ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering

Master in Computer Science

Realisability of Global Types: Decidability and Verification

Supervisor:
Prof. Ivan Lanese

Co-supervisors:
Prof. Cinzia Di Giusto
Chiar.mo Prof.
Étienne Lozes

Presented by:
Gabriele Genovese

Examiner:
Chiar.mo Prof.
Luca Padovani

Session II October 2025
Accademic Year 2024/2025

Qualcosa

Abstract in Italian

I Tipi Comportamentali definiscono come le informazioni vengono scambiate nei sistemi distribuiti. Un esempio sono i Tipi di Sessione Multiparty (MPST), che descrivono le interazioni tra più partecipanti attraverso protocolli globali e le loro controparti locali. Garantire una corretta implementazione, inclusa l'assenza di deadlock e la conformità alla sessione, è un problema di interesse primario nei MPST. Mentre la maggior parte della ricerca si concentra sulla comunicazione punto-a-punto, i sistemi reali spesso utilizzano modelli di comunicazione differenti, come la messaggistica basata su mailbox o l'ordinamento causale dei messaggi. Una sfida fondamentale è che protocolli validi in un modello di comunicazione possono fallire in un altro. In questo lavoro, sviluppiamo un framework, basato sui MPST, flessibile e parametrizzato da diverse semantiche di comunicazione di rete, tra cui asincrona, punto-a-punto, con ordinamento causale e sincrona. Studiamo il problema dell'realizzabilità da una prospettiva semantica ampia, con l'obiettivo di comprenderne i limiti fondamentali. I miei contributi includono un studio sui lavori correlati, una dimostrazione di indecidibilità per la realizzabilità debole sotto semantica sincrona e miglioramenti al tool ReSCu per la verifica dell'assenza di deadlock nei sistemi sincroni. Questo approccio incorpora i modelli di comunicazione come parametro e fornisce una base per la verifica dei sistemi distribuiti oltre i classici scenari.

Abstract in English

Behavioural Types define how information is exchanged in distributed systems. An example are Multiparty Session Types (MPST), which describe interactions between multiple participants using global protocols and their local counterparts. Ensuring correct implementation, including deadlock freedom and session conformance, is a central concern in MPST. While most research targets peer-to-peer communication, real-world systems often use different communication models such as mailbox-based or causally ordered messaging. A key challenge is that protocols valid in one model may fail in another. In this work, we develop a flexible MPST framework parameterized by different network semantics, including asynchronous, peer-to-peer, causal ordering, and synchronous. We study the realisability problem from a broad semantic perspective, aiming to understand its fundamental limits. My contributions include a survey of related work, a proof of undecidability for weak realisability under synchronous semantics, and enhancements to the ReSCu tool for checking deadlock freedom in synchronous systems. This approach embeds communication models as a parameter, and it provides a basis for verifying distributed systems beyond classical settings.

Contents

List of Figures

Listings

Chapter 1

Introduction

Informally, a *distributed system* is a collection of independent computing entities (called also processes, actors, nodes, or participants; with slightly differences in the meaning) that communicate and coordinate their actions through message passing over a medium of communication (typically an **asynchronous network**), with the goal of solving a common problem. For example, a client-server application can be seen as a form of distributed system, where the shared objective is to provide services to an end user.

Distributed systems make it possible to address challenges that are hard to solve without such an architecture, such as high availability and elastic scalability. However, these benefits come with their own set of challanges that computer scientists need to address, for example, ensuring reliability in the presence of failures in critical systems, and maintaining data consistency. Distributed systems are widely adopted in domains such as *cloud computing*, critical infrastructures, and telecommunication-oriented applications (i.e. autonomous cars, aerospace systems, etc.). Given their ubiquity, it is crucial to study every aspect of their **design**, **execution**, and **verification**. To manage these complexities in a mathematical way, researchers rely on formal abstractions and rigorous methodologies. These allow us to move from ad-hoc engineering practices to systematic approaches with provable guarantees.

One recurring difficulty in distributed systems' development is writing **correct programs**. Avoiding programming and logical errors is inherently hard, even for experienced developers. To mitigate this, many abstractions have been introduced, and computer scientists have focused their efforts on developing *formal frameworks* that provide guarantees about program behavior.

Formal methods for distributed systems offer mathematically rigorous techniques to specify, design, and verify such systems. They are valuable both during design and development, by helping detect errors early, and during analysis, by enabling the study of critical properties such as **safety**, **liveness**, and **deadlock-freedom**. Two primary approaches are *model checking* verification and *correctness by-construction*. Model checking systematically explores a system's state space to confirm properties, while by-construction verification ensures correctness through the design process itself, preventing errors from being introduced.

Among the many aspects of distributed systems, communication is particularly prone to subtle errors and inconsistencies. To reason formally about communication protocols, several models have been proposed, including the Calculus of Communicating Systems (CCS), the π-calculus, and choreographies. In this context, *Multiparty Session Types* (MPST) [20] stand out as a powerful framework. MPST are designed specifically to formalize and verify structured communication among multiple participants, providing strong guarantees about protocol correctness.

MPST describe communication through a *global type*, which specifies the entire interaction among all participants. This global type is then *projected* into *local types*, one for each participant. Local types act as contracts, ensuring that each component adheres to the protocol. As a result, MPST allow developers to guarantee properties such as deadlock-freedom and protocol compliance at compile time, making them an especially appealing tool for designing robust communication protocols.

1.1 Goal

The goal of this work is to investigate the **realisability problem** for MPST, which asks whether a global specification can be faithfully realised by a collection of *local processes* in a distributed system. This question naturally arises in top-down development methodologies, such as MPST or choreographic frameworks [28], where the design begins from a *global perspective* and the local behaviour of each participant is derived afterwards.

The realisability problem is central to ensuring that the distributed implementation does not diverge from the intended specification. In essence, the challenge is to determine whether the set of projected local processes can really **respect** the behaviour prescribed by the global model, while preserving essential properties such as correctness, progress, and deadlock-freedom.

A releted-work analysis is provided in Chapter 5, where we examine how similar problems have been addressed in other formal frameworks. To illustrate the relevance of this problem, consider the following example.

Example 1.1.1. Consider four processes A, B, C, and D communicating over an asynchronous network, with four messages x, y, z, and w to be exchanged as specified in Listing 1.1. A natural question arises: can such a specification be faithfully implemented in a real distributed system?

```
A sends B either message x or y.

If A sends B message x,
    then C sends D message z.

If A sends B message y,
then C sends D message w.
```

Listing 1.1: Example specification of message exchanges

While the specification can be expressed using several of the formalisms mentioned earlier, some of them are capable of revealing that it is, in fact, *impossible* to implement in a real distributed system. The reason is that process C cannot determine which message to send to D without knowing which message A sent to B, because this information is not locally available to C.

The realisability problem in this work is examined from a theoretical perspective to provide a more formal and precise understanding of the fundamental limits that exist and why syntactical constraints of certain models work.

Unlike the standard approach to *Global Types* in MPST, which often relies on a purely syntactic representation, in this work we adopt a more *semantic approach*. Specifically, we represent global types as *automata*. This automata-based representation is highly modular, incorporating various *network semantics* (such as asynchronous, peer-to-peer, causal ordering, and synchronous semantics) as explicit parameters of the framework. Such parameterization allows a flexible analysis of different communication models within a unified setting. In this framework, we interpret the semantics of a global type as a set of Message Sequence Charts (MSCs). It is therefore useful to recall related questions that have been studied in the context of MSCs [1, 2]. These formalisms provide both historical context and technical insights, and several known results from this line of work will be directly leveraged in the present study.

