrak{S}})^*)\}$ the set of feasible synchronous executions of $\mathfrak{S}$.

Lemma 6.1.1. *Let $\mathcal{C}$ be a choreography, $\mathcal{C} \subseteq \mathsf{executions}_{\mathsf{sync}}(\alpha(\mathcal{C}))$.*

Proof.

[†]Note that removing ε-transitions from a finite state automaton is always possible [Hopcroft and Ullman 1979].

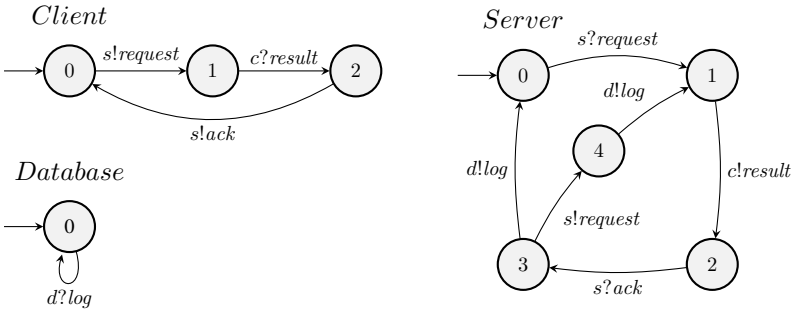


Figure 6.3: Implementation of the choreography $\mathcal{C}_{csd}$ from Figure 6.1

Let $\mathcal{C}$ be a choreography over an alphabet of matched communication $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$, and $\mathcal{A}_{\mathcal{C}} = (\mathsf{Q}_{\mathcal{C}}, \delta_{\mathcal{C}}, \mathsf{q}^0_{\mathcal{C}}, \mathsf{F}_{\mathcal{C}})$ the automaton recognising it. Let $\mathfrak{S}_{nd} = \alpha_{nd}(\mathcal{C})$ be the intermediate implementation of $\mathcal{C}$. Let $w \in \mathcal{C}$, we reason by induction on the length of w.

Let $w = w' \cdot c$, assume by induction that $w' \in \mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}_{\mathsf{nd}})$ with $\gamma_0 \xrightarrow[\mathfrak{S}_{nd}]{\mathsf{cte}(w')} (\mathbf{l}, \mathbf{b}^{\varnothing})$ such that there exists $\mathsf{q} \in \mathsf{Q}_{\mathcal{C}}$ such that for all $p \in \mathbb{P}, l_p = \mathsf{q}$. This is possible because by Definition 6.1.2, all $\mathcal{A}_p$ have the same set of states.

We show that if $w' \cdot c \in \mathcal{C}$, then there exists $\mathsf{q}' \in \mathsf{Q}_{\mathcal{C}}$ such that $(\mathbf{l}, \mathbf{b}^{\varnothing}) \xrightarrow[\mathfrak{S}_{nd}]{\mathsf{cte}(c)} (\mathbf{l}', \mathbf{b}^{\varnothing})$ with for all $p \in \mathbb{P}, l'_p = \mathsf{q}'$. Let $c = \iota!^p?^q\mathsf{v}$. We know that $(\mathsf{q}, c, \mathsf{q}') \in \delta_{\mathcal{A}_{\mathcal{C}}}$, so by Definition 6.1.2, the communicating automaton of each participant in $\mathbb{P}$ can execute a transition to state q':

- $(\mathsf{q}, \iota!\mathsf{v}, \mathsf{q}') \in \delta_p$, so $(\mathbf{l}, \mathbf{b}^{\varnothing}) \xrightarrow[\mathfrak{S}_{nd}]{\iota!^p\mathsf{v}} (\mathbf{l}^s, \mathbf{b})$, with $b_\iota = \mathsf{v}$, $l^s_p = \mathsf{q}'$ and for all $q \in \mathbb{P} \setminus \{p\}$, $l^r_q = \mathsf{q}$;
- $(\mathsf{q}, \iota?\mathsf{v}, \mathsf{q}') \in \delta_q$, so $(\mathbf{l}^s, \mathbf{b}) \xrightarrow[\mathfrak{S}_{nd}]{\iota?^q\mathsf{v}} (\mathbf{l}^r, \mathbf{b}^{\varnothing})$, with $l^r_q = l^r_p = \mathsf{q}'$, and for all $r \in \mathbb{P} \setminus \{p, q\}$, $l^r_r = \mathsf{q}$;
- for all $r \in \mathbb{P} \setminus \{p, q\}$, $(\mathsf{q}, \varepsilon, \mathsf{q}') \in \delta_r$, so $(\mathbf{l}^r, \mathbf{b}^{\varnothing}) \xrightarrow[\mathfrak{S}_{nd}]{\varepsilon} (\mathbf{l}', \mathbf{b}^{\varnothing})$.

As $w \in (\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}})^*$, $\mathsf{cte}(w)$ is a synchronous execution. It is feasible in $\mathfrak{S}_{nd}$, and as $\mathfrak{S}$ is obtained by computing the ε-closure of $\mathfrak{S}_{nd}$, $\mathsf{cte}(w) \in \mathsf{executions}_{\mathsf{sync}}(\mathfrak{S})$. □

We saw that the implementation of a choreography is capable of exhibiting all the synchronous executions of the choreography. However, the Lemma 6.1.1 does not state that all the behaviours of the projection are part of those of the choreography. In fact, it is not always the case, and this is a reason why we need a notion of well-formedness for choreographies.

6.2 Well-formedness of choreographies

Choreographies are a description of synchronous behaviours, and their implementations are asynchronous. The implementation can therefore display more behaviours than the choreography. This can be an issue sometimes, as behaviours that were not desired, and not apparent when designing the choreography, may become possible in the implementation. *Well-formedness* of a choreography is a property ensuring that its implementation will behave accordingly to what was described in the

global point of view. However we do not want to restrict the implementation to only synchronous behaviours, as we want to benefit from asynchrony in the communicating automata. Therefore, we have to define which asynchronous executions are considered as part of what a choreography describes. This is illustrated in the following example.

Example 6.2.1 – Let $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$ be an alphabet of matched communications, $\mathcal{C}_{csd} \in (\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}})^*$ be the choreography from Example 6.1.1, and $\mathfrak{S}$ its implementation. The word $w = s!^c?^s req \cdot c!^s?^c res \cdot s!^c?^s ack \cdot d!^s log \cdot s!^c req$ is obviously not in $\mathcal{C}_{csd}$, as it is not in $(\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}})^*$. However, the execution $e = \mathsf{cte}(w)$ could be completed to become synchronous execution $\mathsf{cte}(w')$ with $w' = s!^c?^s req \cdot c!^s?^c res \cdot s!^c?^s ack \cdot d!^s?^d log \cdot s!^c?^s req$, and $w' \in \mathcal{C}$. We do not want to reject $\mathfrak{S}$ as a valid implementation of $\mathcal{C}_{csd}$ because it admits this non-synchronous execution.

To answer the question from the previous paragraph, we begin by defining *partial executions*.

Definition 6.2.1 (Partial execution)**.** Let $e = a_1 \cdot \ldots \cdot a_n$ and $e' = a'_1 \cdot \ldots \cdot a'_m$ be two executions, e is a *partial execution* of e', denoted $e \trianglelefteq e'$, if there exists a function σ such that for all $i \in \{1, \ldots, n\}$:

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

Relying on action graphs, this definition can be rephrased as so: $e \trianglelefteq e'$ if $\mathsf{agraph}(e)$ is a consistent induced subgraph of $\mathsf{agraph}(e')$. Intuitively, e is a partial execution of e' if it can be obtained by removing a set of actions from e', such that no action in e depends on an action in the set of removed ones. If $a \prec a'$, with $a' \in e$, then $a \in e$ as well. For an execution e, $\mathsf{partials}(e) = \{e' | e' \trianglelefteq e\}$. We extend this notion to languages of executions: $\mathsf{partials}(\mathcal{L}) = \{e | \exists e' \in \mathcal{L}, e \trianglelefteq e'\}$.

We say that any partial execution of a synchronous execution in a given choreography is considered as part of what the choreography describes.

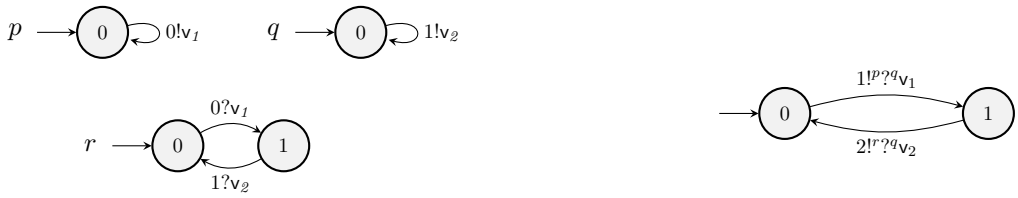
Definition 6.2.2 (Well-Formedness of a choreography)**.** A choreography $\mathcal{C}$ is *well-formed* if

- $\alpha(\mathcal{C})$ is RSC, and
- $\mathsf{partials}(\mathcal{C}) = \mathsf{executions}_{\mathsf{rsc}}(\alpha(\mathcal{C}))$.

Remember that, if the projection of a choreography is RSC, the language of its RSC executions is enough to describe all the executions of the system. Checking that this language is equivalent to the partial closure of the choreography allows us to know whether the implementation is able to exhibit unspecified behaviours or not.

Observe that the conditions of Definition 6.2.2 imply the converse of the statement of Lemma 6.1.1: for a well-formed choreography $\mathcal{C}$, $\mathcal{C} = \mathsf{executions}_{\mathsf{rsc}}(\alpha(\mathcal{C}))$.

When a choreography is not well-formed, we sometimes call them *ill-formed*. Example 6.2.2 illustrates this notion. The reason why the choreography from this example is not well-formed is that it imposes an order on actions without causal dependencies: the two send actions can happen independently and still the choreography imposes that they alternate.



(a) System $\mathfrak{S}_{rsc}$ (b) Choreography $\mathcal{C}_n$

Figure 6.4: $\mathcal{A}_{\mathcal{C}}$ of $\mathcal{C}_n$ which is not well-formed

Example 6.2.2 – Consider the choreography $\mathcal{C}_n$ whose automaton $\mathcal{A}_{\mathcal{C}}$ is represented in Figure 6.4. Its implementation is the system $\mathfrak{S}_{src}$ recalled in Figure 6.4a. This choreography is not well-formed: indeed, even though $\mathfrak{S}$ is RSC, the execution $e = 2!^{q}\mathsf{v}_2 \cdot 1!^{p}\mathsf{v}_1$ is RSC and feasible by $\mathfrak{S}$, but is not a partial execution of any execution in $\mathcal{C}_n$. This execution is causally equivalent to $e' = 1!^{p}\mathsf{v}_1 \cdot 2!^{q}\mathsf{v}_2$ which is a partial execution of $1!^{p}?^{q}\mathsf{v}_1 \cdot 2!^{q}?^{q}\mathsf{v}_2$, but no partial execution of $\mathcal{C}$ can begin with $2!^{r}\mathsf{v}_2$ and still contain $1!^{p}\mathsf{v}_1$ later, so $e \not\trianglelefteq \mathsf{cte}(w)$ for all $w \in \mathcal{C}$.

In contrast, the choreography from Example 6.1.1 is well-formed. We show now that checking well-formedness of a choreography is decidable. This result comes from the fact that for $\mathcal{C}$ a choreography, $\mathsf{partials}(\mathcal{C})$ is a regular language. To show this, we define how to build a finite state automaton $\mathcal{A}_{part}(\mathcal{C})$, and then we will show that its language is $\mathsf{partials}(\mathcal{C})$.

To build $\mathcal{A}_{part}(\mathcal{C})$, we start from $\mathcal{A}_{\mathcal{C}}$, and for each transition (labelled by a matched communication), we add two transitions, towards copies of the state of $\mathcal{A}_{\mathcal{C}}$ reached by the original transition. On one of the new transitions, we remove the reception, and on the other one, we remove the whole communication and leave only an ε. From each copied state, we copy the behaviours of $\mathcal{A}_{\mathcal{C}}$ possible from the original state, removing all the actions that are causally dependent on the actions that were removed. This process is then carried on for the new transitions as well: each matched communication that was copied undergo the same process, and each unmatched communication is doubled by an ε-transition with its own copy of the states.

Definition 6.2.3 ($\mathcal{A}_{part}$)**.** Let $\mathcal{C}$ be a choreography over alphabet $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$, and $\mathcal{A}_{\mathcal{C}} = (\mathsf{Q}_{\mathcal{C}}, \delta_{\mathcal{C}}, \mathsf{q}^0_{\mathcal{C}}, \mathsf{F}_{\mathcal{C}})$ be the finite state automaton recognising executions of $\mathcal{C}$; $\mathcal{A}_{part}(\mathcal{C}) = \left(\mathsf{Q}_{part}, \delta_{part}, \mathsf{q}^0_{part}, \mathsf{F}_{part}\right)$ is the non-deterministic finite state automaton where $\mathsf{Q}_{part} = \mathsf{Q}_{\mathcal{C}} \times 2^{\mathbb{P}} \times 3^{\mathbb{I}^F} \times 3^{\mathbb{I}^B \times \mathbb{V}}$, $\mathsf{q}^0_{part} = (\mathsf{q}^0_{\mathcal{C}}, \varnothing, \varnothing, \varnothing)$, $\mathsf{F}_{part} = \mathsf{Q}_{part}$, and

(1) $((\mathsf{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), \iota!^{p}?^{q}\mathsf{v}, (\mathsf{q}', \mathfrak{p}, \mathfrak{f}, \mathfrak{b})) \in \delta_{part}$ if:

- $(\mathsf{q}, \iota!^{p}?^{q}\mathsf{v}, \mathsf{q}') \in \delta_{\mathcal{C}}$, and
- $\{p, q\} \cap \mathfrak{p} = \varnothing$, and
- $\{\iota^r, \iota^s\} \cap \mathfrak{f} = \varnothing$ and
- $\{(\iota, \mathsf{v})^r, (\iota, \mathsf{v})^s\} \cap \mathfrak{b} = \varnothing$;

(2) $((\mathsf{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), \iota!^{p}\mathsf{v}, (\mathsf{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')) \in \delta_{part}$ if there exists $q \in \mathbb{P}$ such that:

- $(\mathsf{q}, \iota!^{p}?^{q}\mathsf{v}, \mathsf{q}') \in \delta_{\mathcal{C}}$, and
- $\mathfrak{p}' = \mathfrak{p} \cup \{q\}$, and
- $\iota^s \notin \mathfrak{f}$, and

- either:
  - $\iota \in \mathbb{I}^F$, and $\mathfrak{f}' = \mathfrak{f} \cup \{\iota^r\}$ and $\mathfrak{b}' = \mathfrak{b}$, or
  - $\iota \in \mathbb{I}^B$, and $\mathfrak{b}' = \mathfrak{b} \cup \{(\iota, \mathsf{v})^r\}$ and $\mathfrak{f}' = \mathfrak{f}$;

(3) $((\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), \varepsilon, (\mathfrak{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')) \in \delta_{part}$ if there exists $(p, q) \in \mathbb{P}^2$, $\iota \in \mathbb{I}$, and $\mathsf{v} \in \mathbb{V}$ such that:

- $(\mathfrak{q}, \iota^{!p?q}\mathsf{v}, \mathfrak{q}') \in \delta_{\mathcal{C}}$, and
- $\mathfrak{p}' = \mathfrak{p} \cup \{p, q\}$, and
- either:
  - $\iota \in \mathbb{I}^F$, and $\mathfrak{f}' = \mathfrak{f} \cup \{\iota^s\}$, or
  - $\iota \in \mathbb{I}^B$, and $\mathfrak{b}' = \mathfrak{b} \cup \{(\iota, \mathsf{v})^s\}$ and $\mathfrak{f}' = \mathfrak{f}$;

In a state $(\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b})$, $\mathfrak{q}$ represents the control state of $\mathcal{C}$, $\mathfrak{p}$ represents the set of blocked processes, and $\mathfrak{f}$ represents the set of blocked FIFO buffers, and $\mathfrak{b}$ represents the set of blocked bag buffers. The notation $\iota^s \in \mathfrak{f}$ means that ι is blocked for all actions, and ι^r that ι is blocked for receptions. The same notation applies to pairs of bag buffers and messages in $\mathfrak{b}$.

Remember that two actions have a causal dependency if either their processes are the same, or if they are of the same type on the same FIFO buffer, or if they are of the same type, implying the same message, on the same bag buffer. We say that a process, or a buffer, is 'blocked' if a transition necessary to reach $(\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b})$ is labelled by a communication in which an action by this process, or on this buffer, was removed. The intuition is that from this state, no action by such a process, or on such a buffer (implying the same message if the buffer is bag), should be part of a recognised execution.

A FIFO buffer ι, or a pair (ι, v) of bag buffer and message, can be blocked in two ways: either for receiving messages, that is a reception from this buffer was removed, or for sending, if an entire communication using this buffer was removed. Observe that it is not possible to remove a send action on a buffer without removing a reception as well, therefore, we consider a buffer that is blocked for send actions as blocked for all actions.

Remembering blocked processes and buffers ensures that, in an execution e recognised by $\mathcal{A}_{part}(\mathcal{C})$, being the partial execution of an synchronous execution w, if an action a is missing from $\mathsf{cte}(w)$, no action causally dependant on a will be part of e.