Message Sequence Charts (MSCs) are a standardised graphical formalism, introduced in 1992 [21], used to describe trace languages for specifying communication behaviour. Thanks to their simplicity and intuitive semantics, MSCs have been widely adopted in industry. Figure 1.1 illustrates a simple example based on a minimal client–server architecture. To give more context, an extension of this formalism, known as High-Level Message Sequence Charts (HMSCs), was later introduced [22].

HMSCs enable the definition of MSCs as nodes connected by transitions and are used to model more complex patterns of message flows by capturing sequences, alternatives, or iterations of atomic MSC scenarios.

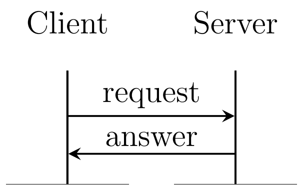

Figure 1.1: Simple example of a client-server architecture.

The *weak realisability problem* for MSCs asks whether there exists a distributed implementation that can realise all behaviours of a finite set of MSCs without introducing additional ones. A stronger variant, called *safe realisability*, requires the implementation to also be **deadlock-free**.

Remark. The term "realisability" has several synonyms in the literature. In other works, it is often referred to as *implementability* or *projectability.* Each of these terms, depending on the formal model considered, comes with slightly different definitions. Some of these variations will be analysed in Chapter 5.

With MSCs, the work of Di Giusto et al. [12] introduces interesting communication semantics and a hierarchy among them. The main goal of their study was to establish a hierarchy that preserves *monotonic* properties: if a property holds for a given communication semantics, it should also hold for all semantics contained within it. However, they showed that this monotonicity only applies to certain properties. In this work, we continue the study within the same framework, focusing on the realisability property.

In the following paragraphs, we describe some of these communication semantics informally, using examples to highlight the differences between them. In particular, we will later formally define `synch` in Definition 2.2.3, as this communication semantics is used in the main contribution of this thesis. Chapter 5 continues the discussion by presenting additional communication semantics and summarizing the relevance of the work by Di Giusto et al. [12]. Some examples of different communication semantics are illustrated in Figures 1.2 and 1.4, whose *membership* in these classes can be verified using an online MSC tool [15].

Fully asynchronous. In the fully asynchronous communication model (`asy`), messages can be received at any time after they have been sent, and send events are non-blocking. This model can be viewed as an unordered "bag" in which all messages are stored and retrieved by processes when needed. It is also referred to as *non-FIFO*. The formal definition coincides with that of an MSC (Definition 2.2.5). Figure 1.2 illustrates an example of asynchronous communication.

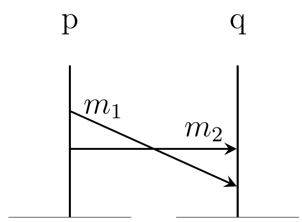

Figure 1.2: Asynchronous semantic example.

Peer-to-peer. In the peer-to-peer (`p2p`) communication model, any two messages sent from one process to another are always received in the same order as they are sent. An alternative name is FIFO. An example is shown in Figure 1.3.

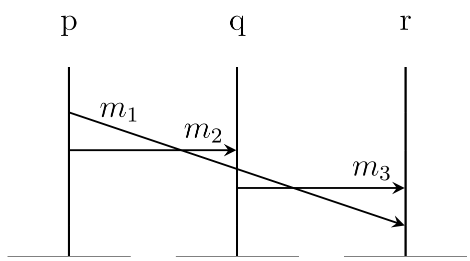

Figure 1.3: Peer-to-peer semantic example.

Synchronous. The synchronous (`synch`) communication model imposes the existence of a scheduling such that any send event is immediately followed by its corresponding receive event. An example for this communication model is shown in Figure 1.4.c. A formal definition is given later for this semantic (Definition 2.2.3).

The definition of these models will become central in the reduction techniques explored in this work: simplifying the study of realisability by reducing richer semantics to the synchronous case.

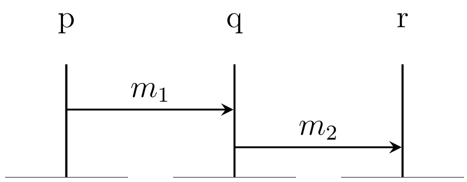

Figure 1.4: Synchronous semantic example.

1.2 Reduction to synchronous semantic

Theorem [14, Theorem 5.3] suggests that reasoning about realisability could become more tractable under *synchronous* semantics for automata-based solutions. In synchronous communication, send and receive actions are tightly coupled, effectively eliminating the nondeterminism introduced by asynchronous message buffering. The theorem is formally stated in Chapter 4.

Formally, the theorem shows that if a global type is realisable under synchronous semantics, then, under certain conditions, it is also realisable in more general models, such as peer-to-peer semantics. This reduction requires constraints such as *orphan-freedom* (no message sent is left unmatched) and *deadlock-freedom*.

We present the theorem here informally, using the standard meanings of terms that have already been introduced: a global type G is deadlock-free realisable in `p2p` if and only if the following four conditions hold

- the language generated by G's local type has synchronous semantics;
- all G's projections are orphan-free;
- all the traces of the MSCs' language of G are deadlock-free in `p2p`;
- G is realisable in synchronous semantics.

The second and third conditions are already known to be decidable and can be automatically verified. The focus of this thesis is instead on the fourth condition, namely checking whether a global type is realisable in synchronous semantics. The undecidability result presented in Chapter 3 shows that this condition cannot be verified in general. Consequently, the theorem above must be refined by introducing further restrictions that ensure decidability.

This observation motivates the second part of the thesis: Chapter 4 presents the extension of the ReSCu tool, which provides practical verification of properties such as *deadlock-freedom* and *progress*. These results should be understood as building blocks toward identifying restricted subclasses of synchronous systems that admit decidable realisability checks, complementing the undecidability findings of the theoretical contribution.

Given the context, the developments presented in this thesis can be grouped into two main contributions, one theoretical and one practical, both closely connected.

1.3 Contributions

The main contributions of this work are:

- a proof of the **undecidability** of the *weak realisability* problem under the synchronous semantics of our framework;
- an extension and improvement of the model-checking tool ReSCu [16], enabling the verification of *deadlock-freedom* and *progress* for synchronous systems.

These two contributions are closely connected: they both address the realisability problem, but from two complementary angles. The first contribution establishes undecidability, showing that in the general case the weak realisability problem cannot be solved for synchronous semantic. This motivates the second contribution: once undecidability is proven, there is a clear need to identify suitable restrictions of the problem that yield decidability results. The model-checking framework presented in the second part of the thesis could be a foundational step towards this direction, providing practical verification techniques that can serve as building blocks for further decidability analyses.

The thesis is structured as follows. Chapters 2 and 3 then introduce the formal definitions and present the main theoretical contribution. Chapter 4 develops the practical contributions through the ReSCu tool. Chapter 5 presents a detailed overview of related work, comparing different approaches in the literature and highlighting how this thesis departs from them. Finally, Chapter 6 concludes with a discussion of the results and outlines directions for future research and development.

Chapter 2

Preliminaries

In this section, the fundamental concepts and definitions necessary to contextualize the main contributions of this work are presented. We, first, introduce automaton, executions, and Message Sequence Charts (MSC), followed by an examination of communication model's semantics that are particularly interesting. Then, the notions of Global Type and Realisability are defined within the scope of this work, along with the foundational elements required to understand the theoretical contributions.

2.1 Standard notions on automata

Words and alphabet. Σ is used to denote a set of finite words. Arr denotes a set of finite action used for a global type. For a string s, let s^l denote the l-th character of the string.

Definition 2.1.1 (NFA)**.** A non-deterministic finite automaton (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

We write $\delta^*(s, w)$ to denote the set of states s' reachable from s along a path labelled with w. The language accepted by $\mathcal{A}$, denoted $\mathcal{L}_{\text{words}}(\mathcal{A})$, is the set of words $w \in \Sigma^*$ such that $\delta^*(q_0, w) \cap F \neq \emptyset$.

Definition 2.1.2 (DFA)**.** A deterministic finite automaton (DFA) is an NFA where the transition relation δ is a partial function $\delta : Q \times \Sigma \to Q$. The DFA is complete if δ is total.

Definition 2.1.3 (Determinization)**.** To every NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, we associate the DFA $\det(\mathcal{A}) = (Q', \Sigma, \delta', q_0', F')$, where $Q' = 2^Q$, $q_0' = \{q_0\}$, F' is the set of subsets of Q that contain at least one accepting state, and δ' is defined by $\delta'(S, a) = \bigcup\{\delta^*(s, a) \mid s \in S\}$ for all $S \in Q'$, $a \in \Sigma$.

We write $\widehat{\mathcal{A}}$ for the automaton obtained from $\mathcal{A}$ by setting $F = Q$.

2.2 Execution, Communication Models and MSC

We assume a finite set of *processes* $\mathbb{P} = \{p, q, \ldots, P1, P2, \ldots\}$ and a finite set of messages (labels) $\mathbb{M} = \{m_1, m_2, \ldots\}$. We consider two kinds of actions:

- *send actions*, of the form $!m^{p\to q}$, executed by process p when sending message m to q;
- *receive actions*, of the form $?m^{p\to q}$, executed by process q when receiving m from p.

Furthermore, we write Act for the set $\mathbb{P} \times \mathbb{P} \times \{!, ?\} \times \mathbb{M}$ of all actions, and Act_p for the subset of actions executable by p (i.e., $!m^{p\to q}$ or $?m^{q\to p}$). When processes are clear from the context, we abbreviate send and receive actions as $!m$ and $?m$, respectively.

An *event* η of a sequence of actions $w \in \mathsf{Act}^*$ is an index $i \in \{1, \ldots, \mathsf{length}(w)\}$. It is a *send event* (resp. *receive event*) if $w[i]$ is a send (resp. receive) action. We denote by $\mathsf{events}_S(w)$ (resp. $\mathsf{events}_R(w)$) the set of send (resp. receive) events of w, and $\mathsf{events}(w) = \mathsf{events}_S(w) \cup \mathsf{events}_R(w)$. When all events are labelled with distinct actions, we identify an event with its action.

Executions.

An execution is a well-defined sequence of actions $e \in \mathsf{Act}^*$, where a receive action is always preceded by a unique corresponding send action.

Definition 2.2.1 (Execution)**.** An *execution* over $\mathbb{P}$ and $\mathbb{M}$ is a sequence of actions $e \in \mathsf{Act}^*$ together with an injective mapping $\mathsf{src}_e : \mathsf{events}_R(e) \to \mathsf{events}_S(e)$ such that for each receive event i labelled $?m^{p\to q}$, its source $\mathsf{src}_e(i)$ is labelled $!m^{p\to q}$ and $\mathsf{src}_e(i) < i$.

For a set of executions $\mathcal{E}$, let $\mathsf{Prefixes}(\mathcal{E})$ be the set of all prefixes of executions in $\mathcal{E}$. The *projection* $\mathsf{proj}_p(e)$ of e on process p is the subsequence of actions in Act_p. A send event s is *matched* if there exists a receive event r such that $\mathsf{src}(r) = s$. An execution is *orphan-free* if all send events are matched, i.e., if src is surjective onto $\mathsf{events}_S(e)$.

Communication Models.

In this thesis, we focus on a communication model: the synchronous model (synch). Nonetheless, this work forms part of a broader and more general project. Some results presented here naturally extend to a wide range of communication models, often requiring only mild additional assumptions. Please, refer to the related work chapter (Chapter 5, Section 5.1). From this perspective, we introduce a general definition of a communication model.

Definition 2.2.2 (Communication model)**.** A *communication model* com is a set $\mathcal{E}_{\mathsf{com}}$ of executions.

In the *synchronous model* synch, every send is immediately followed by its matching receive:

Definition 2.2.3 (synch)**.** An execution $e = (w, \mathsf{src})$ belongs to $\mathcal{E}_{\mathsf{synch}}$ if for every send event $s \in \mathsf{events}_S(e)$, the event $s+1$ is a receive event with $\mathsf{src}(s+1) = s$.

Furthermore, the *source function* src_e is defined as follows.

Definition 2.2.4 (src function for synch)**.** If e is an execution in synch, then for every receive event i we define $\mathsf{src}_e(i) = i - 1$.

Message Sequence Charts.

While executions correspond to a total order of events in a system, message sequence charts (MSCs) provide a distributed view, using a partial order on events. For a tuple $M = (w_p)_{p \in \mathbb{P}}$, each $w_p \in \mathsf{Act}_p^*$ is a sequence of actions executed by process p, according to some total, locally observable order. We write $\mathsf{events}(M)$ for the set $\{(p,i) \mid p \in \mathbb{P} \text{ and } 0 \leq i < \mathsf{length}(w_p)\}$. The label $\mathsf{action}(\eta)$ of an event $\eta = (p,i)$ is the action $w_p[i]$. The event η is a send (resp. receive) event if it is labelled with a send (resp. receive) action. We write $\mathsf{events}_S(M)$ (resp. $\mathsf{events}_R(M)$) for the set of send (resp. receive) events of M. We also write $\mathsf{msg}(\eta)$ for the message sent or received at η, and $\mathsf{proc}(\eta)$ for the process executing η. Finally, we write $\eta_1 \prec_{\mathsf{proc}} \eta_2$ if there exists a process p and indices $i < j$ such that $\eta_1 = (p,i)$ and $\eta_2 = (p,j)$.

Definition 2.2.5 (Message Sequence Chart)**.** An *MSC* over $\mathbb{P}$ and $\mathbb{M}$ is a tuple $M = \big((w_p)_{p \in \mathbb{P}}, \mathsf{src}\big)$ where

1. for each process p, $w_p \in \mathsf{Act}_p^*$ is a finite sequence of actions;
2. $\mathsf{src} : \mathsf{events}_R(M) \to \mathsf{events}_S(M)$ is an injective function from receive events to send events such that for all receive event η labelled with $?m^{p \to q}$, $\mathsf{src}(\eta)$ is labelled with $!m^{p \to q}$.

For an execution e, $\mathsf{msc}(e)$ is the MSC $\big((w_p)_{p\in\mathbb{P}}, \mathsf{src}\big)$ where w_p is the subsequence of e restricted to the actions of p, and src is the lifting of src_e to the events of $(w_p)_{p\in\mathbb{P}}$.

Example 2.2.1. Consider the MSC depicted in Figure 2.1. It consists of $\mathbb{P} = \{p, q, r\}$ and $\mathbb{M} = \{m_1, m_2, m_3\}$ with $M = ((w_p, w_q, w_r), \mathsf{src})$, where $w_p = !m_1?m_2$, $w_q = ?m_1!m_2!m_3$, $w_r = ?m_3$, $\mathsf{src}((p, 2)) = (q, 2)$, $\mathsf{src}((q, 1)) = (p, 1)$, and $\mathsf{src}((r, 1)) = (q, 3)$.