Example 6.2.3 – Figure 6.5 represents automaton $\mathcal{A}_{part}(\mathcal{C})$ for $\mathcal{C}_{csd}$ from Example 6.1.1. Parts of the automaton have been omitted, as illustrated with the dashed transitions leaving some states. These dashed transitions are labelled by ε, and lead to states from which only ε are accepted. In this example, there are no bag buffers, so we did not represent the set of blocked bag buffers on the states of the figure.

Highlighted transitions are the ones leading to a state accepting the execution e from Example 6.2.1.

Example 6.2.4 – Consider the choreography from Example 6.2.2, $\mathcal{A}_{part}(\mathcal{C}_n)$ is represented in Figure 6.6. It can be visually confirmed that no execution starting with $2!^r\mathsf{v}_2$ can contain $1!^p\mathsf{v}_1$.

We show now that the construction of $\mathcal{A}_{part}$ is correct and complete: it recognises all and only partial executions of synchronous ones in the choreography it comes from.

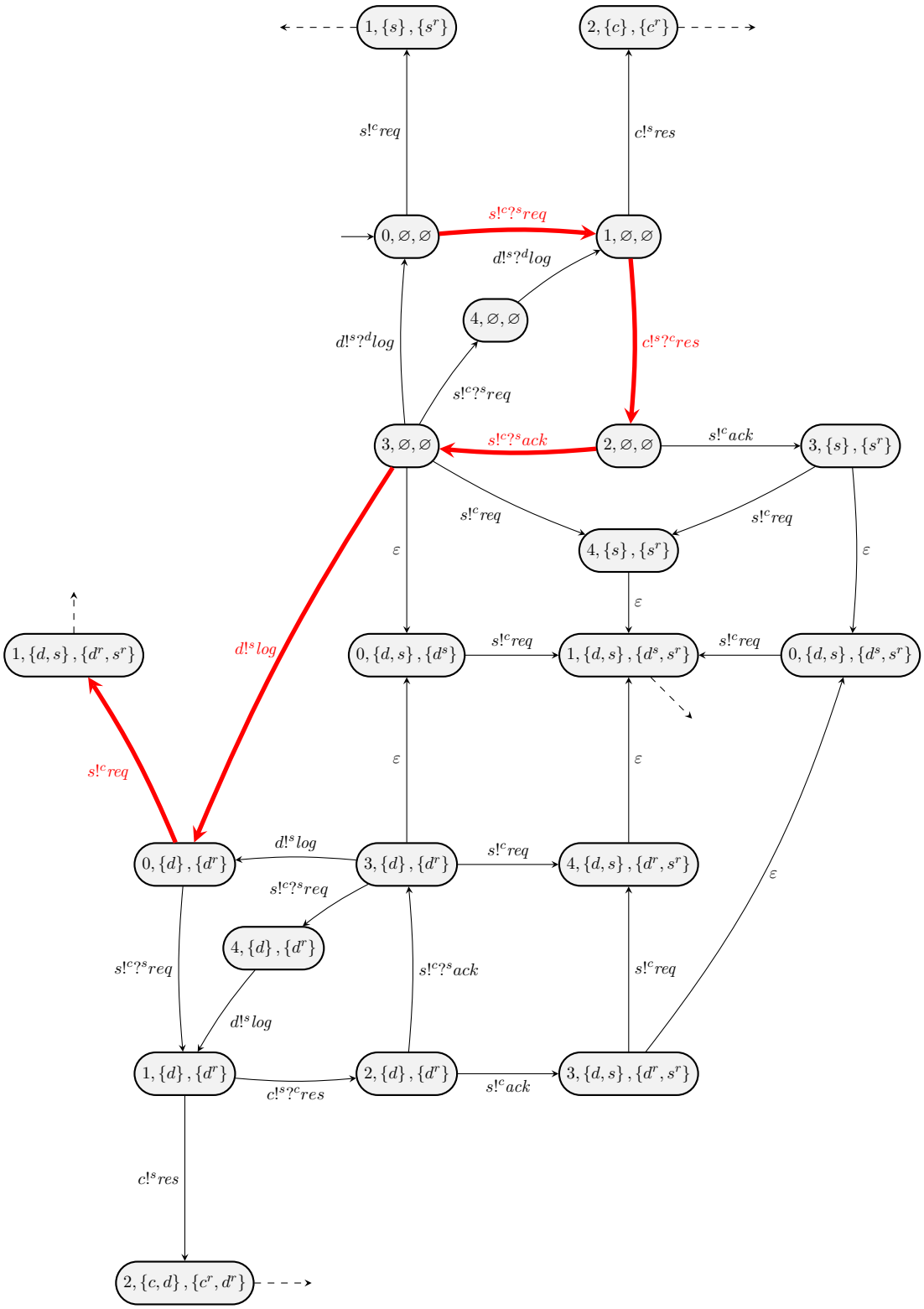


Figure 6.5: $\mathcal{A}_{part}$ $(\mathcal{C}_{csd})$

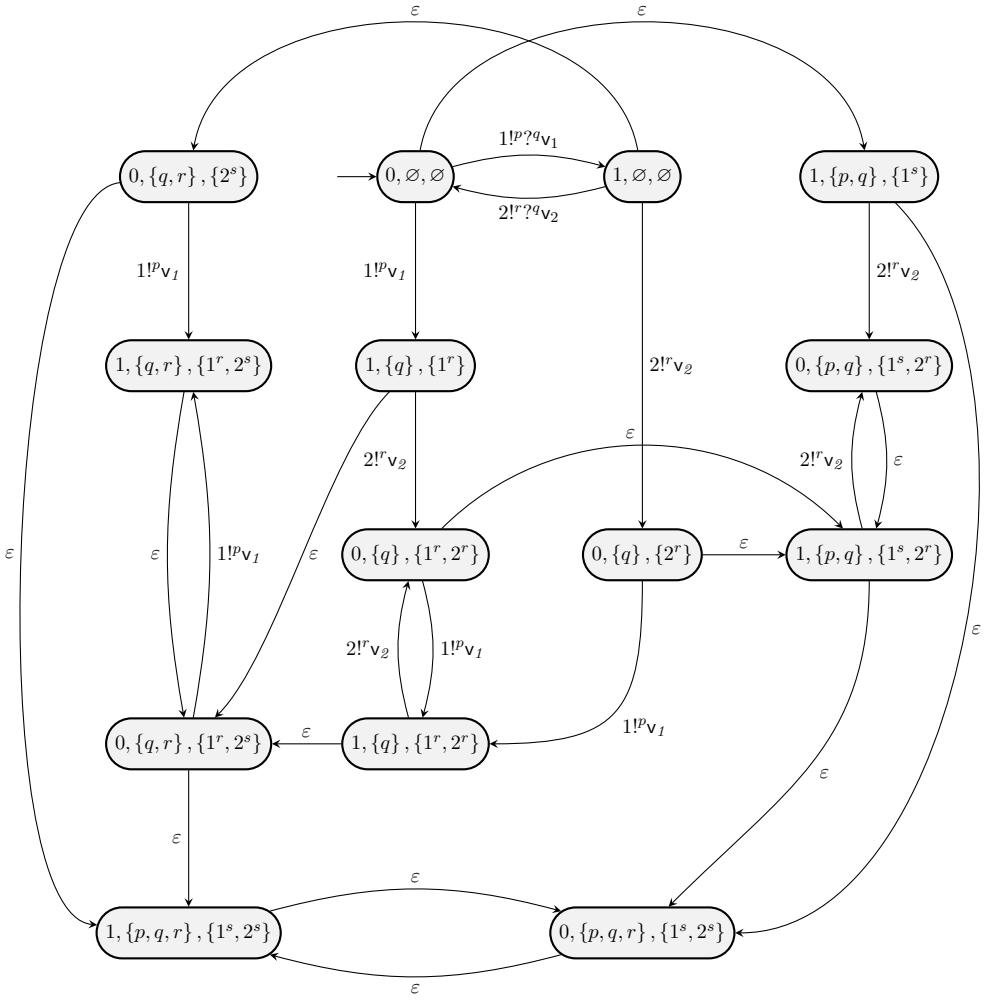


Figure 6.6: $\mathcal{A}_{part}\,(choreography)$ for $\mathcal{C}_n$ from Figure 6.4

Lemma 6.2.1. *Let $\mathcal{C}$ be a choreography, $\mathcal{L}\left(\mathcal{A}_{part}\left(\mathcal{C}\right)\right) = \mathsf{partials}\left(\mathcal{C}\right)$.*

Proof.

Let $\mathcal{C}$ be a choreography over an alphabet $\Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$, with $\mathcal{A}_{\mathcal{C}} = \left(\mathsf{Q}_{\mathcal{C}}, \delta_{\mathcal{C}}, \mathsf{q}^0_{\mathcal{C}}, \mathsf{F}_{\mathcal{C}}\right)$ the finite state automaton recognising it, and $\mathcal{A}_{part}\left(\mathcal{C}\right) = \left(\mathsf{Q}_{part}, \delta_{part}, \mathsf{q}^0_{part}, \mathsf{F}_{part}\right)$ the finite state automaton from Definition 6.2.3.

Let $w = c_1 \cdot \ldots \cdot c_n$ such that $w \in \mathcal{L}\left(\mathcal{A}_{part}\left(\mathcal{C}\right)\right)$, we will show that there exists $w' \in \mathcal{C}$ such that $\mathsf{cte}\left(w\right) \trianglelefteq \mathsf{cte}\left(w'\right)$.

Because w is recognised by $\mathcal{A}_{part}\left(\mathcal{C}\right)$, we know that for all $i \in \{1, \ldots, n\}$, there exists $\mathsf{q}_i \in \mathsf{Q}_{\mathcal{C}}, \mathfrak{p}_i \subseteq \mathbb{P}, \mathfrak{f}_i \subseteq \mathbb{I}^F \times \{r, s\}$, and $\mathfrak{b}_i \subseteq \mathbb{I}^B \times \mathbb{V} \times \{r, s\}$, such that $\left(\left(\mathsf{q}_{i-1}, \mathfrak{p}_{i-1}, \mathfrak{f}_{i-1}, \mathfrak{b}_{i-1}\right), c_i, \left(\mathsf{q}_i, \mathfrak{p}_i, \mathfrak{f}_i, \mathfrak{b}_i\right)\right) \in \delta_{part}$, with $\mathsf{q}_0 = \mathsf{q}^0_{\mathcal{C}}, \mathfrak{p}_0 = \mathfrak{f}_0 = \mathfrak{b}_0 = \varnothing$. We assume that w contains explicitly all the ε needed for $\mathcal{A}_{part}\left(\mathcal{C}\right)$ to recognise this word. By Definition 6.2.3 of $\mathcal{A}_{part}$, this means that for all $i \in \{1, \ldots, n\}$, there exists $c'_i \in \Upsilon_{\mathbb{P},\mathbb{I},\mathbb{V}}$ with $c_i \subseteq c'_i$ (or $c_i = \varepsilon$), such that $\left(\mathsf{q}_{i-1}, c'_i, \mathsf{q}_i\right) \in \delta_{\mathcal{C}}$, with $\mathsf{q}_0 = \mathsf{q}^0_{\mathcal{C}}$. Let $w' = c'_1 \cdot \ldots \cdot c'_n$, and observe that $w' \in \mathcal{C}$. Let $e' = \mathsf{cte}\left(w'\right)$, with $e' = a'_1 \cdot \ldots \cdot a'_k$, and similarly, $e = \mathsf{cte}\left(w\right)$ with $e = a_1 \cdot \ldots \cdot a_m$. We will show that $e \trianglelefteq e'$.

Let ϑ be the function such that $c_{\vartheta(i)}$ is the communication of w to which belongs a_i from e, respectively for w' and e'. Let σ be the function such that for all $i \in \{1, \ldots, m\}$, $a_i = a'_{\sigma(i)}$.

By contradiction, assume that $e \not\trianglelefteq e'$. This means that there exists $i \in \{1, \ldots, m\}$, and $j \in \{1, \ldots, \sigma(i)\}$, such that $a'_j \prec_{e'} a'_{\sigma(i)}$, and for all $i' \in \{1, \ldots, i\}, \sigma(i') \neq j$. In turns, this implies that either:

- $\mathsf{process}\left(a'_j\right) = \mathsf{process}\left(a_i\right)$, or
- $\mathsf{buffer}\left(a'_j\right) = \mathsf{buffer}\left(a_i\right)$, a'_j and a_i are of the same type, and either
  - $\iota \in \mathbb{I}^F$, or
  - $\iota \in \mathbb{I}^B$ and $\mathsf{message}\left(a'_j\right) = \mathsf{message}\left(a_i\right)$.

Let $c'_{\vartheta(j)} = \{\iota!^p\mathsf{v}, \iota?^q\mathsf{v}\}$ be the communication from which an action was removed, there are two possibilities for the removed action a'_j:

- $a'_j = \iota!^p\mathsf{v}$, then $c_{\vartheta(j)} = \varepsilon$, implying $\{p, q\} \subseteq \mathfrak{p}_{\vartheta(j)}$ and
  - if $\iota \in \mathbb{I}^F$, $\iota^s \in \mathfrak{f}_{\vartheta(j)}$,
  - $(\iota, \mathsf{v})^s \in \mathfrak{b}_{\vartheta(j)}$ otherwise; or
- $a'_j = \iota?^q\mathsf{v}$, then either
  - $c_{\vartheta(j)} = \varepsilon$, we already dealt with this case, or
  - $c_{\vartheta(j)} = \{\iota!^p\mathsf{v}\}$, implying $q \in \mathfrak{p}_{\vartheta(j)}$ and
    - if $\iota \in \mathbb{I}^F$, then $\iota^r \in \mathfrak{f}_{\vartheta(j)}$,
    - $(\iota, \mathsf{v})^r \in \mathfrak{b}_{\vartheta(j)}$ otherwise.

As $\vartheta(\sigma(i)) > \vartheta(j)$, and because in Definition 6.2.3 no transition removes element from $\mathfrak{f}$, $\mathfrak{b}$ or $\mathfrak{p}$, either $\mathsf{process}(a_i) \in \mathfrak{p}_{\vartheta(\sigma(i))-1}$ or $\mathsf{buffer}(a_i) \in \mathfrak{f}_{\vartheta(\sigma(i))-1}$. This is a contradiction, because

$$\Big(\left(\mathfrak{q}_{\vartheta(\sigma(i))-1}, \mathfrak{p}_{\vartheta(\sigma(i))-1}, \mathfrak{f}_{\vartheta(\sigma(i))-1}, \mathfrak{b}_{\vartheta(\sigma(i))-1}\right), c_{\vartheta(\sigma(i))-1}, \left(\mathfrak{q}_{\vartheta(\sigma(i))}, \mathfrak{p}_{\vartheta(\sigma(i))}, \mathfrak{f}_{\vartheta(\sigma(i))}, \mathfrak{b}_{\vartheta(\sigma(i))}\right)\Big) \in \delta_{part}.$$

Now for the other inclusion, let $e \in \mathsf{partials}(\mathcal{C})$. We show that $e \in \mathcal{L}(\mathcal{A}_{part}(\mathcal{C}))$.

By definition of $\mathsf{partials}(\mathcal{C})$ and by Definition 6.2.1 of partial executions, there exists $e' \in \mathcal{C}$ such that $e \trianglelefteq e'$, and e' is synchronous. Observe that e is RSC: it is the partial execution of a synchronous one. Let $w = \mathsf{etc}(e)$ and $w' = \mathsf{etc}(e')$. Let $w' = c'_1 \cdot \ldots \cdot c'_n$, and assume w contains explicit ε for each communication that was totally removed from w' to obtain w, hence $w = c_1 \cdot \ldots \cdot c_n$. We will show by induction on the length of w that it is accepted by $\mathcal{A}_{part}(\mathcal{C})$.

Assume by induction that $w_{pre} = c_1 \cdot \ldots \cdot c_j$, for $j < n$,, is accepted by $\mathcal{A}_{part}(\mathcal{C})$, with $(\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b})$ the state accepting w_{pre}, such that:

- $\mathfrak{p} = \{q \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \iota!^p\mathsf{v}\} \cup \{p, q \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \varepsilon\}$,
- $\mathfrak{f} = \{\iota^r \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \iota!^p\mathsf{v}\} \cup \{\iota^s \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \varepsilon\}$, and
- $\mathfrak{b} = \{(\iota,\mathsf{v})^r \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \iota!^p\mathsf{v}\} \cup \{(\iota,\mathsf{v})^s \mid \exists i \in \{1,\ldots,j\}, c'_i = \iota!^p?^q\mathsf{v}, c_i = \varepsilon\}$.

We show that $\left((\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), c_{(j+1)}, (\mathfrak{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')\right) \in \delta_{part}$, with $\mathfrak{p}'$, $\mathfrak{f}'$ and $\mathfrak{b}'$ satisfying the same constraints as $\mathfrak{p}$, $\mathfrak{f}$, and $\mathfrak{b}$ respectively for $j = j + 1$. Depending on $c_{(j+1)}$:

- $c_{(j+1)} = \iota!^p?^q\mathsf{v}$, so $c_{(j+1)} = c'_{(j+1)}$. Because, by hypothesis, we have $\mathsf{cte}(c_1 \cdot \ldots \cdot c_j) \trianglelefteq \mathsf{cte}\left(c'_1 \cdot \ldots \cdot c'_j\right)$, by Definition 6.2.1 of partial executions we know that for all $k \in \{1,\ldots,j\}$, for all $a \in c'_k$, if:
  - $\mathsf{process}(a) \in \{p, q\}$, or
  - $\mathsf{buffer}(a) = \iota$ and either
    - $\iota \in \mathbb{I}^F$ or
    - $\iota \in \mathbb{I}^B$ and $\mathsf{buffer}(a) = \mathsf{v}$,

  then $a \in c_k$. So by induction hypothesis, $\{p, q\} \cap \mathfrak{p} = \varnothing$, $\iota^r \notin \mathfrak{f}$ and $(\iota,\mathsf{v})^r \notin \mathfrak{b}$. By item (1) of Definition 6.2.3, $\left((\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), c_{(j+1)}, (\mathfrak{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')\right) \in \delta_{part}$, with $\mathfrak{p}' = \mathfrak{p}$, $\mathfrak{f}' = \mathfrak{f}$, and $\mathfrak{b} = \mathfrak{b}'$.
- $c_{(j+1)} = \iota!^p\mathsf{v}$, so there exists $a' = \iota?^q\mathsf{v}$ such that $c'_{(j+1)} = c_{(j+1)} \cup \{a'\}$. By Definition 6.2.1 again, we know that for all $k \in \{1,\ldots,j\}$, for all $a \in c'_k$, if $\mathsf{process}(a) = p$ then $a \in c_k$. Moreover, if $a \in S_{\mathbb{P},\mathbb{I},\mathbb{V}}$, and $\mathsf{buffer}(a) = \iota$, if either $\iota \in \mathbb{I}^F$ or $\mathsf{message}(a) = \mathsf{v}$, $a \in c_k$. So, by induction hypothesis, $p \notin \mathfrak{p}$, $\iota^s \notin \mathfrak{f}$, and $(\iota,\mathsf{v})^s \notin \mathfrak{b}$. By item (2) of Definition 6.2.3, $\left((\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), c_{(j+1)}, (\mathfrak{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')\right) \in \delta_{part}$, with $\mathfrak{f}' = \mathfrak{f} \cup \{\iota\}$ and $\mathfrak{p}' = \mathfrak{p} \cup \{q\}$.

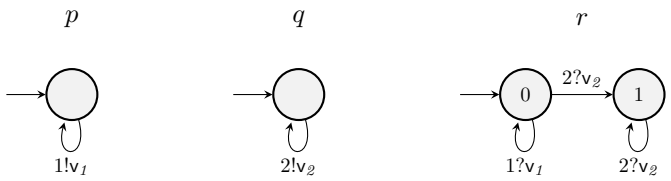


Figure 6.7: System that is not deadlock-free

- $c_{(j+1)} = \varnothing$, let $c'_{(j+1)} = \iota!^{p}?^{q}\mathsf{v}$. By item (3) of Definition 6.2.3, we have that $((\mathfrak{q}, \mathfrak{p}, \mathfrak{f}, \mathfrak{b}), \varepsilon, (\mathfrak{q}', \mathfrak{p}', \mathfrak{f}', \mathfrak{b}')) \in \delta_{part}$, with $\mathfrak{p}' = \mathfrak{p} \cup \{p, q\}$, and either
  - $\mathfrak{f}' = \mathfrak{f} \cup \{\iota^{r}\}$ if $\iota \in \mathbb{I}^{F}$, or
  - $\mathfrak{b}' = \mathfrak{b} \cup \{(\iota, \mathsf{v})^{r}\}$ otherwise.

In all the cases, for all $a \in c'_{(j+1)}$ such that $a \notin c_{(j+1)}$, $\mathsf{buffer}\,(a) \in \mathfrak{f}'$ and $\mathsf{process}\,(a) \in \mathfrak{p}'$.

As the initial state of $\mathcal{A}_{part}\,(\mathcal{C})$ is $(\mathfrak{q}^{0}_{\mathcal{C}}, \varnothing, \varnothing, \varnothing)$, the induction hypothesis holds for $w = \varepsilon$. □

Because the partial closure of a choreography is a regular language (Lemma 6.2.1), checking if a system of communicating automata is RSC is decidable (Theorem 4.1.4), and the language of RSC executions of a system is regular (Theorem 3.3.3), we can state the following theorem.

Theorem 6.2.2. *Checking whether a choreography is well-formed is decidable.*

6.3 Properties of well-formed choreographies

If by definition, well-formedness of a choreography ensures that its implementation does not exhibit new behaviours, other interesting properties come with this characterisation.

6.3.1 Deadlock-freedom

A property which is usually associated with safety of distributed systems is *deadlock-freedom*. From any configuration of a system satisfying this property, reachability of a 'final' configuration is guaranteed. A final configuration is a configuration in which all automata are in an accepting state and all the buffers are empty, that is there are no pending message. All our control states are accepting, so we consider stable configurations as final.

Definition 6.3.1 (Deadlock)**.** Let $\mathfrak{S}$ be a system, $\gamma \in RS(\mathfrak{S})$ is a *deadlock* if for all $\gamma' \in RS(\gamma)$, γ' is not stable.

A *deadlock* is a configuration from which no stable configuration is reachable.

Example 6.3.1 – Consider the system of communicating automata $\mathfrak{S}$, whose graphical representation is depicted in Figure 6.7. This system admits a deadlock: if participant r reaches its state 1 while buffer 1 is not empty, no execution will ever lead to a stable configuration.

A system $\mathfrak{S}$ admits a deadlock if there exists $\gamma \in RS(\mathfrak{S})$ such that γ is a deadlock, and is deadlock-free otherwise.

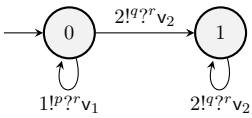


Figure 6.8: Example of $\mathcal{A}_{sync}$

Definition 6.3.2 (Deadlock-Freedom)**.** A system $\mathfrak{S}$ is *deadlock-free* if for all $\gamma \in RS(\mathfrak{S})$, $\exists \gamma' \in RS(\gamma)$ such that γ_s is a stable configuration.

In other words, a system is deadlock-free if no deadlock is reachable. From all reachable configuration of such a system, an execution can lead to a stable configuration.

There is a connection between the notion of deadlock-freedom and synchronous executions. By essence, synchronous execution lead to stable configurations. If all executions of a system are prefixes of one of its synchronous executions, we can be sure that this system is deadlock-free.

The language of synchronous executions of a system of communicating automata is regular. To show this, we begin by defining how to build an automaton $\mathcal{A}_{sync}$ such that $\mathcal{L}\left(\mathcal{A}_{sync}\left(\mathfrak{S}\right)\right) = \mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)$.

Definition 6.3.3 (Automaton $\mathcal{A}_{sync}$)**.** Let $\mathfrak{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of communicating automata such that $\mathsf{product}\left(\mathfrak{S}\right) = \left(L_{\mathfrak{S}}, \mathbb{V}_{\mathfrak{S}}, \mathbb{I}_{\mathfrak{S}}, \mathsf{Act}_{\mathfrak{S}}, \delta_{\mathfrak{S}}, \mathbf{l}^0_{\mathfrak{S}}\right)$, $\mathcal{A}_{sync}\left(\mathfrak{S}\right) = \left(\mathsf{Q}_{sync}, \delta_{sync}, \mathsf{q}^0_{sync}, \mathsf{F}_{sync}\right)$ is a finite state automaton over alphabet $\Upsilon_{\mathfrak{S}}$ such that

- $\mathsf{Q}_{sync} = L_{\mathfrak{S}}$,
- $\mathsf{q}^0_{sync} = \mathbf{l}_0$,
- $\mathsf{F}_{sync} = \mathsf{Q}_{sync}$, and
- $(\mathbf{l}, c, \mathbf{l}') \in \delta_{sync}$ if $(\mathbf{l}, \mathbf{b}^{\varnothing}) \xRightarrow[\mathfrak{S}]{\mathsf{cte}(c)} (\mathbf{l}', \mathbf{b}^{\varnothing})$.

Intuitively, $\mathcal{A}_{sync}$ matches the send actions and the receptions that can happen ‘at the same time’, with all the buffers empty at all time. Its control states are the states of the product of the system, all of them being accepting. It allows a transition from a first control state to a second one if it is labelled by a communication whose send action can be executed from the first control state, and whose reception can happen from the said send action, leading to the second control state.

Example 6.3.2 – Figure 6.8 shows the graphical representation of $\mathcal{A}_{sync}\left(\mathfrak{S}\right)$, for $\mathfrak{S}$ the system of communicating automata from Example 6.3.1. As participant r was the only one having more than one state, $\mathcal{A}_{sync}\left(\mathfrak{S}\right)$ has the same shape as its automaton.

As mentioned earlier, if all executions of a system are prefixes of some of its synchronous executions, this system is deadlock free. Remember that, for an RSC system, the set of RSC executions is equal to the set of all executions, up to causal reordering. This means that, for such a system, checking that its RSC executions are prefixes of its synchronous executions is enough to ensure its deadlock-freedom. To show this, we need the following lemma, stating that the actions missing from a partial execution can be added after it to form an new execution, causally equivalent to the complete one.

Lemma 6.3.1. *Let e_p and e be two executions. If $e_p \trianglelefteq e$, then there exists e' such that $e_p \cdot e' \sim e$.*

Proof.

Let $e = a_1 \cdot \ldots \cdot a_n$ and $e_p a'_1 \cdot \ldots \cdot a'_m$ be two executions, such that $e_p \trianglelefteq e$.

By Definition 6.2.1 we know that there exists a function σ such that for all $i \in \{1, \ldots, n\}$, for all $j \in \{1, \ldots, m\}$, if $i \prec_e \sigma(j)$ then there exists $j' \in \{1, \ldots, j-1\}$ such that $\sigma(j') = i$. Therefore, for all $a_i \in e$ such that for all $j \in \{1, \ldots, m\}, \sigma(j) \neq i$, a can commute at least up to the last action of e_p. □

We now state that equality between the RSC executions of an RSC system, and the partial closure of its synchronous executions, ensures deadlock-freedom of this system.

Lemma 6.3.2. *Let $\mathfrak{S}$ be an RSC system of communicating automata. If* $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) = \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}))$*, then $\mathfrak{S}$ is deadlock-free.*

Proof.

Let $\mathfrak{S} = (\mathcal{A}_p)_{p \in \mathbb{P}}$ be an RSC system of communicating automata, and assume $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) = \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}))$. We will show that for all $e \in \mathsf{executions}(\mathfrak{S})$ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$, there exists a stable configuration γ_s and an execution e_c such that $\gamma \overset{e_c}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma_s$.

Let $e \in \mathsf{executions}(\mathfrak{S})$ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$. Let e' be an RSC execution such that $e' \sim e$. We know that $\mathsf{etc}(e') \in \mathcal{L}(\mathcal{A}_{rsc}(\mathfrak{S}))$ (by Lemma 3.3.1, page 33), and because $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) = \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}))$, we know that there exists $e_t \in \mathsf{executions}_{\mathsf{sync}}(\mathfrak{S})$ such that $e' \trianglelefteq e_t$. By Lemma 6.3.1, this means that there exists e_c such that $e' \cdot e_c = e'_t$, with $e'_t \sim e_t$. By definition of synchronous executions, there exists γ' such that $\gamma_0 \overset{e_t}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma'$, with γ' stable. Therefore, $\gamma \overset{e_c}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma'$, proving our point. □

Notice however that the condition in Lemma 6.3.2 is not necessary for an RSC system to be deadlock-free. Two causally equivalent executions are not necessarily partial executions of the same execution: for e_1, e_2, e_3 three executions over an arbitrary alphabet Λ, if $e_1 \trianglelefteq e_3$, and $e_1 \sim e_2$, it is not always the case that $e_2 \trianglelefteq e_3$. Observing that $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}) \neq \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}))$ does *not* imply that $\mathsf{executions}(\mathfrak{S}) \neq \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}))^{\sim}$. Example 6.3.3 presents a deadlock-free system that dos not satisfies the premise of Lemma 6.3.2.

Example 6.3.3 – Consider $\mathfrak{S}_{rsc}$ from Figure 3.4a. This system is deadlock-free: from any configuration, the execution were either p or q sends its message in order to have the two buffers filled with the same amount of messages, and then r receives all the messages in its buffer, leads to a stable configuration. However, $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}_{\mathsf{rsc}}) \not\subseteq \mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}_{\mathsf{rsc}}))$. Indeed, observe that $\mathcal{A}_{sync}(\mathfrak{S}_{rsc}) = \mathcal{A}_{\mathcal{C}}$ for $\mathcal{C}$ the choreography from Example 6.2.2. As it was discussed in this example, execution e was part of $\mathsf{executions}_{\mathsf{rsc}}(\mathfrak{S}_{\mathsf{rsc}})$, but not in $\mathsf{partials}(\mathcal{A}_{\mathcal{C}})$ and therefore not in $\mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}_{\mathsf{rsc}}))$. This can be visually confirmed with Figure 6.6, which shows the automaton $\mathsf{partials}(\mathsf{executions}_{\mathsf{sync}}(\mathfrak{S}_{\mathsf{rsc}}))$, and Figure 6.6, displaying $\mathcal{A}_{rsc}(\mathfrak{S}_{rsc})$.

Definition 6.2.2 of a well-formed choreography $\mathcal{C}$ does not require that the partial closure of the synchronous executions of $\alpha(\mathcal{C})$ is equal to its RSC executions. However, the following lemma states that this partial closure is equal to the partial closure of $\mathcal{C}$.

Lemma 6.3.3. *Let $\mathcal{C}$ be a well-formed choreography,* $\mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\alpha\left(\mathcal{C}\right)\right)\right) = \mathsf{partials}\left(\mathcal{C}\right)$.

Proof.

Let $\mathcal{C}$ be a well-formed choreography, with $\mathcal{A}_{\mathcal{C}}$ the associated finite state automaton. Let $\mathfrak{S} = \alpha\left(\mathcal{C}\right)$ be the implementation of $\mathcal{C}$.

We begin by showing that $\mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)\right) \subseteq \mathsf{partials}\left(\mathcal{C}\right)$. Let $e \in \mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)\right)$. As it is a partial execution of a synchronous execution, we know that e is RSC. This implies that $e \in \mathsf{executions}_{\mathsf{rsc}}\left(\mathfrak{S}\right)$, because $\mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)\right) \subseteq \mathsf{executions}\left(\mathfrak{S}\right)$. Therefore, as $\mathcal{C}$ is well-formed, by Definition 6.2.2 $\mathsf{executions}_{\mathsf{rsc}}\left(\mathfrak{S}\right) = \mathsf{partials}\left(\mathcal{C}\right)$ so $e \in \mathsf{partials}\left(\mathcal{C}\right)$.

For the other inclusion, remember that by Lemma 6.1.1, $\mathcal{C} \subseteq \mathsf{executions}_{\mathsf{sync}}\left(\alpha\left(\mathcal{C}\right)\right)$. For all $e \in \mathsf{partials}\left(\mathcal{C}\right)$, there exists $e' \in \mathcal{C}$ such that $e \trianglelefteq e'$. As $e' \in \mathsf{executions}_{\mathsf{sync}}\left(\alpha\left(\mathcal{C}\right)\right)$ as well, $e \in \mathsf{partials}\left(\alpha\left(\mathcal{C}\right)\right)$. □

The consequence of the previous lemma is that implementations of well-formed choreographies are deadlock-free.

Theorem 6.3.4. *Let $\mathcal{C}$ be a choreography. If $\mathcal{C}$ is well-formed, then $\alpha\left(\mathcal{C}\right)$ is deadlock-free.*

Proof.

Let $\mathcal{C}$ be a well-formed choreography, and $\mathfrak{S} = \alpha\left(\mathcal{C}\right)$ be its implementation. As $\mathcal{C}$ is well-formed, by Definition 6.2.2 we have that $\mathsf{partials}\left(\mathcal{C}\right) = \mathsf{executions}_{\mathsf{rsc}}\left(\mathfrak{S}\right)$. By Lemma 6.3.3, we know that $\mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)\right) = \mathsf{partials}\left(\mathcal{C}\right)$. Therefore, $\mathsf{partials}\left(\mathsf{executions}_{\mathsf{sync}}\left(\mathfrak{S}\right)\right) = \mathsf{executions}_{\mathsf{rsc}}\left(\mathfrak{S}\right)$, and by Lemma 6.3.2, we can conclude that $\mathfrak{S}$ is deadlock-free. □

6.3.2 RSC implementation

Apart from deadlock-freedom, and therefore the safety properties that are implied by deadlock-freedom, systems of communicating automata obtained by implementations of a well-formed choreography are not guaranteed to have any safety properties. However, as implementations of such choreographies are RSC, regular safety properties can be checked on the implementations. This includes progress (see Section 4.2.4, page 56).

6.4 Discussion

The approach consisting in describing the protocols globally is quite natural. From the design perspective, it is easier to think in terms of sequences of message exchange rather than in terms of behaviour of each participant individually. One difficult aspect of the formalisms allowing to do this is that the distributed system obtained from the global description may present behaviours that were not planned globally. Ensuring there are no such behaviours is the point of the *realisability* problem.

We begin this discussion with an overview of the formalisms allowing to do this, and a comparison between them and our choreographic setting. We will follow with a study of realisability throughout the different formalisms we considered, and we will finish with some comparison between the classes of communicating automata obtained by projection of global types and RSC.