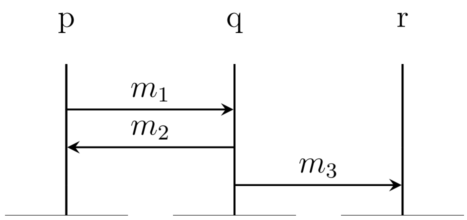

Figure 2.1: Simple example with an exchange of three messages.

Given a set of processes $\mathbb{P}$, an MSC $M = \big((w_p)_{p\in\mathbb{P}}, \mathsf{src}\big)$ is said to be a *prefix* of another MSC $M' = \big((w'_p)_{p\in\mathbb{P}}, \mathsf{src}'\big)$, denoted by $M \leq_{\mathsf{pref}} M'$, if the following conditions hold:

- for every $p \in \mathbb{P}$, the sequence w_p is a prefix of w'_p;
- for every receive event e of M, it holds that $\mathsf{src}'(e) = \mathsf{src}(e)$.

The *concatenation* of two MSCs M_1 and M_2 is the MSC $M_1 \cdot M_2$ obtained by stacking M_1 vertically above M_2. Formally, let $M_1 = ((w^1_p)_{p\in\mathbb{P}}, \mathsf{src}_1)$ and $M_2 = ((w^2_p)_{p\in\mathbb{P}}, \mathsf{src}_2)$. Then: (i) for each process p, the sequence is $w_p = w^1_p \cdot w^2_p$; (ii) the source function src is defined so that $\mathsf{src}(e) = \mathsf{src}_i(e)$ for all receive events e belonging to M_i, with $i \in \{1, 2\}$.

Happens-before relation and linearisations

In a given MSC M, an event η happens before η', if η and η' are events of a same process p and happen in that order on the timeline of p; η is send event matched by η'; and a sequence of such situations defines a path from η to η'.

Definition 2.2.6 (Happens-before relation)**.** Let M be an MSC. The happens-before relation over M is the binary relation $\prec_M$ defined as the least transitive relation over $\mathsf{events}(M)$ such that:

- for all p, i, j, if $i < j$, then $(p, i) \prec_M (p, j)$, and

- for all receive events η, $\mathsf{src}(\eta) \prec_M \eta$.

Example 2.2.2. Consider the Example 2.2.1. The following happens-before relations are valid:

$$!m_1 \prec_M ?m_1 \prec_M !m_2 \prec_M !m_3 \prec_M ?m_3$$

and

$$!m_1 \prec_M ?m_1 \prec_M !m_2 \prec_M ?m_2.$$

Definition 2.2.7 (Linearisation)**.** A *linearisation* of an MSC M is a total order $\ll$ on $\mathsf{events}(M)$ that refines $\prec_M$: for all events η, η', if $\eta \prec_M \eta'$, then $\eta \ll \eta'$.

We write $\mathsf{lin}(M)$ for the set of all linearisations of M. We often identify a linearisation with the execution it induces.

Example 2.2.3. Considering the Example 2.2.1, let M be the MSC in Figure 2.1. The elements of the set $\mathsf{lin}(M)$ are

$$!m_1?m_1!m_2?m_2!m_3?m_3,$$

$$!m_1?m_1!m_2!m_3?m_2?m_3,$$

$$!m_1?m_1!m_2!m_3?m_3?m_2.$$

Given an MSC M, we write $\mathsf{lin}_{\mathsf{com}}(M)$ to denote $\mathsf{lin}(M) \cap \mathcal{E}_{\mathsf{com}}$; the executions of $\mathsf{lin}_{\mathsf{com}}(M)$ are called the linearisations of M in the communication model $\mathtt{com}$.

Definition 2.2.8 (com-linearisable MSC)**.** An MSC M is *linearisable* in a communication model com if $\mathsf{lin}_{\mathsf{com}}(M) \neq \emptyset$. We write $\mathcal{M}_{\mathsf{com}}$ for the set of all MSCs linearisable in com.

Example 2.2.4. Consider the Example 2.2.1 and the respective linearisation listed in Example 2.2.3. The MSC M is *linearisable* in the synch communication model because $\mathsf{lin}_{\mathsf{synch}}(M) \neq \emptyset$. The only element of $\mathsf{lin}_{\mathsf{synch}}(M)$ is

$$!m_1?m_1!m_2?m_2!m_3?m_3.$$

All the send events are followed by the respective receive events.

Communicating finite state machines.

We recall the definition of communicating finite state machines [7].

Definition 2.2.9 (CFSM)**.** A communicating finite state machine (CFSM) is an NFA with ε-transitions $\mathcal{A}$ over the alphabet Act. A system of CFSMs is a tuple $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$.

Given a system of CFSMs $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$, we write $\widehat{\mathcal{S}}$ for the system of CFSMs $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}_p})_{p\in\mathbb{P}}$ where all states are accepting, i.e., $F_p = Q_p$.

Definition 2.2.10 (Executions of CFSMs in com)**.** Given a system $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ and a model com, $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$ is the set of executions $e \in \mathcal{E}_{\mathsf{com}}$ such that $\mathsf{proj}_p(e) \in \mathcal{L}_{\mathsf{words}}(\mathcal{A}_p)$ for all p.

We write $\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\mathcal{S})$ for the set $\{\mathsf{msc}(e) \mid e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})\}$.

A system is orphan-free if, whenever all machines have reached an accepting state, no sent message remains in transit, i.e. no message is sent but not received.

Definition 2.2.11 (Orphan-free)**.** A system $\mathcal{S}$ is *orphan-free* in a model com if all its executions in $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$ are orphan-free.

Remark. All synchronous executions are orphan-free by definition.

A system is said to be *deadlock-free* if every *partial* execution (i.e., an execution that may have stopped before reaching a final state) can always be extended to a complete or accepting execution. Intuitively, this means that the system can never reach a state in which all processes are waiting indefinitely, and no further progress is possible.

Definition 2.2.12 (Deadlock-freedom)**.** A system $\mathcal{S}$ is *deadlock-free* in com if, for every execution $e \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\widehat{\mathcal{S}})$, there exists a completion e' such that $e \leq_{\mathsf{pref}} e'$ and $e' \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$.

Remark. This definition captures the idea that, in a deadlock-free system, no reachable configuration represents a permanent waiting state, every intermediate behaviour can eventually lead to a valid full execution. In other words, the system's communication and synchronisation structure guarantees continuous potential for progress. For a detailed discussion of this property and its verification within our framework, refer to Chapter 4.

2.3 Global Types

This section introduces the fundamental notions required to understand the formal proof presented in Chapter 3, focusing in particular on the concepts of *Global Types* and *Weak Realisability.*

Definition 2.3.1 (Global Type)**.** An *arrow* is a triple $(p, q, m) \in \mathbb{P} \times \mathbb{P} \times \mathbb{M}$ with $p \neq q$; we often write $p \xrightarrow{m} q$ instead of (p, q, m), and use Arr to denote the finite set of arrows. A *Global Type* G is a *deterministic finite automaton* (DFA) over the alphabet Arr.

We use the notation $p \xleftrightarrow{m} q$ to denote the *round-trip* exchange of a message m: first p sends m to q, and then q sends back the same message m to p. This serves as an acknowledgment message for p, ensuring that both directions of communication are explicitly represented in the model.