6.4.1 Global description and safety

There are several formalisms allowing to describe globally a communication. We give an overview of three of them: choreographies, MSGs, and multiparty session types. Almost all of them assume a peer-to-peer communication architecture, and some of them rely on a mailbox one ([Basu, Bultan and Ouederni 2012a]). In all the settings we will discuss here, the buffer implied in a message exchange can be deduced from the sender and the receiver. This is not the case for our choreographies, the buffer is a parameter of the communication, and we can describe hypothetical protocols where a buffer would be used as a bus, with several participants sending and receiving messages from it. In the following comparisons, we always consider our choreographies restricted to the same communication architecture as the one of the other formalism.

6.4.1.1 Choreographies

Barbanera et al. defined in [Barbanera et al. 2020] choreography automata, which are very similar to the automata $\mathcal{A}_C$ of choreographies from Definition 6.1.1. Choreography automata can be projected to obtained a local description of the behaviour, using communicating automata. Authors of [Barbanera et al. 2020] characterised well-formedness of choreography automata thanks syntactical properties. Both synchronous and asynchronous semantics were considered for the distributed representation.

A system of communicating automata obtained by projection of a well-formed choreography automaton benefits from three safety properties: *live*, *lock-free*, and reception deadlock-free. Liveness and lock-freedom are stronger properties than progress (Definition 4.2.5), imposing that all the participants that did not reach a final state are able to execute transitions.

More recently, Barbanera et al. defined a generic framework for choreographic languages [Barbanera et al. 2022]. They give a general condition for the projection of choreographies, based on the notion of *closure under unknown information*. Intuitively, a language of communication is closed under unknown information if it contains all words that cannot be distinguished from words it contains by some participants. The results of this work are not entirely applicable to our setting, as the only semantics studied for the local languages is synchronous. The safety properties ensured for well-formed choreographies are not guaranteed to hold in an asynchronous setting.

Choreography conformance is a notion similar to well-formedness of choreography, but from a different angle: its purpose is to ensure that a distributed description of a protocol satisfies a global one (a choreography). In [Basu and Bultan 2011], this problem is addressed with an approach seemingly similar to the one we used to characterise and decide well-formedness of choreographies. Indeed, using a model similar to communicating automata for the formalisation of the distributed description, they relied on the notion of *synchronisability* to decide choreography conformance. As discussed in Section 3.4, the notion of synchronisability is very close to RSC, which we based our well-formed choreographies on. However, remember that, contrary to what was stated in [Basu and Bultan 2011], whether a system is synchronisable is not decidable [Finkel and É. Lozes 2017].

6.4.1.2 Multiparty Session Types

Multiparty session types resemble choreographies. One of their specificity compared to other choreographic settings is the interest in typing programs, and therefore the use of process algebra to reason on these programs. Well-formedness of the global type is often addressed through the projection function (e.g. in [Coppo et al. 2015], [Honda et al. 2008]): a global type is well-formed if

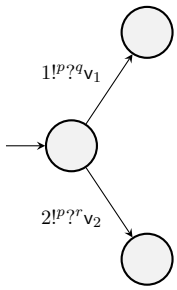


Figure 6.9: Example of a choreography whose implementation does not satisfy progress from [Coppo et al. 2015]

the projection function is defined for it. *Choice* is the mechanism allowing branching. A participant may choose within a list which message to send to another participant. For a given choice, both the sender and receiver are typically fixed: this is called 'directed choice'. In the standard multiparty session type settings, for a global type to be well-formed, all the other participants (those who are not the sender or the receiver of the choice) should behave in the same way, whatever message is chosen. Choreographies do not have the same constraint, which means they can express more behaviours than standard global types.

In exchange for this reduced expressiveness, the composition of processes typed by projection of a well-formed global type benefit from safety properties. Typically, like in both [Coppo et al. 2015] and [Honda et al. 2008], *communication safety*, *protocol fidelity* and *strong progress* are ensured to hold for composition of processes (in π-calculus) typed by local types obtained by projection of a global type. *Communication safety* means that the types of the messages sent and received are compatible, this property is strongly connected with unspecified reception freedom (Definition 4.2.4) in our setting. *Protocol fidelity* (or *session fidelity* in [Honda et al. 2008]) is equivalent to the second point of Definition 6.2.2, it ensures that all the behaviours of the distributed system is accounted for in the global type. This property is related to the realisability problem we will discuss in Section 6.4.2. Finally, 'strong progress' (called 'progress' in Coppo et al.) is a stronger notion than the one from Definition 4.2.5. It imposes that every message sent will eventually be received, and every message expected by a participant will eventually be sent. Definition 6.3.2 of deadlock-freedom, which is ensured for implementation of our well-formed choreographies, corresponds to the first part of these two properties. This is because in a deadlock-free system, from any reachable configuration, a stable configuration is reachable, which means all pending messages can be received. However, our well-formed choreographies do not ensure that all participant expecting a message receive it, as shown in Example 6.4.1 below.

Example 6.4.1 – Consider the choreography $\mathcal{C}$ whose automaton $\mathcal{A}_{\mathcal{C}}$ is depicted in Figure 6.9. It is well-formed, because participant p is the only one sending messages, and it chooses to which participant it sends it, so the RSC language of $\alpha(\mathcal{C})$ is obviously equal to the partial closure of $\mathcal{C}$. However, depending on which transition is executed by p, either q or r will never execute its reception transition.

Several studies on multiparty session types attempted to circumvent the restrictions of the standard projection operator. A notable work in this context is [Majumdar et al. 2021], which defines a generalised projection operator, allowing the sender to choose a receiver in addition of the

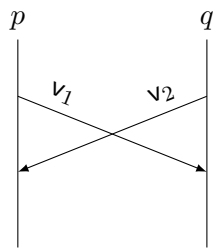


Figure 6.10: MSC with a non synchronous pattern

message to send. A thorough comparison between existing projection operators can be found in [Stutz 2023].

6.4.1.3 MSCs languages

MSGs (for Message Sequence Graphs) [Muscholl and Peled 1999] are a formalism allowing to define languages of MSCs. They are graphs, the nodes of which are labelled with MSCs. The concatenation of MSCs found along a path in the graph are part of the *language of the MSG*. High-level MSCs, or HMSCs in short, were introduced in [ITU-TS 1996]. They are graphs where nodes themselves are labelled by HMSCs. HMSCs and MSGs can be considered equivalent, as an HMSC can always be flattened to be defined as an MSG [Alur and Yannakakis 1999]. An MSG is bounded if in each cycle, the graph a communication (a directed graph where the nodes are the participants, and there is an arc between p and q if p sends a message to q) is strongly connected (excluding the actors not taking part in the communication at all). If this same graph is connected in an undirected manner, the MSG is said *globally cooperative*.

While choreographies can be seen as languages of synchronous executions, MSGs can describe executions that are not synchronous: an MSC on one of the nodes constituting an MSG can contain patterns where two participants must send a message first before receiving the one from the other for example (see Figure 6.10). In this case, the language of executions of the MSG (the set of all linearisation of all the MSCs in the language of the MSG) is not RSC. It also cannot be represented by a choreography, as in this formalism the atomic element is typically a communication. Here, the atomic element is an MSC, and an MSC can describe interleaving of actions from different communications.

In a way, choreographies can be represented by MSGs. Indeed, an MSG can be represented by a graph where the arcs, instead of the vertices, are labelled by MSCs, without changing the expressiveness of the formalism (this is done in [Lohrey 2003] for instance). Limiting labels to MSCs containing one message exchange makes this formalism very close to our automata $\mathcal{A}_C$ (and to choreography automata [Barbanera et al. 2020]). The difference is that in the case of MSGs, the words of the language are MSCs, as opposed to the executions in a language described by a choreography. If we consider an execution recognised by a choreography up to causal closure, choreographies and MSGs restricted in the way we described above are equivalent. In any case, the language of any choreography is included in the language of an MSG.

In [Stutz and Zufferey 2022], an embedding of multiparty session types in HMSCs is defined. The global types studied in this paper are standard ones, with branching and recursion (but no parallel composition). The only addition to the global types from [Honda et al. 2008] is the possibility to send a message to different participants in a choice. Stutz and Zufferey states that the

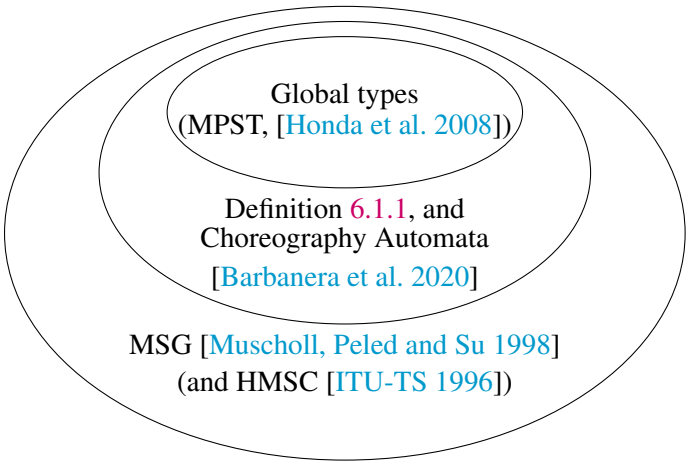


Figure 6.11: Comparison between the expressive power of different formalisms

language of a global type is included in the language of the HMSC it is embedded in, and that only the causal closure of the language of the global type is equal to that of the HMSC. This makes the embedding from [Stutz and Zufferey 2022] similar to the one we sketched in the previous paragraph for choreographies.

Figure 6.11 illustrates the relative expressive power of the different formalisms we discussed.

6.4.2 Realisability problem

To our knowledge, the *realisability problem* was first defined for MSC languages in [Alur, Etessami et al. 2000]. It consists in checking whether there exists a distributed implementation of all the behaviours of a finite set of MSCs, without inducing new ones. Alur, Etessami et al. also introduced a variant of this problem: *safe realisability*, which requires the distributed implementation to be deadlock-free. The communicating automata used as distributed implementation are defined with a discrete set of accepting states, and the notion of deadlock-freedom is defined as the possibility to reach an accepting state from any reachable state of the system. For the finite sets of MSCs considered here, both problems are decidable. These results were extended to generic communication models in [Alur, Etessami et al. 2003]. The communication model is abstracted by a function, enabling or not an action to happen after a given execution.

Realisability was then studied in [Alur, Etessami et al. 2003; 2001] for infinite languages of MSCs, defined as MSGs. For bounded MSGs, safe realisability is decidable, but *weak realisability* is not (the realisability problem is called *weak realisability* to differentiate it from safe realisability). The same results were also extended to non-FIFO semantics in [Morin 2002], that is allowing reception of messages out of order. This setting corresponds to our bag semantics, but is still constrained by a peer-to-peer communication architecture.

Lohrey showed in [Lohrey 2003] that in general safe realisability is undecidable. He also showed that this problem is decidable (and EXPSPACE-complete) for *globally cooperative* MSGs.

The problem we address with our notion of well-formedness can be compared to safe-realisability. We check that the behaviours described by a choreography encompass all those of its implementation, and if our check gives positive results, we are assured that the implementa-

tion is deadlock-free. In the context of this thesis, as choreographies are closed by prefixes, the notion of *deadlock-freedom* from Definition 6.3.2 coincide with the deadlock-freedom for communicating automata with specific accepting states: a stable configuration in an accepting control state must be reachable from any reachable configuration. All our control states are accepting, so being able to reach a stable configuration is the only condition to satisfy.

The problem of realisability was previously studied for choreographies as well [Basu and Bultan 2011; Fu et al. 2004]. These works are quite close to ours as they rely on the notion of synchronisability [Fu et al. 2005] to establish their results. As we mentioned in Section 3.4, this notion is conceptually close to our notion of RSC. Unfortunately, the decidability results obtained relying on synchronisability do not hold, as synchronisability is not decidable [Finkel and É. Lozes 2017].

In [Barbanera et al. 2020] in addition to the safety properties ensured for a well-formed choreography automata (see Section 6.4.1), the language of a well-formed choreography automata is ensured to be the same as that of its projection: well-formedness implies realisability A key difference between realisability of choreographies from [Barbanera et al. 2020; 2022] and from this thesis is that for Barbanera et al., it is ensured by the well-formedness, which is syntactically checked, while in our approach we are immediately checking realisability itself, and we characterise our well-formedness thanks to this property.

To complete this section on realisability, me discuss how this notion is handled for multiparty session types. In these frameworks, a *global type* gives an overall description of the modelled protocol, and its projection on *local types* constitutes the distributed implementation of the same protocol. In such a setting, it is obviously very important to ensure that the combined local types behave accordingly to the global type. This property is called *session fidelity* (or sometimes *protocol fidelity*) throughout the literature on this formalism. Here the question is usually slightly different from realisability: the problem is generally not to check, for a given global type, whether there exists a local type being its exact projection, but rather to ensure that when defined, the projection of a well-formed global type always satisfies session fidelity. This is similar to what we just saw with choreography automata: if the global description (here the global type) satisfies some syntactic conditions, its realisability is ensured. Moreover, projection of well-formed global types are typically ensured to be deadlock-free, among other safety properties. This means that projection of well-formed global types can be related to the problem of safe realisability as defined for MSGs in [Alur, Etessami et al. 2000; 2001].

In recent works, another approach emerged with the definition of *implementability* of a global type [Majumdar et al. 2021]. A global type is *implementable* if there exists a deadlock-free system of communicating automata whose language is the same as the causal closure of the language of the global type (the executions it describes). This definition is equivalent to the definition of safe realisability, and in [Stutz 2023], decidability of the implementability problem was proved to be decidable by showing that the encoding of a global type to an HMSC, as defined in [Stutz and Zufferey 2022], is globally cooperative.

These results were the base of [Li et al. 2023], which defined a new projection function for multiparty session types: a system is obtained in a similar way to Definition 6.1.3, and the projection function is defined if this system is an implementation of the global type. This approach is close to the one we use to ensure well-formedness. We also obtain a system from any choreography, and then check that this system is a correct implementation. However, in [Li et al. 2023], the projection is complete: every implementable global type is accepted. In contrast, one of our choreographies may be realisable but not well-formed.

6.4.3 RSC and typeable systems

In the context of multiparty session types, the relationship between local types and communicating automata have been studied in various works. We call *typeable* a system that can be obtained by projection of a global type. The intuition is that there is a global type typing this system. In the context of our choreographies, even if we cannot exactly call it typing, we will allow the use of this term to describe implementation of well-formed choreographies. In this context, RSC is very clearly the class of typeable systems. In this section we discuss how the different classes of communicating automata defined with the goal of characterising the systems obtained as projection of global types can be compared to RSC.

To our knowledge, the first paper establishing a correspondence between between local types and communicating automata is [Deniélou and Yoshida 2012a]. In this work, Deniélou and Yoshida defined Multiparty Session Automata (MSA for short) as the class of systems obtained by projection of a well-formed generalised global type. Because of the parallel composition in generalised global types, mixed states are possible in MSA: in a given state, a participant may be able to send a message or receive another one. The implementation of our choreographies can have mixed states as well, and an example of that can be found in the automaton of $Server$ in Figure 6.3. This example illustrates how parallel composition can be accounted for in our setting: all the interleaving must be expressed. Because of how MSA are defined, they have some safety properties, like all projection of a well-formed global type. More precisely, they are unspecified reception free, orphan message free, and reception deadlock-free. This makes MSA different form RSC, as no safety property is ensured for RSC systems. However, all the safety properties satisfied for MSA are decidable for RSC systems. Additionally, MSA are shown to have a property called 'stable-output decomposition': each reachable configuration is reachable through a *stable-output* execution. An execution is stable-output if it has an RSC prefix continued by a sequence of unmatched send actions. A stable output execution is trivially RSC, but stable-output decomposition does not mean MSA are necessarily RSC. In fact this notion makes MSA a subclass of eager systems [Heußner, Leroux et al. 2012].

One of the main drawback of MSA is that is not characterised, it relies only on the existence of a generalised global type that can be projected on it. *Multiparty compatibility* [Deniélou and Yoshida 2013] is an attempt to close this gap. As this characterisation relies on the standard global types (without parallel composition), and Deniélou and Yoshida impose a first condition on the systems: all the communicating automata composing it must be *basic*. A communicating automaton is basic if it has no mixed state (states from which both send and receive actions are available), it is deterministic, and it is directed (all the send actions, respectively receptions, from a given states are toward, respectively from, the same participant). A basic system is multiparty compatible if, from any stable configuration reachable through a locally 1-bounded execution, all the sequences of actions possible for a participant can be matched by the rest of the system by doing only alternations. Alternations are basically RSC executions without unmatched messages, but because the condition applies to configurations reachable through locally bounded executions, this condition may not be enough to imply RSC. However, we conjecture that for basic systems, multiparty compatibility implies RSC. Here again, the same safety properties are ensured for multiparty compatible systems as for MSA: unspecified reception freedom, orphan message freedom, and reception deadlock-freedom. In addition to these, multiparty compatible systems benefit from a liveness property: from any reachable configuration, a final configuration is reachable. This is equivalent to our definition

of deadlock-freedom (Definition 6.3.2). We do not know whether this property is decidable for RSC systems, however it is ensured for implementation of well-formed choreographies.

Multiparty compatibility was latter generalised in [Lange, Tuosto et al. 2015a]. *General Multiparty Compatibility* (GMC for short) is to the generalised global types from [Deniélou and Yoshida 2012a] the same as multiparty compatibility is to standard global types. It essentially characterises MSA, GMC systems can therefore have mixed states. To be part of this class, systems must satisfy two conditions, one of them being that all sequences of actions a participant can produce must be part of the synchronous language of the system. This condition led us to believe that these systems should be RSC, but it turns out it is not the case. Indeed, [Lange, Tuosto et al. 2015a] is supported by a tool, and we could run the tool we developed to check whether a system is RSC (see Chapter 7) on examples that are GMC to find out that some of them are not RSC.