Remark. In our setting, Global Types are viewed as automata that generate a language of MSCs (Definition 2.3.1 and 2.3.3), following the line of work by Di Giusto et al. [14]. It is important to note that this notion of Global Types differs significantly from the one commonly adopted in the Multiparty Session Types (MPST) literature, where Global Types are usually defined as structured syntactic descriptions of communication protocols. Here, the automata-based interpretation aligns with other lines of work that study of the *realisability problem*, providing a formal foundation for reasoning about synchrony, causality, and implementability. For a comparison and discussion of similarities and differences with other formalisms, refer to Chapter 5.

Example 2.3.1. An example of a Global Type expressed as an automaton is the following. Consider the not-realisable specification stated in Listing 1.1. The specification automaton depicted in Figure 2.2 is a DFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where:

- $\mathbb{P} = \{A, B, C, D\}$;
- $Q = \{1, 2, 3, 4, 5\}$ is the set of states;
- $\Sigma = \{A \xrightarrow{x} B, A \xrightarrow{y} B, C \xrightarrow{z} D, C \xrightarrow{w} D\}$ is the alphabet Arr;
- The transition function $\delta : Q \times \Sigma \to Q$ is defined as:

  $$\delta(1, A \xrightarrow{x} B) = 2, \quad \delta(1, A \xrightarrow{y} B) = 3, \quad \delta(2, C \xrightarrow{z} D) = 4, \quad \delta(3, C \xrightarrow{w} D) = 5;$$

- The initial state is $q_0 = 1$;
- The set of accepting states is $F = \{4, 5\}$.

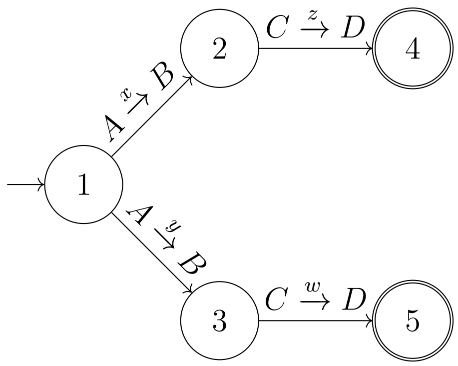

Figure 2.2: Listing 1.1's global type.

The projection of a Global Type G on a process p is the CFSM G_p obtained by replacing each arrow $q \xrightarrow{m} r$ in the transitions of G with the corresponding local action of p: $!m^{p \to r}$ if $p = q$, $?m^{q \to p}$ if $p = r$, and ε otherwise.

Definition 2.3.2 (Projected system of CFSMs)**.** The *projected system of CFSMs* $\mathsf{proj}(\mathsf{G})$ associated with a Global Type G is the tuple $(\mathsf{G}_p)_{p \in \mathbb{P}}$, where each G_p is the local behaviour of process p obtained via projection from G.

We can now formally define the relationship between MSCs and Global Types. Intuitively, Global Types represent a set of MSCs, allowing us to reason about multiple message sequence scenarios.

A Global Type defines a language of MSCs in two different ways, one existential and one universal. Let $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$ be the set of sequences of arrows w accepted by G. Informally, the existential MSC language $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$ of a Global Type G is the set of MSCs that admit at least one representation as a sequence of arrows in $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$, and the universal MSC language $\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathsf{G})$ of a Global Type G is the set of MSCs whose representations as sequences of arrows are all in $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$. We will just give the formal definition of $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$:

Definition 2.3.3 ($\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$)**.**

$$\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G}) \stackrel{\mathsf{def}}{=} \{\mathsf{msc}(w) \mid w \in \mathcal{L}_{\mathsf{words}}(\mathsf{G})\}$$

When a Global Type is implemented in a concrete system, its semantics depends on the chosen communication model.

Definition 2.3.4 (Global Type Language)**.** Let G be a global type and com a communication model. The language of G in com is $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathsf{G}) \stackrel{\mathsf{def}}{=} \bigcup\{\mathsf{lin}_{\mathsf{com}}(M) \mid M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})\}$.

We now give the definitions of *weak* and *safe* realisability.

Definition 2.3.5 (Weak realisability)**.** A global type G is *weak realisable* in the communication model com if there is a system CFSM $\mathcal{S}$ such that the following condition hold: $\mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S}) = \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathsf{G})$.

Although this work does not focus on safe realisability, we will still define it formally to highlight the main differences and similarities with other works in Chapter 5.

Definition 2.3.6 (Safe realisability)**.** A global type G is *safe realisable* in the communication model com if there is a system $\mathcal{S}$ that is *weak realisable* and $\mathcal{S}$ is deadlock-free in com (Def. 2.2.12).

The definition of Weak realisability corresponds to the property of *global type conformance*: all system executions faithfully follow the behaviours prescribed by the global type. When com is $\mathsf{p2p}$ or synch, our notion of safe realisability coincides with the notion of *safe realisability* introduced in [3]. This equivalence does not extend to more general communication models, such as the mailbox model [14]. We are now ready to present the main contributions of this work.

Chapter 3

Weak-Realisability is Undecidable for Synch Global Types

The first contribution is Theorem 1, which establishes that *Weak-realisability is undecidable for synchronous global types*. To prepare for this result, we have introduced in Chapter 2 the basic notions of MSCs, Global Types, and Weak-realisability. We now present the main objects used in the proof of Theorem 1, which we adapt from Alur et al. [3], and we highlight along the way the key differences with the original construction.

3.1 Definitions

The proof is a *reduction* from the **Relaxed Post Correspondence Problem (RPCP)**, a variant of the classical Post Correspondence Problem (PCP). RPCP was shown to be undecidable by Alur et al. [3], via reduction from PCP. The main idea is to encode the existence of a solution to an RPCP instance into the non-realisability of our formal specification. In the original proof, MSCs are directly used to build an HMSC called M^*. In our case, we will define a *global type* (called L^*) built from global types. A generic solution for the RPCP problem will correspond to the global type L^*. Therefore, we need to prove:

$$\Delta \in \text{RPCP} \quad \iff \quad L^* \text{ is not realisable.}$$

Definition 3.1.1 (Relaxed Post Correspondence Problem)**.** Given a set of tiles $\{(v_1, w_1), (v_2, w_2), ..., (v_r, w_r)\}$, determining whether there exist indices $i_1, ..., i_m$ such that

$$x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m},$$

where $x_{i_j}, y_{i_j} \in \{v_{i_j}, w_{i_j}\}$, such that:

- there exists at least one index i_ℓ for which $x_{i_\ell} \neq y_{i_\ell}$, and
- for all $j \leq m$, $y_{i_1} \cdots y_{i_j}$ is a strict or not-strict prefix of $x_{i_1} \cdots x_{i_j}$.

Intuitively, RPCP requires that the concatenation on the left-hand side always grows at least as fast as the right-hand side, while ensuring that at least one chosen tile differs between the two sequences. Moreover, in constructing the strings, we may freely choose which element of each tile (either v_i or w_i) contributes to the left or right sequence.

Example 3.1.1 (Simple RPCP instance)**.** Consider the tile set

$$(v_1, w_1) = (\mathtt{b},\ \mathtt{bb}), \quad (v_2, w_2) = (\mathtt{a},\ \mathtt{ab}), \quad (v_3, w_3) = (\mathtt{c},\ \mathtt{c}).$$

Take the index sequence $(2, 1, 3)$ and the choices

$$x_1 = w_2,\ y_1 = v_2; \quad x_2 = v_1,\ y_2 = w_1; \quad x_3 = v_3,\ y_3 = w_3.$$

Then

$$x_1x_2x_3 = \mathtt{ab\ b\ c} = \mathtt{abbc}, \qquad y_1y_2y_3 = \mathtt{a\ bb\ c} = \mathtt{abbc},$$

so the two sides are equal.

We now check the RPCP conditions:

- **at least one mismatch:** here $x_1 \neq y_1$ and $x_2 \neq y_2$, so the "some index differs" condition holds;
- **prefix property:** for every prefix length j we have $y_1 \cdots y_j$ is a prefix of $x_1 \cdots x_j$:
  - $j = 1$: $y_1 = \mathtt{a}$ is a prefix of $x_1 = \mathtt{ab}$;
  - $j = 2$: $y_1y_2 = \mathtt{abb}$ is a prefix of $x_1x_2 = \mathtt{abb}$;
  - $j = 3$: $y_1y_2y_3 = \mathtt{abbc}$ is a prefix of $x_1x_2x_3 = \mathtt{abbc}$.

We have now identified the main problem to which our proof reduces. The next step is to encode an RPCP instance into the formal model.

Definition 3.1.2 (M_i^n)**.** Given the index i of a tile (v_i, w_i), and given an interger $n \in \{0, 1\}$, where:

- if $n = 0$, then $x_i = v_i$;
- if $n = 1$, then $x_i = w_i$;

The behavior of the MSC M_i^n is as follows: first, Process 1 synchronously sends message $m_1 = (i, n)$ to Process 2, then Process 1 transmits the index $m_2 = i$ to Process 4. Subsequently, Process 4 sends $m_3 = (i, n)$ synchronously to Process 3. After these control messages, Process 2 sends the characters $m_i^1 = x_i^1, ..., m_i^c = x_i^c$ synchronously to Process 3 (where c is the length of x_i). This MSC is depicted in Figure 3.1,

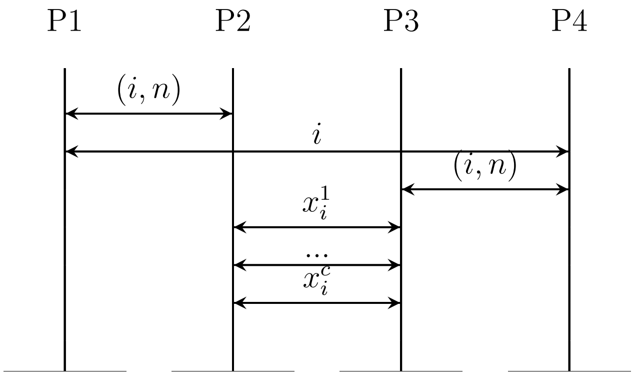

Figure 3.1: The M_i^n MSC.

Given a RPCP instance $\{(v_1, w_1), \ldots, (v_m, w_m)\}$, we associate with each pair (v_i, w_i) two MSCs M_i^0 and M_i^1, following Definition 3.1.2. Each MSC M_i^n is *synchronous* (Lemma 1). Intuitively, the MSC M_i^n encodes the construction of a string given some tiles through the interaction of four processes. Processes 2 and 3 are responsible for building the string itself, while Processes 1 and 4 transmit the index information to Processes 2 and 3, respectively. In particular, Process 1 initiates the choice and forwards it to Process 4. This encoding applies equally to definition 3.1.4.

Lemma 1. *The MSC M_i^n belongs to $\mathcal{M}_{\mathsf{synch}}$.*

Proof. By Definition 2.2.5, each communication in M_i^n consists of a send event $!m^{p\to q}$ and its corresponding receive event $?m^{p\to q}$, with $\mathsf{src}(?m^{p\to q}) = !m^{p\to q}$. By Definition 2.2.3, an MSC is synchronous if there exists a linearisation in which every send event immediately precedes its matching receive.

In M_i^n, the set of messages exchanged is

$$\{\, m_1, m_2, m_3, m_i^1, \ldots, m_i^c \,\},$$

with $c = |x_i|$. A valid synchronous linearisation therefore exists and is given by:

$$!m_1?m_1 \;\; !m_2?m_2 \;\; !m_3?m_3 \;\; !m_i^1?m_i^1 \;\; \ldots !m_i^c?m_i^c.$$

This linearisation satisfies the synchronous ordering constraint, as every send is immediately followed by its matching receive. Hence, by Definition 2.2.8, $M_i^n \in \mathcal{M}_{\mathsf{synch}}$. □

We now define how to obtain a Global Type on top of an MSC.

Definition 3.1.3 (G_M)**.** Let $M \in \mathcal{M}_{\mathsf{synch}}$ be a message sequence chart (MSC) over the set of processes $\mathbb{P}$ and messages $\mathbb{M}$. We construct the corresponding Global Type $\mathsf{G}_M = (Q, \Sigma, \delta, q_0, F)$ as follows.

- The alphabet Σ is the set of *synchronous communication arrows*

$$\Sigma = \{\, p \xrightarrow{m} q \mid \exists\, \eta_s \in \mathsf{events}_S(M),\ \eta_r \in \mathsf{events}_R(M) \text{ such that}$$

$$\mathsf{src}(\eta_r) = \eta_s,\ \mathsf{proc}(\eta_s) = p,\ \mathsf{proc}(\eta_r) = q,\ \mathsf{msg}(\eta_s) = m \,\}.$$

- Since $M \in \mathcal{M}_{\mathsf{synch}}$, there exists a *synchronous linearisation* $w = \alpha_1\alpha_2 \dots \alpha_k$, where each $\alpha_j = p_j \xrightarrow{m_j} q_j \in \Sigma$ represents a complete synchronous communication step (send immediately followed by its matching receive).
- Let $Q = \{q_0, q_1, \dots, q_k\}$, where q_0 is the initial state and q_k is the unique accepting state.
- The transition function $\delta : Q \times \Sigma \to Q$ is defined sequentially along the synchronous interactions of M:

$$\forall j \in \{1, \dots, k\}, \quad \delta(q_{j-1}, \alpha_j) = q_j.$$

  All other transitions are undefined.

Intuitively, G_M captures the exact synchronous execution order of M: it is the deterministic automaton that accepts the single word $w = \alpha_1\alpha_2 \dots \alpha_k$ over Σ, representing the sequence of message exchanges in M. Equivalently, G_M recognises the set of MSCs whose synchronous linearisations are w. We now use this definition to encode M_i^n in a Global Type format.

Definition 3.1.4 (G_i^n)**.** Let (v_i, w_i) be a tile and $n \in \{0, 1\}$. Set $x_i = v_i$ if $n = 0$, and $x_i = w_i$ if $n = 1$. Let $c = |x_i|$ and write $x_i = x_i^1 x_i^2 \cdots x_i^c$.

The global type G_i^n (shown in Figure 3.2) is the DFA $G_i^n = (Q, \Sigma, \delta, q_0, F)$, obtained on top of the MSC M_i^n (Definition 3.1.2) using the construction in Definition 3.1.3, and defined as follows:

- $\mathbb{P} = \{P1, P2, P3, P4\}$;