To finish this overview, we mention a newer evolution of multiparty compatibility: k-multiparty compatibility [Lange and Yoshida 2019a]. This characterisation is based on existential boundedness, and is parametrised by the bound k. This time, the global types considered are the standard ones again, so k-multiparty compatible systems cannot have mixed states. Contrary to general multiparty compatibility and multiparty compatibility, k-multiparty compatibility is not decidable. However Lange and Yoshida provide decidable conditions under which a system is k-multiparty compatible. In a 1-multiparty compatible system, patterns where two participants start by sending each other a message before receiving the one from the other are accepted. This means that these systems are not RSC.

Chapter 7

Tool support

ReSCu, for **Re**alisable with **S**ynchronous **C**omm**u**nication, is a tool allowing to check whether a system of communicating automata is RSC, and reachability of a regular set of configurations through RSC executions. It relies on the results presented in Chapter 4. In this chapter, we will present this tool, and discuss some results we obtained with it. We will discuss its performance as well, and compare it to that of other tools. We begin this chapter by providing an overview of the existing tools. This allows us to compare our work with some of them later. ReSCu was introduced in [Desgeorges and Germerie Guizouarn 2023], and can be found at [SW, Germerie Guizouarn 2023].

7.1 Related tools

One of the closest tools to ReSCu is McScM [Heußner, Le Gall et al. 2012]. It takes a description of a system and a set of bad configurations and checks whether a bad configuration is reachable. It uses a description language for the representation of systems called SCM, for System of Communicating Machines. ReSCu uses the same input language. McScM was designed as a framework, with four components implementing model-checking approaches based on abstract interpretation: Abstract Interpretation (`absint`) [Gall et al. 2006], Abstract Regular Model Checking (`armc`) [Bouajjani, Habermehl et al. 2004], Counterexample Guided Abstraction Refinement (`cegar`) [Clarke et al. 2003; Heußner, Gall et al. 2009], and a lazy abstraction approach (`lart`) developed in [McMillan 2006]. The two last model-checking engine, `cegar` and `lart`, are parametrised by a trace checker and a invariant generator, providing four variants for each of these engines. Among all the algorithms, `absint` is the only one to always terminate. The other ones are semi-algorithms and will provide counter examples for unsafe protocols, but they may run infinitely checking safe ones.

One of the most prominent model-checker for systems of communicating automata is SPIN [Holzmann 1997]. This tool differs from ReSCu as it only allows to verify systems whose buffers are bounded, with the bound known. RSC systems may be unbounded, and for any system, ReSCu allows to check whether it is RSC, not requiring any information to be known *a priori*. A strength of SPIN is that it allows to check properties specified in linear temporal logic, while ReSCu does not implement any temporal logic.

We mentioned *stability* in previous discussions (Sections 3.4 and 4.3), STABC is a tool built around this notion [Akroun and Salaün 2018]. It allows to check whether a system is stable or not. As stability is not decidable in general, this tool may not provide a conclusive answer for some systems. It uses CADP [Garavel and Thivolle 2009] as a back-end for equivalence checking. Like ReSCu, it works with systems using bag or FIFO buffers, but the only communication architecture it considers is mailbox.

Another tool focused on checking membership of a class is KMC [Lange and Yoshida 2019a]. This tool checks for various properties, which when combined together ensure the system is unspecified reception free and satisfies progress (no participant is ever stuck in a reception state). Tool support can also be found for a related class of communicating automata, which we discussed previously: generalised multiparty-compatibility [Lange, Tuosto et al. 2015a]. The notion of k-multiparty compatibility was more recently used as the background of `kmclib` [Imai, Lange et al. 2022]. This tool is an OCaml library allowing to enforce safety of message passing programs, thanks to an encoding of multiparty session types in the language [Imai, Neykova et al. 2020].

7.2 Features and implementation

We now present ReSCu, its features, the input language it uses, and its implementation.

7.2.1 Features

ReSCu allows to check both membership of the class of RSC systems and reachability of a regular set of configurations through an RSC execution. It is built around the semantics we used throughout this thesis: it can take as input systems with any topology, and any combination of FIFO and bag buffers.

ReSCu is a command line tool: `rescu -isrsc <system>` checks whether the system described in the SCM file `<system>` is RSC, and `rescu -mc <system>` checks that no bad configuration is reachable. The results are displayed on the standard output. The two options can be combined in one call to ReSCu. Option `-fifo` specifies that all buffers should be considered as FIFO buffers, overriding specifications of bag buffers. When a system is not safe, using `-counter` in conjunction with `-mc` provides the user with an RSC execution leading to the bad configuration that was found reachable. The same option, used with `-isrsc`, outputs the borderline violation that was found, in the case the tested system was not RSC.

For convenience, the progression of the computation can be displayed while performing model-checking. This allows to estimate the remaining time during long computations. Another interesting secondary feature is the `-to_dot` option, allowing to output a DOT representation of a the input file, to visualise the systems of communicating automata.

7.2.2 SCM description language

We chose the SCM language used in [Heußner, Le Gall et al. 2012] as an input format. This allowed to compare easily ReSCu with this tool.

A grammar of a part of this language is displayed in Grammar 7.1, we omit some features of the language that are not implemented in ReSCu. In the syntax we chose, <*ident*> can contain either letters or numbers, and must start with a letter. We write <*int*> when we require a positive integer number. As usual, choices are represented by a vertical bar, `a | b` meaning that either `a` or `b` should be present, and elements in brackets are optional.

To declare a system, a name must be provided with `scm <name>`. The next element is the declaration of the number of buffers: `nb_channels: <number>`, followed by the optional special comment declaring the list of identifiers of buffers to be treated according to bag semantics (`//# bag_buffers = <id 1>, <id 2>, ...`). The identifier of a buffer is its number, from 0 to the number of buffer minus 1. We rely on a special comment to keep compatibility with

```
prog          ::=  <header> <aut_list> [<bad_confs>]
header        ::=  scm <ident>:<channels> [<bags>] <parameters>
channels      ::=  nb_channels = <int>;
bags          ::=  //# bag_buffers = <int_list>
int_list      ::=  <int>
                 |  <int_list>, <int>
parameters    ::=  parameters = <param_list>
param_list    ::=  <param>
                 |  <param> <param_list>
param         ::=  {int  |  real} <ident>;
aut_list      ::=  automaton <ident>:<initial>; <state_list>
initial       ::=  initial:<int_list>;
state_list    ::=  <state>
                 |  <state_list> <state>
state         ::=  state <int> :  <trans_list>
trans_list    ::=  <transition>
                 |  <trans_list> <transition>
transition    ::=  to <int>:when true , <int> <action> <ident>
action        ::=  "!"  |  "?"
bad_confs     ::=  bad_states:  <bad_list>
bad_list      ::=  (<bad_conf>)
                 |  <bad_list> (<bad_conf>)
bad_conf      ::=  <bad_state>
                 |  <bad_state> with <bad_buffers>
bad_state     ::=  automaton <ident>:  in <int>:  true [<bad_state>]
bad_buffers   ::=  <regular_expression>
```

Grammar 7.1: Simplified SCM grammar

McScM: if we added an actual instruction in the language, we could not run this tool on files with specified bag buffers for ReSCu. To conclude the preliminary declarations, the list of the names of the messages that will be used must be declared, after the keyword `parameters`. Each of them is assigned a type (either `real` or `int`), which is ignored by ReSCu, but was kept to preserve compatibility with McScM.

Each automaton of a system is given a name, before the list of its initial states is declared thanks to the keyword `initial`. Here, SCM differs from the semantics we introduced in Chapter 2, as a communicating automaton (and therefore the product of a system of such automata) may have more than one initial state. This clearly does not prevent this language from being used to model the systems according to the semantics we considered. The states of an automaton are declared thanks to the keyword `state`, and are identified as integers. After each state, an optional list of transitions may be declared. Each transition is of the form `to <state>: when true, <act>` where `<state>` is a state identifier that must be declared, and `<act>` follows the syntax of communicating automata actions we used throughout this thesis. The `when true` statements are there as a remainder of McScM, and are kept there only for compatibility. There purpose was to have some values assigned to the message names, and to be able to have guards on the transitions referencing them.

The definition of bad configurations differs a little bit from the encoding defined in Definition 4.2.1, page 48. With the syntax of SCM, it is not possible to declare the control state as part of a regular expression. Instead, for each specification of a bad configuration, a local control state must be specified for one or more participants. For instance, a specification `automaton p: in 0: true automaton q: in 1: true` describes configurations where automaton `p` is in state 0 and automaton `q` is in state 1. If there were more participants than the two mentioned in the specification, this example of bad configuration would match any global control state satisfying the constraint for `p` and `q`, regardless of the state of the other participants. The specification of the buffer contents of bad configurations is similar to the encoding of Definition 4.2.1. Regular expression, where as usual, `a.b` stands for the concatenation of messages (or expressions) `a` and `b`, `a|b` stands for the disjunction between expressions `a` and `b`, `a^*` stands for the iterative closure of expression `a`, and `a^+` stands for `a.a^*`.

Example 7.2.1 – Figure 7.1 shows the SCM description of the system in Example 2.2.2. The bad states of this listing corresponds to the unspecified receptions of the system, described in Example 4.2.7.

7.2.3 Implementation

ReSCu is a stand-alone tool implemented in OCaml. While it first reused the parser of McScM, it now benefits from its own, avoiding a step of translation between the data structures of this tool and the one we needed. The only element of McScM that ReSCu relies on is the regular expression data structure.

For both reachability and membership, the algorithms that are presented in Chapter 4 consist in checking the emptiness of the intersection of two finite state automata. While this was a convenient approach for proving decidability and showing the complexity of these problems, it is not ideal for the implementation. Indeed, this method imposes to build entirely both automaton $\mathcal{A}_{rsc}$, and either $\mathcal{A}_{bv}$ for membership, or $\mathcal{A}_{ep}$ for reachability, before computing their intersection and providing an answer. This approach would lead to computation times that are always equal to the worst case.

In ReSCu, the intersection itself is built step by step, allowing to end early if a borderline violation or bad configuration is found. For both problems, a transition function is defined for the intersection of the automata, and a breadth first search is performed to find an accepting state. An advantage of using breadth first search is that the counter-examples provided are minimal in length.

Another aspect of the theoretical algorithms that required some optimisation in the implementation is the set of initial states of automaton $\mathcal{A}_{ep}$. The term 'initial state' is abusive, but remember that in this automaton, there is a set of states reachable through an epsilon transition from the actual initial state (Definition 4.2.2, page 51). We refer to states of this set as initial states here. There are as many initial states as there are combinations of $|\mathbb{I}|$ control states in the automaton $\mathcal{A}_{P(\mathfrak{S})}$. The size of this set is exponential with respect to the number of buffers. However, many of these states cannot lead to an accepting state of $\mathcal{A}_{ep}$. Indeed, remember that for a control state of $\mathcal{A}_{ep}$ to be accepting, the letter $\#$ must be accepted from states of $\mathcal{A}_{P(\mathfrak{S})}$ marked by each pebble, *and* reading this letter must lead to the initial position of the next pebble.

We implemented two slight optimisations of the algorithms from Section 4.2. The first one was to restrict the position of the first pebble to the initial state of $\mathcal{A}_{P(\mathfrak{S})}$. This was quite natural because in the implementation, $\mathcal{A}_{P(\mathfrak{S})}$ describes only buffer contents, and there is only one target control state. The second optimisation we made was to restrict the initial position of all pebbles

```
scm client_server_database :

nb_channels = 3;
parameters: int req; int res; int log; int ack;

automaton server:
initial: 0
state 0:
to 1: when true, 0 ? req;
state 1:
to 2: when true, 1 ! res;
state 2:
to 3: when true, 0 ? ack;
state 3:
to 0: when true, 2 ! log;

automaton database:
initial: 0
state 0:
to 0: when true, 2 ? log;

automaton client:
initial: 0
state 0:
to 1: when true, 0 ! req;
state 1:
to 2: when true, 1 ? res;
state 2:
to 0: when true, 0 ! ack;

bad_states:
(automaton client: in 0: true automaton server: in 1: true)

(automaton server: in 0: true with
   (log|res|ack).(req|res|log|ack)^*.#.(req|res|log|ack)^*.#.
   (req|res|log|ack)^*)
```

Figure 7.1: SCM representation of Example 2.2.2

but the first one to states reachable through a transition labelled by a letter #. In practice, we have been able to roughly halve our model-checking time with this technique.

The same idea could be used for further optimisation. Observe that in any word matching a regular expression describing buffer contents, the number of occurrences of the letter # must be exactly $|\mathbb{I}| - 1$. This implies that this letter cannot be found in a stared expression. Therefore we can match statically each state of $\mathcal{A}_{P(\mathfrak{S})}$ reachable through a transition labelled by # with a buffer identifier.

7.3 Protocol library

To us, the most interesting use of ReSCu was to check the existence of actual systems member of the RSC class. In order to do so, we ran our tool on the set of SCM files provided with McScM. But even though these example were varied and very useful to us, they were too few to be considered a representative set of actual systems. To widen our test sample, we ported the protocols that were available with both KMC and STABC to SCM.

KMC allows two input format, one is a textual representation of global types, and the other is a textual language similar to SCM. We only focused on the latter. A noticeable difference is that instead of specifying the buffer to or from which messages are sent or received, the identifier of the other participant of the communication is mentioned. As a peer-to-peer communication architecture is assumed, the identifier of the buffers of a communication can be deduced from the pair of participants taking part in it. We used a tool to automatically port the examples of protocols.

Porting examples from STABC was slightly more challenging, as in the labels of the transitions in the AUT language, there are no mention to the buffer, or to the other participant involved in the communication. As in [Akroun and Salaün 2018], all the automata are considered with a mailbox communication architecture, each message can be assigned to a buffer identifier as long as two different participants do not receive the same message name. If, in addition to this constraint, no two different participants send the same message, messages can be assigned to a buffer assuming a peer-to-peer architecture as well. In fact, most of the protocols from STABC satisfy both these conditions. We used to port these protocols. This utility allows to interpret AUT files according to a mailbox or a peer-to-peer communication architecture. In the rest of this chapter, we only discuss the results we obtain with the protocols we ported with a mailbox architecture.

Ultimately, we collected a set of more than three hundred examples of protocols, including some real life and well known protocols, e.g. POP3 or SSH. We ran ReSCu on these example, as well as the tool they came with on their original version to reproduce the results published in [Akroun and Salaün 2018] and [Lange and Yoshida 2019a]. The proportion of examples we found to be RSC is 60% of the protocols we ported from KMC, 38% of the protocols that came with STABC, and 30% of the ones distributed with McScM. When using exclusively bag buffers, these results are respectively 41%, 11%, and 12%. It was expected that using bags instead of FIFO buffers would reduce the proportion of RSC systems. Intuitively, bag buffers make it easier to build a borderline violation, because they allow more receptions to happen.

To give a better insight into the representativeness of RSC systems, we reproduce the results from both [Lange and Yoshida 2019a, Table 2] and [Akroun and Salaün 2018, Table 2], in Table 7.2 and Table 7.1 respectively. We added in these tables the results of ReSCu, showing that a significant portion of these protocols, coming from the literature, are RSC.

In addition to these tables, we ran ReSCu on the selected protocols from STABC using FIFO buffers rather than bag, and we provide the results of both our tool and STABC with this setting in Table 7.3. Once again, the fact that there are more RSC systems when using FIFO buffers was expected. However we are quite surprised by the fact that the result are identical with FIFO and bag buffers for STABC. We think that this might be an issue with STABC, but the required investigation is left as future work.

Protocol	$\lvert\mathbb{P}\rvert$	S	T	RSC	t_{rsc}	k-MC	t_{kmc}
Client-Server-Logger [1]	3	11	12	No	3	Yes	17
4 players game [2]	4	13	16	Yes	13	Yes	20
Bargain [2]	3	9	8	Yes	4	Yes	35
Filter collaboration [3]	2	6	10	Yes	4	Yes	33
Alternating bit [4]	2	12	15	Yes	9	Yes	24
TPMContract v2 [5]	2	10	14	Yes	4	Yes	31
Sanitary agency [6]	4	25	30	Yes	15	Yes	39
Logistic [7]	4	26	26	Yes	8	Yes	32
Cloud system v4 [8]	4	14	16	Yes	6	Yes	22
Commit protocol [9]	4	12	12	Yes	4	Yes	15
Elevator [9]	3	13	23	No	7	Yes	41
Dev system [10]	4	22	23	Yes	7	Yes	17
Fibonacci [11]	2	6	6	Yes	3	Yes	17
SH [11]	3	22	30	Yes	18	Yes	33
Travel agency [11]	3	17	20	Yes	8	Yes	15
SMTP [12][11]	2	64	108	Yes	17	Yes	34
HTTP [13]	2	12	48	Yes	17	Yes	28

Table 7.1: Comparison between the membership results of ReSCu and KMC. $\lvert\mathbb{P}\rvert$ is the number of participants, S the number of states, and T the number of transitions. t_{rsc} and t_{kmc} are the time (in ms) of execution of ReSCu and KMC respectively.

7.4 Performance

To evaluate performance of ReSCu, we compared its computation time for reachability with McScM, which is the only tool we studied performing this operation. We also crafted special examples in order to show how the size of the input affects computation time of both membership and model-checking. Finally, we compare the time required to check membership of the class of RSC systems and that of the class of stable systems, and we discuss the computation times of KMC.

All the tools were ran on the same desktop computer with an AMD Ryzen™ 5 3600 CPU, and equipped with 16Gb of RAM. The times displayed here are the mean between three consecutive runs, except from the examples of protocols timing out, which were ran only twice to confirm the timeout. Timeout was set to 30 minutes.

7.4.1 Comparison with McScM

McScM is a collection of algorithms to compute reachability of a regular set of configurations. We compared model-checking time of these algorithms with ReSCu by running them on the subset of RSC protocols that came with McScM, plus one of them which is not RSC: tcp_error. We added this example as an illustration of the fact that ReSCu can find bad configurations in some unsafe systems even if they are not RSC. We added to this set four examples that came with STABC and KMC. Because these two tools do not implement reachability, their examples do not come with specifications of bad configurations, so we had to define some. For each example, we crafted the specification of an unspecified reception. We did so in the same way as the second bad state is defined in Example 7.2.1.

Protocol	$\lvert\mathbb{P}\rvert$	S	T	RSC	t_{rsc}	k	t_{stabc}
Estelle specification [14]	2	7	9	No	3	max	100,461
News server [15]	2	10	10	No	3	3	34,927
Client/Server [16]	2	6	10	**Yes**	3	1	16,713
CFSM system [14]	2	6	7	No	3	max	98,597
Promela program (1) [17]	2	6	6	No	3	2	25,825
Promela program (2) [18]	2	6	7	**Yes**	3	max	97,852
Trade system [19]	3	12	12	**Yes**	3	1	22,499
FTP transfer [20]	3	20	17	**Yes**	3	4	57,469
Client/Server [21]	3	15	15	**Yes**	3	2	34,071
Mars explorer [22]	3	36	34	No	3	3	48,160
Online computer sale [23]	3	26	26	**Yes**	3	2	35,269
E-museum [24]	4	19	24	No	5	3	57,709
Client/supplier [25]	3	31	33	**Yes**	3	2	34,300
Restaurant service [26]	3	16	16	No	4	2	34,367
Travel agency [27]	3	34	38	No	3	4	65,593
Vending machine [28]	3	15	14	**Yes**	3	2	34,326
Travel agency [29]	3	43	56	No	4	3	46,674
Train station [30]	4	20	18	No	3	2	42,969
Factory job manager [31]	4	20	20	No	3	2	42,698
Bug report repository [32]	4	11	11	No	3	-	T_{max}
Cloud application [8]	4	8	10	No	4	max	161,496
Sanitary agency [33]	4	35	42	**Yes**	3	3	57,949
SQL server [34]	4	33	38	**Yes**	3	3	58,686
SSH [35]	4	27	28	**Yes**	3	1	28,329
Booking system [36]	5	45	50	**Yes**	3	2	51,982

Table 7.2: Comparison between the membership results of ReSCu and STABC, using bag buffers and 'strong equivalence'. $\lvert\mathbb{P}\rvert$ is the number of participants, S the number of states, and T the number of transitions. max means the arbitrary limit for k, set at 10, was reached. t_{rsc} and t_{stabc} are the time (in ms) of execution of ReSCu and STABC respectively.

The result of this comparison are displayed in Table 7.4. All the tools agreed on all the protocols (except when they did not provide an answer), and all the protocols but tcp_error were found safe. For `cegar` and `lart`, we chose to represent only the best time of all the variants. Note that this means that a timeout in one of these columns means that four runs timed out. Immediately, it is obvious that ReSCu is faster than McScM. For all the protocols we included in our test, it was the fastest tool, and sometimes by a substantial margin. An important remark however: for all the protocol we have tested McScM on, there is always one of its algorithms that finishes in a short time (less than a second). This means that even if ReSCu is faster, it is not fast enough to make McScM irrelevant, especially considering that the latter is able to provide reachability results for some non RSC systems.

A drawback of McScM however is that its best performing algorithm is not always the same. This can be seen in Table 7.4 already, and is emphasised by Table 7.5, in which the times of the

Protocol	$\|\mathbb{P}\|$	S	T	RSC	t_{rsc}	k	t_{stabc}
Estelle specification [14]	2	7	9	No	3	max	101,775
News server [15]	2	10	10	No	4	3	35,197
Client/Server [16]	2	6	10	**Yes**	3	1	16,765
CFSM system [14]	2	6	7	No	3	max	98,932
Promela program (1) [17]	2	6	6	No	3	2	26,042
Promela program (2) [18]	2	6	7	**Yes**	3	max	98,365
Trade system [19]	3	12	12	**Yes**	3	1	22,548
FTP transfer [20]	3	20	17	**Yes**	4	4	59,664
Client/Server [21]	3	15	15	**Yes**	3	2	34,271
Mars explorer [22]	3	36	34	**Yes**	4	3	48,631
Online computer sale [23]	3	26	26	**Yes**	3	2	34,709
E-museum [24]	4	19	24	**Yes**	6	3	58,441
Client/supplier [25]	3	31	33	**Yes**	3	2	34,839
Restaurant service [26]	3	16	16	No	4	2	34,706
Travel agency [27]	3	34	38	**Yes**	3	4	65,657
Vending machine [28]	3	15	14	**Yes**	4	2	34,667
Travel agency [29]	3	43	56	No	4	3	47,132
Train station [30]	4	20	18	**Yes**	3	2	43,225
Factory job manager [31]	4	20	20	**Yes**	3	2	43,251
Bug report repository [32]	4	11	11	**Yes**	4	-	T_{max}
Cloud application [8]	4	8	10	No	5	max	202,510
Sanitary agency [33]	4	35	42	**Yes**	3	3	58,398
SQL server [34]	4	33	38	**Yes**	4	3	59,083
SSH [35]	4	27	28	**Yes**	3	1	28,474
Booking system [36]	5	45	50	**Yes**	4	2	52,122

Table 7.3: Comparison between the membership results of ReSCu and STABC, using FIFO buffers and 'strong equivalence'. $|\mathbb{P}|$ is the number of participants, S the number of states, and T the number of transitions. max means the arbitrary limit for k, set at 10, was reached. t_{rsc} and t_{stabc} are the time (in ms) of execution of ReSCu and STABC respectively.

variants of `cegar` and `lart` respectively are detailed. The performances of each algorithm vary greatly from one protocol to another.

7.4.2 Evaluation benchmark

To evaluate how the computation time of verification and membership evolves in function of the size of the input, we crafted sets of parametric RSC systems. These systems are as represented in Figure 7.2. It is a kind of ring protocol, where each of the p participants start by sending n messages to the next one, and then receive the same number of messages from the previous. The first participant has a different behaviour: it starts by sending its messages before receiving, avoiding a deadlock. These systems use a mailbox communication architecture, so in each of them, there are as many buffers as there are participants. All the systems built this way are RSC, regardless their size. This makes them suitable for benchmarking our membership implementation,

Protocol	$\lvert\mathbb{P}\rvert$	S	ReSCu	absint	armc	cegar	lart
ring	2	20	**5**	(16,999)	240,426	323	1,437
NonRegular	2	6	**3**	53	T_{max}	13	12
pop3	2	43	**5**	622	1,909	5,568	T_{max}
Nested	2	5	**4**	5	12	284	1,803
con_disc_reg	2	4	**4**	(17)	5	9	6
tcp_error*	2	28	**9**	(88)	22	45	11
http-fsm	2	12	**4**	43	T_{max}	T_{max}	T_{max}
smtp	2	64	**3**	133	81	22	73
FTP	3	20	**5**	22	46	51	58
SSH	4	27	**3**	466	310	147	687

Table 7.4: Model-checking time of McScM and ReSCu in ms. Examples marked with a * are not safe with respect to their specifications, all the others are told to be safe by all the tools. $\lvert\mathbb{P}\rvert$ and S are the number of processes and states of the system respectively. Numbers in brackets indicate inconclusive runs. T_{max} indicates timeouts. Numbers in bold indicate the best time for each protocol.

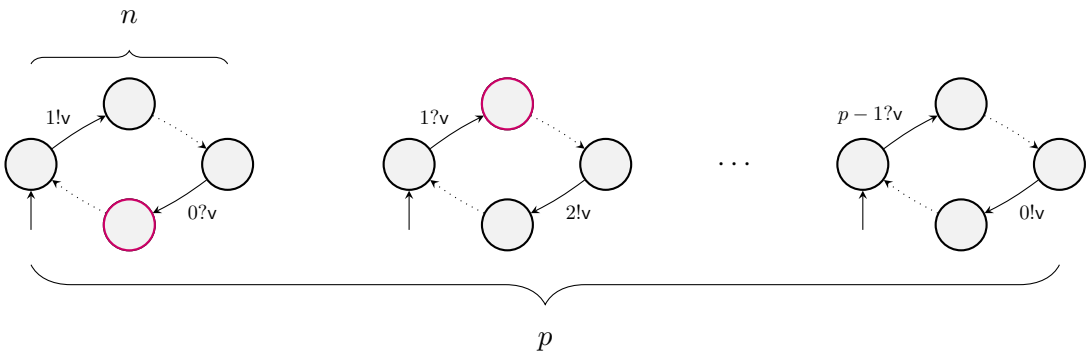


Figure 7.2: Illustration of parametric systems for benchmarking

as they represent a worst case for our tool. Indeed, to ensure a system is RSC, ReSCu must build the entire automaton $\mathcal{A}_{bv} \cap \mathcal{A}_{rsc}$.

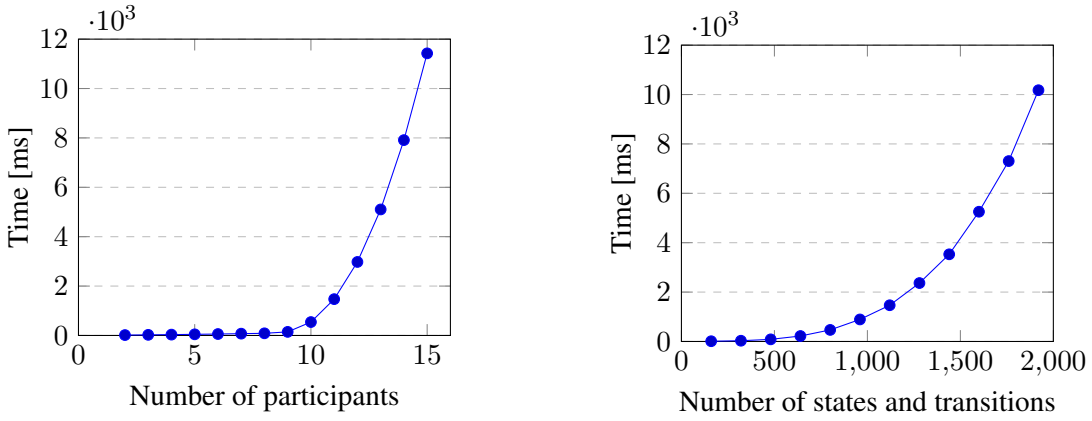
To use the same systems to test reachability time, we needed to specify some bad configurations for them. As for membership, the worst case for ReSCu when checking reachability is when no bad configuration is reachable, so that is what we aimed for. We settled on specification of control states only: the ones where both the first and second automata received their first message. This configuration of control states is illustrated by the highlighted states in Figure 7.2.

Figure 7.3 shows how the time for checking whether a system is RSC evolves with the size of this system. Figures 7.3a and 7.3b show the impact of the number of participants and the number of transitions respectively. These graphics confirm visually the complexity we discussed in Section 4.1: checking membership of RSC systems is exponential in the number of participants (and number of buffers, which are equal here), and polynomial in the number of states and transitions.

Similar observations can be made for the reachability problem, for which the results are shown in Figures 7.4a and 7.4b. We can observe that checking reachability of a control state is a lot faster than checking membership. This is expected as $\mathcal{A}_{bv}$ embeds in its set of states all the combinations

Protocol	$\|\mathbb{P}\|$	S	cegar				lart			
			apinv		upinv		apinv		upinv	
			bwd	fwd	bwd	fwd	bwd	fwd	bwd	fwd
ring	2	20	8,972	**323**	103,909	1,836	T_{max}	**1,437**	T_{max}	49,887
NonRegular	2	6	**13**	24	27	26	**12**	17	190	24
pop3	2	43	**5,568**	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}
Nested	2	5	**284**	8,617	575	469	T_{max}	50,203	T_{max}	**1,803**
con_disc_reg	2	4	12	**9**	14	10	**6**	**6**	9	7
tcp_error*	2	28	84	**45**	242	50	18	**11**	31	12
http-fsm	2	12	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}	T_{max}
smtp	2	64	23	**22**	**22**	**22**	**73**	74	**73**	74
FTP	3	20	151	88	506	**51**	358	122	1,849	**58**
SSH	4	27	527	292	226,601	**147**	1,679	1,359	T_{max}	**687**

Table 7.5: Details of `cegar` and `lart` results



(a) Depending on the number of participants

(b) Depending on the number of states and transitions

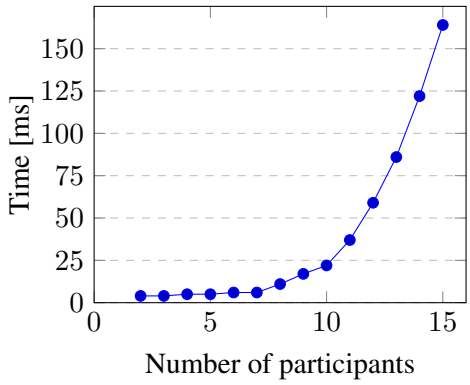
Figure 7.3: Computation time of membership depending on the size of the input

of unmatched send paired with a communication, while with the optimisation we mentioned in Section 7.2.3, checking reachability of a control state requires only an automaton of the size of $\mathcal{A}_{rsc}$.

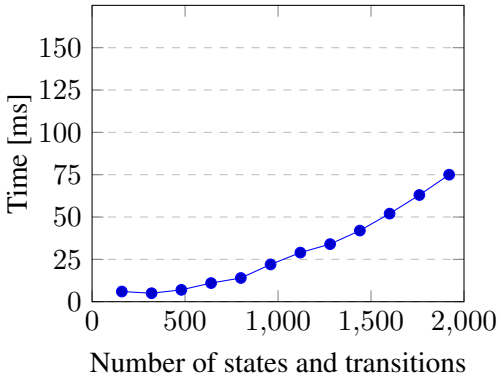
7.4.3 Comparison with STABC and KMC

Both STABC and KMC only allow to check membership of a class of communicating automata: stable and k-multiparty compatible systems respectively. Checking stability of a system allows to know whether bounded model-checking can be used to address the reachability problem, while multiparty compatibility implies some safety properties.

The columns t_{rsc} and t_{stabc} of Tables 7.2 and 7.3 give an insight about the difference in performance of ReSCu and STABC. The huge difference in time between the two tools to perform a comparable operation could come from the fact that STABC is not a native tool, and it relies on



(a) Depending on the number of participants



(b) Depending on the number of states and transitions

Figure 7.4: Model-checking time depending of the size of the input

another tool to perform the actual checking. This introduces some overhead, which could explain why the fastest examples are treated in more than 16 seconds.

The time required to check multiparty compatibility is displayed in column t_{kmc} of Table 7.1, where it is compared to the time of checking membership of the class of RSC systems (column t_{rsc}). KMC performs well, and although slightly slower, its computation time is comparable to the membership time of ReSCu. Note however that membership of the class of k-multiparty compatible systems ensures safety properties, like unspecified reception freedom for instance. In contrast, membership of the class of RSC systems only means that verification techniques described in Chapter 4 can be applied to ensure safety. Deciding similar safety properties as KMC is possible with ReSCu, but would require additional verification time.

7.5 Perspectives

The most obvious direction for ReSCu is the integration of our work on choreographies. Implementing the techniques we developed for this formalism would be straightforward, as they rely on finite state automata, like membership and reachability.

For an improvement of the current state of ReSCu, a nice feature could be the automatic generation of the specifications of bad configurations. For instance, generating automatically the description of all the potential unspecified receptions of a system (where an automaton is in receiving state, and the buffers it receives from begin by a message they cannot receive) could be useful. In addition to helping the user, it would decrease the likelihood of human error while crafting the specifications of a system.

CHAPTER 8

Conclusion and perspectives

In this thesis, we studied a class of systems of communicating automata: the systems whose executions are all Realisable with Synchronous Communications (RSC for short). The definition we provided for this class is based on the RSC computations from [Charron-Bost et al. 1996], extended to account for bag buffers in addition to FIFO ones. The characterisation of RSC systems relies only on the definition of matching pairs in an execution, and does not depend on the communication architecture. All the results we obtained around this class are applicable to systems of communicating automata without any condition on the buffers: any participant can in principle read and write from and to any buffer.

We showed that we can compute a finite representation of the executions of an RSC system, and we relied on this result to show that both membership to the class, and reachability of a regular set of configuration in RSC systems, are decidable. This implies that a lot of interesting safety problems are decidable, like unspecified reception freedom, or reception deadlock freedom. As future work, these results could be extended to other semantics for buffers, like 'lossy channels' from [Schnoebelen 2002] (were messages can randomly disappear from a buffer). Another path for generalisation could be the study of fully-bag definitions of matching pairs (and causal equivalence), allowing reordering of actions of the same type implying the same message on the same bag buffer.