- $\mathbb{M} = \{m_1, m_2, m_3, m_{x_i^1}, \ldots, m_{x_i^c}\}$, where $m_1 = (i, n)$, $m_2 = i$, $m_3 = (i, n)$, and $m_{x_i^j} = x_i^j$ for $1 \leq j \leq c$;
- $\text{Arr} = \{\, P1 \xleftrightarrow{m_1} P2 \,, P1 \xleftrightarrow{m_2} P4 \,, P4 \xleftrightarrow{m_3} P3 \,, P2 \xleftrightarrow{m_{x_i^1}} P3 \,, \ldots, P2 \xleftrightarrow{m_{x_i^c}} P3 \,\}$, where each arrow denotes a synchronous message with acknowledgment;
- $Q = \{q_0, q_1, q_2, q_3, q_4, \ldots, q_{3+c}\}$, where q_0 is the initial state and $F = \{q_{3+c}\}$ is the unique accepting state;
- The alphabet Σ is the finite set of arrows (synchronous message labels): $\Sigma = \{\, P1 \xleftrightarrow{m_1} P2 \,, P1 \xleftrightarrow{m_2} P4 \,, P4 \xleftrightarrow{m_3} P3 \,\} \cup \{\, P2 \xleftrightarrow{m_{x_i^j}} P3 \mid 1 \leq j \leq c \,\}$.
- The (deterministic) transition function $\delta : Q \times \Sigma \to Q$ is defined by: $\delta(q_0, P1 \xleftrightarrow{m_1} P2\,) = q_1$, $\delta(q_1, P1 \xleftrightarrow{m_2} P4\,) = q_2$, $\delta(q_2, P4 \xleftrightarrow{m_3} P3\,) = q_3$, $\delta(q_{2+j}, P2 \xleftrightarrow{m_{x_i^{j+1}}} P3\,) = q_{3+j}$, $0 \leq j < c$. Hence, the sequence of transitions from q_3 to q_{3+c} corresponds to the synchronous exchanges between q and r labelled by $m_{x_i^1}, \ldots, m_{x_i^c}$.

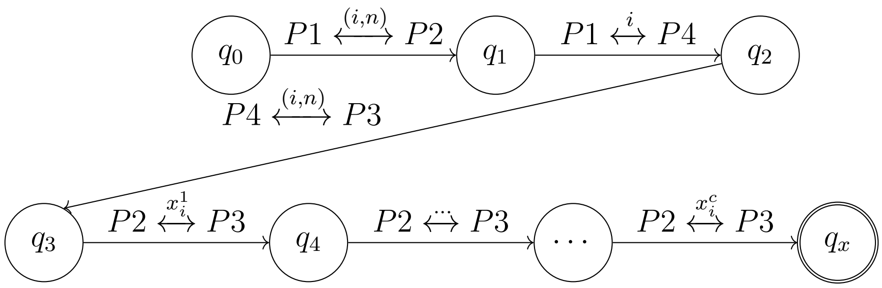

Figure 3.2: The global type G_i^n.

Intuitively, using the costruction defined in Definition 3.1.3, we obtain a Global Type whose existential language of representation correspond exactly to the given MSC.

Lemma 2. *For every MSC $M \in \mathcal{M}_{\mathsf{synch}}$, there exists a Global Type G_M such that $\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G}_M) = \{M\}$.*

Proof. Let $M \in \mathcal{M}_{\mathsf{synch}}$ and let G_M be the Global Type constructed as in Definition 3.1.3. By definition, G_M is the DFA that accepts exactly the synchronous linearisation $w = \alpha_1\alpha_2 \ldots \alpha_k$ of M, where each $\alpha_j = p_j \xrightarrow{m_j} q_j$ corresponds to a complete synchronous communication. Since every execution of G_M under the synchronous semantics follows the exact sequence of communications in w, it induces the same causal and message relations as in M. Hence, the unique MSC corresponding to any accepting run of G_M is precisely M. Conversely, any MSC M'

whose synchronous linearisation is accepted by G_M must have the same sequence of synchronous interactions as M, and therefore $M' = M$. Therefore, the existential MSC-language of G_M contains exactly M, that is,

$$\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G}_M) = \{M\}.$$

$\square$

Lemma 2 establishes a direct correspondence between a single synchronous MSC and a Global Type. In particular, every synchroonus MSC can be captured precisely by a Global Type whose language contains only that MSC. This correspondence will be useful when embedding RPCP instances into the Global Type framework. We now introduce a more structured Global Type, parameterized by a string S, which will serve as the building block in the reduction.

Thanks to Lemma 2 and given that M_i^n is a synchronous MSC (Lemma 1), we can establish now that $\mathcal{L}^{\exists}_{\mathsf{msc}}(G_i^n) = \{M_i^n\}$. After establishing the connection between the MSC's encoding and Global Type's encoding, we briefly summarize the rationale behind the design of M_i^n and G_i^n.

Suppose that $\Delta = (i_1, a_1, b_1, \dots, i_m, a_m, b_m)$ is a solution to the RPCP instance. From this solution we construct two MSCs sequences:

$$M_x = M_{i_1}^{a_1} \cdots M_{i_m}^{a_m}, \qquad M_y = M_{i_1}^{b_1} \cdots M_{i_m}^{b_m}.$$

Both M_x and M_y are concatenations of synchronous MSCs. We then define a third MSC $M_{\mathtt{sol}}$, obtained by projecting M_y onto processes $P1, P2$ and M_x onto processes $P3, P4$. Intuitively, processes $P1, P2$ represent the construction of the *right-hand string* $y_{i_1} \cdots y_{i_m}$, while processes $P3, P4$ represent the construction of the *left-hand string* $x_{i_1} \cdots x_{i_m}$. The prefix property of RPCP guarantees that $M_{\mathtt{sol}}$ is acyclic and *synchronous*. Establishing that $M_{\mathtt{sol}} \in \mathcal{M}_{\mathsf{synch}}$ is non-trivial, and this step is an addition to the original proof.

With these constructions in place, we proceed to introduce the main objects used in the proof. Specifically, we first show how a Global Type can represent a single `synch` MSC.

The following definition introduces the global type L^*, which encapsulates all possible compositions derived from a given RPCP instance. Intuitively, for each MSC $M \in \mathcal{M}$, there exists a corresponding global type $G \in G^*$ that captures the behaviour described by M. The automaton defining L_N^* then combines all such global types in G^* into a single structure, allowing transitions between them through ε-moves. The determinisation of this automaton yields the global type L^*, representing the full set of possible interactions generated by the collection of MSCs.

Definition 3.1.5 (The L^* global type)**.** Given an instance $\{(v_1, w_1), \ldots, (v_m, w_m)\}$ of RPCP, we construct a set $M^* = \{M_i^0, M_i^1 \mid i \in \{1, \ldots, m\}\}$ of MSCs over four processes as follows. For each pair (v_i, w_i), we define two MSC, M_i^0 and M_i^1, as specified in Definition 3.1.2 and illustrated in Figure 3.1. For each MSC in M^*, we construct G^* using Definition 3.1.3. Every Global Type in G^* is shaped like Definition 3.1.4 (shown in Figure 3.2). We define the global type L_N^* as the automaton $\mathcal{A} = (Q, \Sigma, \delta, l_0, F)$ where:

- $Q = \{v_I, v_T\} \cup \bigcup_{G \in G^*} Q^G$;
- $\Sigma = \{\epsilon\} \cup \bigcup_{G \in G^*} \Sigma^G$;
- $\delta : Q \times \Sigma \to 2^Q$ is defined by:
  1. $\forall G \in G^*,\ \delta(v_I, \varepsilon) = q_0^G$ where q_0^G is the initial state of G,
  2. $\forall G \in G^*,\ \forall q_f^G \in F^G,\ \delta(q_f^G, \varepsilon) = v_T$,
  3. $\forall G, G' \in G^*,\ \forall q_f^G \in F^G,\ \delta(q_f^G, \varepsilon) = q_0^{G'}$.
- $l_0 = v_I$ is the initial state;
- $F = v_T$ is the accepting state.

The automaton of L_N^* is shown in Figure 3.3. Finally, the Global Type L^* is obtained as the determinisation of L_N^* (Definition 2.1.3).

In other words, L^* is a Global Type whose language of executions captures all possible combinations and exchanges of choices arising in a generic instance of the RPCP problem. This global type is constructed from the family of global types representing all tiles, and it forms the basis for proving the non-realisability result.