We implemented the techniques we used to obtain these decidability results in a tool: ReSCu, and we studied extensively its performances. We gathered a large database of examples of protocols from the literature for this tool, which allows us to claim that many actual protocols are RSC.

We studied another class of systems of communicating automata: half-duplex systems, that were introduced in [Cécé and Finkel 2005]. In this work, binary half-duplex systems were studied, and because their reachability space is regular, all regular safety properties were shown to be decidable for them. Some generalisations to multiple participants were studied, but they are either too restrictive, or the reachability problem is not decidable for them. We studied a new generalisation, for which the reachability problem ended up being undecidable as well. We provided a proof for this undecidability. Lastly, we established RSC system as a good generalisation to any number of participants of the class of binary half-duplex systems.

Finally, we defined a choreographic setting, and studied its realisability by relying on properties inherent to the notion of RSC. Contrary to most of the works on this topic, the communication architecture is not fixed in our approach, and which buffer is used to exchange a message between two participants is not fixed. We used this definition as a basis to study the notion of realisability across a wide range of global description formalisms, from languages of MSCs to multiparty session types.

Expanding the capabilities of ReSCu to integrate the work we have done on choreographies would be an interesting task to tackle. This would mean making this tool able to accept choreographies as input, to check various safety properties on their implementation, and adding the possibility to check whether a choreography is well-formed.

Bibliography

Parosh Aziz Abdulla and Bengt Jonsson. 1996a. 'Undecidable Verification Problems for Programs with Unreliable Channels'. *Inf. Comput.*, 130, 1, 71–90. DOI: 10/dt6tk5.

Parosh Aziz Abdulla and Bengt Jonsson. 1996b. 'Verifying Programs with Unreliable Channels'. *Inf. Comput.*, 127, 2, 91–101. DOI: 10/fgw8h9.

Lakhdar Akroun and Gwen Salaün. 2018. 'Automated verification of automata communicating via FIFO and bag buffers'. *Formal Methods Syst. Des.*, 52, 3, 260–276. DOI: 10/gsmrgs.

Lakhdar Akroun, Gwen Salaün and Lina Ye. 2016. 'Automated Analysis of Asynchronously Communicating Systems'. In: *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings* (Lecture Notes in Computer Science). Ed. by Dragan Bosnacki and Anton Wijs. Vol. 9641. Springer, 1–18. DOI: 10/gsmrgt.

Rajeev Alur, Kousha Etessami and Mihalis Yannakakis. 2003. 'Inference of Message Sequence Charts'. *IEEE Trans. Software Eng.*, 29, 7, 623–633. DOI: 10/cckm73.

Rajeev Alur, Kousha Etessami and Mihalis Yannakakis. 2000. 'Inference of message sequence charts'. In: *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*. Ed. by Carlo Ghezzi, Mehdi Jazayeri and Alexander L. Wolf. ACM, 304–313. DOI: 10/c3mqm3.

Rajeev Alur, Kousha Etessami and Mihalis Yannakakis. 2001. 'Realizability and Verification of MSC Graphs'. In: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings* (Lecture Notes in Computer Science). Ed. by Fernando Orejas, Paul G. Spirakis and Jan van Leeuwen. Vol. 2076. Springer, 797–808. DOI: 10/ffncmh.

Rajeev Alur and Mihalis Yannakakis. 1999. 'Model Checking of Message Sequence Charts'. In: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings* (Lecture Notes in Computer Science). Ed. by Jos C. M. Baeten and Sjouke Mauw. Vol. 1664. Springer, 114–129. DOI: 10/cxm7xw.

Franco Barbanera, Ivan Lanese and Emilio Tuosto. 2020. 'Choreography Automata'. In: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings* (Lecture Notes in Computer Science). Ed. by Simon Bliudze and Laura Bocchi. Vol. 12134. Springer, 86–106. DOI: 10/gsmrgv.

Franco Barbanera, Ivan Lanese and Emilio Tuosto. 2022. 'Formal Choreographic Languages'. In: *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings* (Lecture Notes in Computer Science). Ed. by Maurice H. ter Beek and Marjan Sirjani. Vol. 13271. Springer, 121–139. DOI: 10/gsmzhf.

Samik Basu and Tevfik Bultan. 28th Mar. 2011. 'Choreography conformance via synchronizability'. In: *Proceedings of the 20th international conference on World wide web*. WWW '11: 20th International World Wide Web Conference. ACM, Hyderabad India, (28th Mar. 2011), 795–804. ISBN: 978-1-4503-0632-4. DOI: 10/bsh5nw.

Samik Basu and Tevfik Bultan. 2016. 'On deciding synchronizability for asynchronously communicating systems'. *Theor. Comput. Sci.*, 656, 60–75. DOI: 10/f9jv2x.

Samik Basu, Tevfik Bultan and Meriem Ouederni. 2012a. 'Deciding choreography realizability'. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. Ed. by John Field and Michael Hicks. ACM, 191–202. DOI: 10/fz3n7m.

Samik Basu, Tevfik Bultan and Meriem Ouederni. 2012b. 'Synchronizability for Verification of Asynchronously Communicating Systems'. In: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by Viktor Kuncak and Andrey Rybalchenko. Vol. 7148. Springer, 56–71. DOI: 10/fzmzwh.

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini and Nobuko Yoshida. Aug. 2008. 'Global Progress in Dynamically Interleaved Multiparty Sessions'. In: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings* (Lecture Notes in Computer Science). Ed. by Franck van Breugel and Marsha Chechik. Vol. 5201. Springer, (Aug. 2008), 418–433. DOI: 10/cxxt7z.

Gregor von Bochmann. 1978. ‘Finite State Description of Communication Protocols’. *Comput. Networks*, 2, 361–372. DOI: 10/brj9tm.

Bernard Boigelot and Patrice Godefroid. 1996. ‘Symbolic Verification of Communication Protocols with Infinite State Spaces Using QDDs (Extended Abstract)’. In: *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings* (Lecture Notes in Computer Science). Ed. by Rajeev Alur and Thomas A. Henzinger. Vol. 1102. Springer, 1–12. DOI: 10/dr434r.

Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes and Amrita Suresh. 2021. ‘A Unifying Framework for Deciding Synchronizability’. In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*, 14:1–14:18. DOI: 10/gsmrg3.

Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern and Martin Leucker. 2010. ‘Learning Communicating Automata from MSCs’. *IEEE Trans. Software Eng.*, 36, 3, 390–408. DOI: 10/fd5m6n.

Ahmed Bouajjani, Constantin Enea, Kailiang Ji and Shaz Qadeer. 2018a. ‘On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony’. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Springer, 372–391. DOI: 10/gsmrg4.

Ahmed Bouajjani, Peter Habermehl and Tomás Vojnar. 2004. ‘Abstract Regular Model Checking’. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings* (Lecture Notes in Computer Science). Ed. by Rajeev Alur and Doron A. Peled. Vol. 3114. Springer, 372–386. DOI: 10/fnfjnp.

Daniel Brand and Pitro Zafiropulo. 1983a. ‘On Communicating Finite-State Machines’. *J. ACM*, 30, 2, 323–342. DOI: 10/dw9xwr.

Gérard Cécé and Alain Finkel. 2005. ‘Verification of programs with half-duplex communication’. *Inf. Comput.*, 202, 2, 166–190. DOI: 10/b8wcwv.

Pierre Chambart and Philippe Schnoebelen. 2008. ‘Mixing Lossy and Perfect Fifo Channels’. In: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings* (Lecture Notes in Computer Science). Ed. by Franck van Breugel and Marsha Chechik. Vol. 5201. Springer, 340–355. DOI: 10/cr7cs8.

Bernadette Charron-Bost, Friedemann Mattern and Gerard Tel. 1996. ‘Synchronous, Asynchronous, and Causally Ordered Communication’. *Distributed Comput.*, 9, 4, 173–191. DOI: 10/dcf58p.

Florent Chevrou, Aurélie Hurault and Philippe Quéinnec. 2016. ‘On the diversity of asynchronous communication’. *Formal Aspects Comput.*, 28, 5, 847–879. DOI: 10/f833k9.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. 2003. ‘Counterexample-guided abstraction refinement for symbolic model checking’. *J. ACM*, 50, 5, 752–794. DOI: 10/cx67fj.

Lorenzo Clemente, Frédéric Herbreteau and Grégoire Sutre. 2014. ‘Decidable Topologies for Communicating Automata with FIFO and Bag Channels’. In: *CONCUR 2014* (LNCS). Vol. 8704, 281–296. DOI: 10/gsmrg5.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani and Nobuko Yoshida. 2015. ‘A Gentle Introduction to Multiparty Asynchronous Session Types’. In: *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures* (Lecture Notes in Computer Science). Ed. by Marco Bernardo and Einar Broch Johnsen. Vol. 9104. Springer, 146–178. DOI: 10/gsmrg6.

René David and Hassane Alla. 1st Feb. 1994. ‘Petri nets for modeling of dynamic systems: A survey’. *Automatica*, 30, 2, (1st Feb. 1994), 175–202. DOI: 10/fh9fkg.

Pierre-Malo Deniélou and Nobuko Yoshida. July 2013. ‘Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types’. In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska and David Peleg. Vol. 7966. Springer, (July 2013), 174–186. DOI: 10/gsmrhb.

Pierre-Malo Deniélou and Nobuko Yoshida. 2012a. ‘Multiparty Session Types Meet Communicating Automata’. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by Helmut Seidl. Vol. 7211. Springer, 194–213. DOI: 10/gsmrg9.

Loïc Desgeorges and Loïc Germerie Guizouarn. 2023. ‘RSC to the ReSCu: Automated Verification of Systems of Communicating Automata’. In: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing*

Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings (Lecture Notes in Computer Science). Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. Springer, 135–143. DOI: 10/gsmrhc.

Cinzia Di Giusto, Loïc Germerie Guizouarn and Étienne Lozes. 2021. 'Towards Generalised Half-Duplex Systems'. In: *Proceedings 14th Interaction and Concurrency Experience, ICE 2021, Online, 18th June 2021* (EPTCS). Ed. by Julien Lange, Anastasia Mavridou, Larisa Safina and Alceste Scalas. Vol. 347, 22–37. DOI: 10/gsmrhd.

Cinzia Di Giusto, Loïc Germerie Guizouarn and Etienne Lozes. 2023. 'Multiparty half-duplex systems and synchronous communications'. *Journal of Logical and Algebraic Methods in Programming*, 131, 100843. DOI: 10/gsmrhf.

Cinzia Di Giusto, Laetitia Laversa and Étienne Lozes. Apr. 2020. 'On the k-synchronizability of Systems'. In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings* (Lecture Notes in Computer Science). Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Springer, (Apr. 2020), 157–176. DOI: 10/gsmrhh.

Volker Diekert and Grzegorz Rozenberg, eds. . 1995. *The Book of Traces*. World Scientific. ISBN: 978-981-02-2058-7. DOI: 10.1142/2563.

Alain Finkel and Étienne Lozes. July 2017. 'Synchronizability of Communicating Finite State Machines is not Decidable'. In: *Proceedings of the 44th International Colloquium on Automata, Languages and Programming (ICALP'17)* (Leibniz International Proceedings in Informatics). Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Anca Muscholl and Fabian Kuhn. Vol. 80. Leibniz-Zentrum für Informatik, Warsaw, Poland, (July 2017), 122:1–122:14. DOI: 10/gsmrhm.

Xiang Fu, Tevfik Bultan and Jianwen Su. Nov. 2004. 'Conversation protocols: a formalism for specification and verification of reactive electronic services'. *Theoretical Computer Science*, 328, 1, (Nov. 2004), 19–37. DOI: 10/dwd7jp.

Xiang Fu, Tevfik Bultan and Jianwen Su. 2005. 'Synchronizability of Conversations among Web Services'. *IEEE Trans. Software Eng.*, 31, 12, 1042–1055. DOI: 10/fn7hqt.

Tristan Le Gall, Bertrand Jeannet and Thierry Jéron. 2006. 'Verification of Communication Protocols Using Abstract Interpretation of FIFO Queues'. In: *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by Michael Johnson and Varmo Vene. Vol. 4019. Springer, 204–219. DOI: 10/db833s.

Hubert Garavel and Damien Thivolle. 2009. 'Verification of GALS Systems by Combining Synchronous Languages and Process Calculi'. In: *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings* (Lecture Notes in Computer Science). Ed. by Corina S. Pasareanu. Vol. 5578. Springer, 241–260. DOI: 10/fwps7d.

Blaise Genest, Dietrich Kuske and Anca Muscholl. 2007. 'On Communicating Automata with Bounded Channels'. *Fundam. Inform.*, 80, 1, 147–167. http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09.

Blaise Genest, Anca Muscholl and Doron A. Peled. 2003. 'Message Sequence Charts'. In: *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]* (Lecture Notes in Computer Science). Ed. by Jörg Desel, Wolfgang Reisig and Grzegorz Rozenberg. Vol. 3098. Springer, 537–558. DOI: 10/c6msfx.

Blaise Genest, Anca Muscholl, Helmut Seidl and Marc Zeitoun. 2006. 'Infinite-state high-level MSCs: Model-checking and realizability'. *J. Comput. Syst. Sci.*, 72, 4, 617–647. DOI: 10/d2j38k.

Alexander Heußner, Tristan Le Gall and Grégoire Sutre. 2009. 'Extrapolation-Based Path Invariants for Abstraction Refinement of Fifo Systems'. In: *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings* (Lecture Notes in Computer Science). Ed. by Corina S. Pasareanu. Vol. 5578. Springer, 107–124. DOI: 10/fhqrs6.

Alexander Heußner, Tristan Le Gall and Grégoire Sutre. 2012. 'McScM: A General Framework for the Verification of Communicating Machines'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Springer, 478–484. DOI: 10/gsmrht.

Alexander Heußner, Jérôme Leroux, Anca Muscholl and Grégoire Sutre. 2012. 'Reachability Analysis of Communicating Pushdown Systems'. *Log. Methods Comput. Sci.*, 8, 3. DOI: 10/gsmrhs.

Gerard J. Holzmann. 1997. 'The Model Checker SPIN'. *IEEE Trans. Software Eng.*, 23, 5, 279–295. DOI: 10/d7wqxt.

Kohei Honda. 1993. 'Types for Dyadic Interaction'. In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings* (Lecture Notes in Computer Science). Ed. by Eike Best. Vol. 715. Springer, 509–523. DOI: 10/dhpsrd.

Kohei Honda, Nobuko Yoshida and Marco Carbone. Jan. 2008. 'Multiparty asynchronous session types'. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, (Jan. 2008), 273–284. DOI: 10/b8m6mf.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley. ISBN: 0-201-02988-X.

Keigo Imai, Julien Lange and Rumyana Neykova. 2022. 'Kmclib: Automated Inference and Verification of Session Types from OCaml Programs'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I* (Lecture Notes in Computer Science). Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Springer, 379–386. DOI: 10/gsn2v2.

Keigo Imai, Rumyana Neykova, Nobuko Yoshida and Shoji Yuen. 2020. 'Multiparty Session Programming With Global Protocol Combinators'. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)* (LIPIcs). Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:30. DOI: 10/gsmrhw.

Dietrich Kuske and Anca Muscholl. 2021. 'Communicating automata'. In: *Handbook of Automata Theory*. Ed. by Jean-Éric Pin. European Mathematical Society Publishing House, Zürich, Switzerland, 1147–1188. DOI: 10.4171/Automata-2/9.

Julien Lange, Emilio Tuosto and Nobuko Yoshida. 14th Jan. 2015a. 'From Communicating Machines to Graphical Choreographies'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by Sriram K. Rajamani and David Walker. ACM, (14th Jan. 2015), 221–232. DOI: 10/gsmrh3.

Julien Lange and Nobuko Yoshida. July 2019a. 'Verifying Asynchronous Interactions via Communicating Session Automata'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (Lecture Notes in Computer Science). Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Springer, (July 2019), 97–117. DOI: 10/gsmrh4.

Laetitia Laversa. 14th Dec. 2021. 'La synchronisabilité pour les systèmes distribués'. PhD thesis. Université Côte d'Azur, (14th Dec. 2021). Retrieved 24th Aug. 2023 from https://theses.hal.science/tel-03574701.

Elaine Li, Felix Stutz, Thomas Wies and Damien Zufferey. 2023. 'Complete Multiparty Session Type Projection with Automata'. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III* (Lecture Notes in Computer Science). Ed. by Constantin Enea and Akash Lal. Vol. 13966. Springer, 350–373. DOI: 10.1007/978-3-031-37709-9_17.

Markus Lohrey. 2003. 'Realizability of high-level message sequence charts: closing the gaps'. *Theor. Comput. Sci.*, 309, 1, 529–554. DOI: 10/cspvjw.

Markus Lohrey and Anca Muscholl. 2002. 'Bounded MSC Communication'. In: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings* (Lecture Notes in Computer Science). Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Springer, 295–309. DOI: 10/bwrdzf.

Markus Lohrey and Anca Muscholl. 2004. 'Bounded MSC communication'. *Inf. Comput.*, 189, 2, 160–181. DOI: 10/dhrsrp.

Rupak Majumdar, Madhavan Mukund, Felix Stutz and Damien Zufferey. 2021. 'Generalising Projection in Asynchronous Multiparty Session Types'. In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference* (LIPIcs). Ed. by Serge Haddad and Daniele Varacca. Vol. 203. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:24. DOI: 10/gsmrh7.

Antoni W. Mazurkiewicz. 1986. 'Trace Theory'. In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986* (Lecture Notes in Computer Science). Ed. by Wilfried Brauer, Wolfgang Reisig and Grzegorz Rozenberg. Vol. 255. Springer, 279–324. DOI: 10/b2kf67.

Kenneth L. McMillan. 2006. 'Lazy Abstraction with Interpolants'. In: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Springer, 123–136. DOI: 10/d2vkb3.

Rémi Morin. 2002. ‘Recognizable Sets of Message Sequence Charts’. In: *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings* (Lecture Notes in Computer Science). Ed. by Helmut Alt and Afonso Ferreira. Vol. 2285. Springer, 523–534. DOI: 10/db8xnd.

Anca Muscholl and Doron A. Peled. 1999. ‘Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces’. In: *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS’99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings* (Lecture Notes in Computer Science). Ed. by Miroslaw Kutylowski, Leszek Pacholski and Tomasz Wierzbicki. Vol. 1672. Springer, 81–91. DOI: 10/d4pp52.

Anca Muscholl, Doron A. Peled and Zhendong Su. 1998. ‘Deciding Properties for Message Sequence Charts’. In: *Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS’98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings* (Lecture Notes in Computer Science). Ed. by Maurice Nivat. Vol. 1378. Springer, 226–242. DOI: 10/d3cg4k.

Christos H. Papadimitriou. 1979. ‘The serializability of concurrent database updates’. *J. ACM*, 26, 4, 631–653. DOI: 10/b2rcdk.

Emil L. Post. 1946. ‘A variant of a recursively unsolvable problem’. *Bull. Amer. Math. Soc. 52*, 264–268. DOI: 10/dtqc88.

Philippe Schnoebelen. 2002. ‘Verifying lossy channel systems has nonprimitive recursive complexity’. *Inf. Process. Lett.*, 83, 5, 251–261. DOI: 10/d36q8k.

Felix Stutz. 2023. ‘Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts’. In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States* (LIPIcs). Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:31. DOI: 10/gsmrjk.

Felix Stutz and Damien Zufferey. 2022. ‘Comparing Channel Restrictions of Communicating State Machines, High-level Message Sequence Charts, and Multiparty Session Types’. In: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022* (EPTCS). Ed. by Pierre Ganty and Dario Della Monica. Vol. 370, 194–212. DOI: 10/gsmrjm.

Kaku Takeuchi, Kohei Honda and Makoto Kubo. 1994. ‘An Interaction-based Language and its Typing System’. In: *PARLE ’94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings* (Lecture Notes in Computer Science). Ed. by Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou and Sergios Theodoridis. Vol. 817. Springer, 398–413. DOI: 10/ch8jf2.

Salvatore La Torre, P. Madhusudan and Gennaro Parlato. 2008. ‘Context-Bounded Analysis of Concurrent Queue Systems’. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings* (Lecture Notes in Computer Science). Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Springer, 299–314. DOI: 10/bvrdjj.

Pitro Zafiropulo, Colin H. West, Harry Rudin, D. D. Cowan and Daniel Brand. 1980. ‘Towards Analyzing and Synthesizing Protocols’. *IEEE Trans. Commun.*, 28, 4, 651–661. DOI: 10/ddwq48.

Web links

Introduction to protocol egineering. (2006). http://cs.uccs.edu/%20cs522/pe/pe.htm.

ITU-TS. Mar. 1993. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. (Mar. 1993). https://www.itu.int/rec/T-REC-Z.120-199303-S/en.

ITU-TS. Oct. 1996. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. (Oct. 1996). https://www.itu.int/rec/T-REC-Z.120-199610-S/en.

ITU-TS. Feb. 2011. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. (Feb. 2011). https://www.itu.int/rec/T-REC-Z.120-201102-I/en.

OMG. 2018. *Business Process Model and Notation*. (2018). https://www.omg.org/spec/BPMN/2.0.

Software

[SW] Loïc Germerie Guizouarn, *ReSCu* 2023. VCS: https://src.koda.cnrs.fr/loic.germerie.guizouarn/rescu.

References of the examples

[1] Julien Lange and Nobuko Yoshida. 2019b. 'Verifying Asynchronous Interactions via Communicating Session Automata'. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (Lecture Notes in Computer Science). Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Springer, 97–117. DOI: 10/gsmrh4.

[2] Julien Lange, Emilio Tuosto and Nobuko Yoshida. 14th Jan. 2015b. 'From Communicating Machines to Graphical Choreographies'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Ed. by Sriram K. Rajamani and David Walker. ACM, (14th Jan. 2015), 221–232. DOI: 10/gsmrh3.

[3] Daniel M. Yellin and Robert E. Strom. 1997. 'Protocol Specifications and Component Adaptors'. *ACM Trans. Program. Lang. Syst.*, 19, 2, 292–333. DOI: 10/fg2k78.

[4] *Introduction to protocol egineering*. (2006). http://cs.uccs.edu/%20cs522/pe/pe.htm.

[5] Sylvain Hallé and Tevfik Bultan. 2010. 'Realizability analysis for message-based interactions using shared-state projections'. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by Gruia-Catalin Roman and André van der Hoek. ACM, 27–36. DOI: 10/cbqd8h.

[6] Gwen Salaün, Lucas Bordeaux and Marco Schaerf. 2006. 'Describing and reasoning on Web Services using Process Algebra'. *Int. J. Bus. Process. Integr. Manag.*, 1, 2, 116–128. DOI: 10/bkp55b.

[7] OMG. 2018. *Business Process Model and Notation*. (2018). https://www.omg.org/spec/BPMN/2.0.

[8] Matthias Güdemann, Gwen Salaün and Meriem Ouederni. 2012. 'Counterexample Guided Synthesis of Monitors for Realizability Enforcement'. In: *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by Supratik Chakraborty and Madhavan Mukund. Vol. 7561. Springer, 238–253. DOI: 10/gsmrhr.

[9] Ahmed Bouajjani, Constantin Enea, Kailiang Ji and Shaz Qadeer. 2018b. 'On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony'. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II* (Lecture Notes in Computer Science). Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Springer, 372–391. DOI: 10/gsmrg4.

[10] Roly Perera, Julien Lange and Simon J. Gay. 2016. 'Multiparty Compatibility for Concurrent Objects'. In: *Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016, Eindhoven, The Netherlands, 8th April 2016* (EPTCS). Ed. by Dominic A. Orchard and Nobuko Yoshida. Vol. 211, 73–82. DOI: 10/gks43x.

[11] Rumyana Neykova, Raymond Hu, Nobuko Yoshida and Fahd Abdeljallal. 2018. 'A session type provider: compile-time API generation of distributed protocols with refinements in F#'. In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. Ed. by Christophe Dubach and Jingling Xue. ACM, 128–138. DOI: 10/gsmrh8.

[12] Raymond Hu. 2017. 'Distributed programming using Java APIs generated from session types'. *Behavioural Types: from Theory to Tools*, 287–308.

[13] Raymond Hu and Nobuko Yoshida. 2016. 'Hybrid Session Verification Through Endpoint API Generation'. In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings* (Lecture Notes in Computer Science). Ed. by Perdita Stevens and Andrzej Wasowski. Vol. 9633. Springer, 401–418. DOI: 10.1007/978-3-662-49665-7_24.

[14] Thierry Jéron and Claude Jard. 1993. 'Testing for unboundedness of fifo channels'. *Theoretical Computer Science*, 113, 1, 93–117. DOI: 10/fq3ssc.

[15] Meriem Ouederni, Gwen Salaün and Tevfik Bultan. 2013. 'Compatibility Checking for Asynchronously Communicating Software'. In: *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers* (Lecture Notes in Computer Science). Ed. by José Luiz Fiadeiro, Zhiming Liu and Jinyun Xue. Vol. 8348. Springer, 310–328. DOI: 10/gsmrjc.

[16] Daniel Brand and Pitro Zafiropulo. 1983b. 'On Communicating Finite-State Machines'. *J. ACM*, 30, 2, 323–342. DOI: 10/dw9xwr.

[17] Stefan Leue, Richard Mayr and Wei Wei. 2004. 'A Scalable Incomplete Test for Message Buffer Overflow in Promela Models'. In: *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings* (Lecture Notes in Computer Science). Ed. by Susanne Graf and Laurent Mounier. Vol. 2989. Springer, 216–233. DOI: 10/dvk5vm.

[18] Stefan Leue, Alin Stefanescu and Wei Wei. 2008. 'Dependency Analysis for Control Flow Cycles in Reactive Communicating Processes'. In: *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings* (Lecture Notes in Computer Science). Ed. by Klaus Havelund, Rupak Majumdar and Jens Palsberg. Vol. 5156. Springer, 176–195. DOI: 10/dkk32b.

[19] Pierre-Malo Deniélou and Nobuko Yoshida. 2012b. 'Multiparty Session Types Meet Communicating Automata'. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings* (Lecture Notes in Computer Science). Ed. by Helmut Seidl. Vol. 7211. Springer, 194–213. DOI: 10/gsmrg9.

[20] Andrea Bracciali, Antonio Brogi and Carlos Canal. 2005. 'A formal approach to component adaptation'. *Journal of Systems and Software*, 74, 1, 45–54. DOI: 10/dr9w9j.

[21] Carlos Canal, Pascal Poizat and Gwen Salaün. 2006. 'Synchronizing Behavioural Mismatch in Software Composition'. In: *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006, Bologna, Italy, June 14-16, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by Roberto Gorrieri and Heike Wehrheim. Vol. 4037. Springer, 63–77. DOI: 10/bn2s99.

[22] Antonio Brogi and Razvan Popescu. 2006. 'Automated Generation of BPEL Adapters'. In: *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings* (Lecture Notes in Computer Science). Ed. by Asit Dan and Winfried Lamersdorf. Vol. 4294. Springer, 27–39. DOI: 10/fgxcsd.

[23] Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel and Pascal Poizat. 2007. 'A Model-Based Approach to the Verification and Adaptation of WF/.NET Components'. In: *Proceedings of the 4th International Workshop on Formal Aspects of Component Software, FACS 2007, Sophia-Antipolis, France, September 19-21, 2007* (Electronic Notes in Theoretical Computer Science). Ed. by Markus Lumpe and Eric Madelaine. Vol. 215. Elsevier, 39–55. DOI: 10/dbkrtp.

[24] Carlos Canal, Pascal Poizat and Gwen Salaün. 2008. 'Model-Based Adaptation of Behavioral Mismatching Components'. *IEEE Trans. Software Eng.*, 34, 4, 546–563. DOI: 10/cq9n3c.

[25] Javier Cámara, José Antonio Martín, Gwen Salaün, Carlos Canal and Ernesto Pimentel. 2010. 'Semi-Automatic Specification of Behavioural Service Adaptation Contracts'. *Electron. Notes Theor. Comput. Sci.*, 264, 1, 19–34. DOI: 10/fksdm7.

[26] Wil MP van der Aalst, Arjan J Mooij, Christian Stahl and Karsten Wolf. 2009. 'Service interaction: Patterns, formalization, and analysis'. *Formal Methods for Web Services: 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Bertinoro, Italy, June 1-6, 2009, Advanced Lectures 9*, 42–88.

[27] Ricardo Seguel, Rik Eshuis and Paul W. P. J. Grefen. 2010. 'Generating Minimal Protocol Adaptors for Loosely Coupled Services'. In: *IEEE International Conference on Web Services, ICWS 2010, Miami, Florida, USA, July 5-10, 2010*. IEEE Computer Society, 417–424. DOI: 10/fjnmg3.

[28] Christian Gierds, Arjan J. Mooij and Karsten Wolf. 2012. 'Reducing Adapter Synthesis to Controller Synthesis'. *IEEE Trans. Serv. Comput.*, 5, 1, 72–85. DOI: 10/fmjmd6.

[29] Amel Bennaceur, Chris Chilton, Malte Isberner and Bengt Jonsson. 2013. 'Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning'. In: *Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings* (Lecture Notes in Computer Science). Ed. by Robert M. Hierons, Mercedes G. Merayo and Mario Bravetti. Vol. 8137. Springer, 274–288. DOI: 10/f2znt8.

[30] Gwen Salaün, Tevfik Bultan and Nima Roohi. 2012. 'Realizability of Choreographies Using Process Algebra Encodings'. *IEEE Trans. Serv. Comput.*, 5, 3, 290–304. DOI: 10/ftstqs.

[31] Tevfik Bultan, Chris Ferguson and Xiang Fu. 2009. 'A Tool for Choreography Analysis Using Collaboration Diagrams'. In: *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009*. IEEE Computer Society, 856–863. DOI: 10/ffhkv6.

[32] Gregor Gößler and Gwen Salaün. 2011. 'Realizability of Choreographies for Services Interacting Asynchronously'. In: *Formal Aspects of Component Software - 8th International Symposium, FACS 2011, Oslo, Norway, September 14-16, 2011, Revised Selected Papers* (Lecture Notes in Computer Science). Ed. by Farhad Arbab and Peter Csaba Ölveczky. Vol. 7253. Springer, 151–167. DOI: 10/gsmrhq.

[33] Gwen Salaün, Lucas Bordeaux and Marco Schaerf. 2004. 'Describing and Reasoning on Web Services using Process Algebra'. In: *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 43. DOI: 10/fv8ncq.

[34] Pascal Poizat and Gwen Salaün. 2007. 'Adaptation of Open Component-Based Systems'. In: *Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings* (Lecture Notes in Computer Science). Ed. by Marcello M. Bonsangue and Einar Broch Johnsen. Vol. 4468. Springer, 141–156. DOI: 10.1007/978-3-540-72952-5_9.

[35] José Antonio Martín and Ernesto Pimentel. 2011. 'Contracts for security adaptation'. *The Journal of Logic and Algebraic Programming*, 80, 3, 154–179. DOI: 10/ctjt9z.

[36] Radu Mateescu, Pascal Poizat and Gwen Salaün. 2008. 'Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques'. In: *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Sydney, Australia, December 1-5, 2008. Proceedings* (Lecture Notes in Computer Science). Ed. by Athman Bouguettaya, Ingolf Krüger and Tiziana Margaria. Vol. 5364, 84–99. DOI: 10/bt6p89.

List of Figures

List of Definition

List of Examples

Automates communicants et communications quasi-synchrones

Loïc GERMERIE GUIZOUARN

Résumé

Les systèmes distribués sont le plus souvent basés sur l'échange asynchrone de messages entre des agents. La programmation par échanges de messages est largement utilisée en calcul haute performance, en programmation événementielle, dans les architectures orientées service, etc. Malheureusement du fait de la variété des modèles de communication, des ambiguïtés dans les spécifications, de la portabilité limitée du code, ou encore de la difficulté à exécuter des tests, il est très difficile de vérifier les systèmes communicants. Le model-checking de systèmes communicants vise à analyser des modèles formels de systèmes distribués et à détecter automatiquement des erreurs comme des pertes de messages ou des inter-blocages. Ces problèmes sont indécidables pour des systèmes à partir de deux machines, et plusieurs hypothèses restrictives ont été étudiées pour rendre les problèmes décidables. Nous définissons dans cette thèse une nouvelle classe de systèmes : les systèmes réalisables avec des communications synchrones (RSC pour faire court). Les comportements de ces systèmes approximent des comportement synchrones, où les messages sont envoyés et reçus simultanément. Nous nous basons sur cette définition pour étudier la généralisation d'une autre classe de systèmes : les systèmes half-duplex. Un système à deux machines est half-duplex si lorsqu'une machine envoie des messages, l'autre ne peut pas lui en envoyer. Nous étudions également un autre formalisme, permettant de raisonner sur les systèmes de manière globale : les chorégraphies. Ce formalisme décrit les exécutions de manière synchrone, et un des problèmes qui y est associé est de vérifier si la combinaison des comportements de chaque acteur qui y est décrit est conforme à la description globale. Nous proposons d'utiliser les propriétés des systèmes RSC pour traiter ce problème.

Mots-clés : Communications, vérification, automates communicants.

Abstract

Most of the distributed systems we use nowadays are based on the message-passing paradigm where systems are structured into parties that interact only by sending and receiving messages asynchronously. Message-passing programming is largely employed in high performance computing (MPI, OpenMP, etc), event-driven applications built on top of actor-based languages (Scala, Erlang, etc), service-oriented architectures, peer-to-peer applications, etc. Unfortunately, because of the variety of communication models (peer to peer, mailbox, etc), of the ambiguities of the specifications of the communication primitives, of a limited portability of the code, and of the difficulty of running representative tests, etc, it is error prone and therefore often reserved to experts. Model-checking of communicating automata aims at analysing formal models of distributed systems and discovering bugs like message loss or deadlocks. Due to the asynchronous nature of the communications, this problem is undecidable in general, even with two machines only, and several restrictions have been considered to restore decidability. We define a new one in this thesis: systems that are realisable with synchronous communications (RSC for short), that is the systems whose behaviours are equivalent to synchronous ones. We propose the class of RSC systems as a generalisation of half-duplex systems, which are system of two machines, where a machines does not send any message if it still has some pending messages to be received in its queue. We study another formalism as well: choreographies, which provide a way to reason globally on a system. Choreographies describe synchronous executions, and one of the problems associated with it is checking whether the combination of all participants of the described communication will behave accordingly to the global description. We propose to rely on the properties of RSC systems to study this problem.

Keywords: Communication, model-checking, communicating automata.