Given the system of CFSMs $\mathsf{proj}(L^*)$ and the MSC $M_{\mathtt{sol}}$, we need to show that $\mathcal{L}_{\mathsf{exec}}^{\mathsf{synch}}(\mathsf{proj}(L^*)) \neq \mathcal{L}_{\mathsf{exec}}^{\mathsf{synch}}(L^*)$. By construction of L^*, we have $M_{\mathtt{sol}} \in \mathcal{L}_{\mathsf{msc}}^{\mathsf{synch}}(\mathsf{proj}(L^*))$. In contrast, $M_{\mathtt{sol}} \notin \mathcal{L}_{\mathsf{msc}}^{\mathsf{synch}}(L^*)$, since at least one tile differs. This demonstrates that there exists an execution that is valid for $\mathsf{proj}(L^*)$ but invalid for the Global Type L^*. Therefore, L^* is *not realisable*.

3.2 Undecidability proof

Given the definitions and lemmas stated in the last section, we are now ready to present the proof for the undecidability result.

Theorem 1. *Given a global type G, checking if G is weakly-realisable is undecidable.*

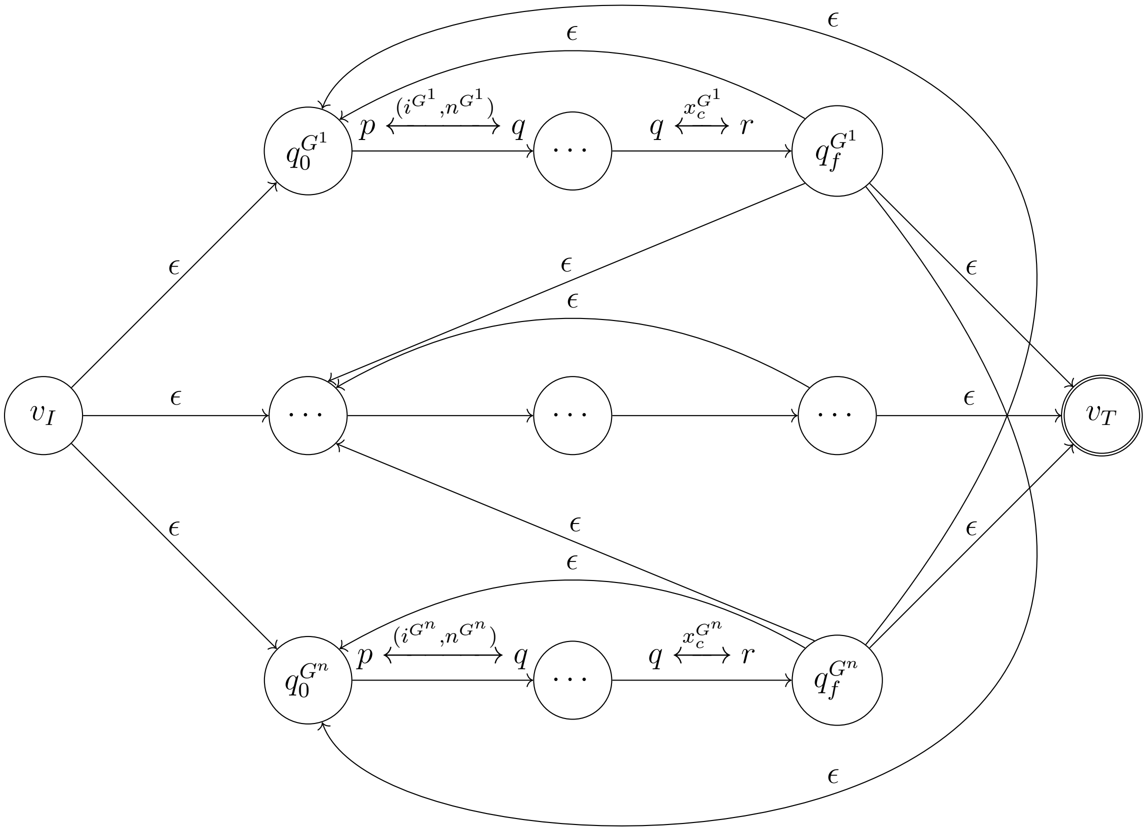

Figure 3.3: The automaton of the global type L_N^*.

Proof. The proof proceeds via a reduction from the RPCP problem. Given an instance $\{(v_1, w_1), \ldots, (v_m, w_m)\}$ of RPCP, we construct L^* as specified in Definition 3.1.5.

We need to prove:

$$\Delta \in \text{RPCP iff the global type } L^* \text{ is not weakly-realisable.}$$

$\Rightarrow$ Assume that $\Delta = (i_1, a_1, b_1, i_2, a_2, b_2, \ldots, i_m, a_m, b_m)$ are the indices for a solution to a generic RPCP problem instance, and the bits a_j and b_j indicate which string (v_{i_j} or w_{i_j}) is chosen to go into the two (left and right) long strings. Assume also synchronous communication semantic. Consider the MSCs M_x and M_y obtained from the concatenation of $M_x = M_{i_1}^{a_1} \cdots M_{i_m}^{a_m}$ and

$M_y = M_{i_1}^{b_1} \cdots M_{i_m}^{b_m}$. The possible linearisations of both of these sequences of MSCs must be included in the language of execution of L^*, by construction of L^* (Definition 3.1.5). This means that $M_x, M_y \in \mathcal{L}_{\mathsf{msc}}^{\mathsf{synch}}(L^*)$. Additionally, $M_x, M_y \in \mathcal{M}_{\mathsf{synch}}$ because they are sequences of MSCs included in $\mathcal{M}_{\mathsf{synch}}$ (Lemma 1). M_x corresponds to the construction of the left side of the equivalence of the RPCP problem, and, instead, M_y represents the construction of the right side. We then look at the projections $M_x|_{P1}$, $M_x|_{P2}$, $M_x|_{P3}$, and $M_x|_{P4}$ of M_x, and $M_y|_{P1}$, $M_y|_{P2}$, $M_y|_{P3}$, $M_y|_{P4}$ of M_y onto the 4 processes. Given that these are projection of MSCs included in L^*, they are possible execution of a CFSM $\mathcal{S}$ that can execute L^*. Now consider the MSC $M_{\mathtt{sol}}$ formed from $M_y|_{P1}$, $M_y|_{P2}$, $M_x|_{P3}$, and $M_x|_{P4}$. This MSC represents the construction of the solution to the problem. Processes 1 and 2 construct the right part $(y_{i_1}...y_{i_m})$ and processes 3 and 4 construct the left part $(x_{i_1}...x_{i_m})$. The claim is that the combined MSC $M_{\mathtt{sol}}$ is made by L^*'s projections, therefore, it exists a CFSM $\mathcal{S}$ that $M_{\mathtt{sol}} \in \mathcal{L}_{\mathsf{msc}}^{\mathsf{synch}}(\mathcal{S})$, but the CFSM $\mathcal{S}$ is not part of the language of execution of L^* $\mathcal{L}_{\mathsf{exec}}^{\mathsf{synch}}(\mathcal{S}) \neq \mathcal{L}_{\mathsf{exec}}^{\mathsf{synch}}(L^*)$. In other words, the language of the execution of $M_{\mathtt{sol}}$ is included in the execution of the system, but it is not included in the execution of L^*. By definition, the first thing to establish is that $M_{\mathtt{sol}}$ is indeed well-formed and synchronous MSC. The only new situation in terms of communication in $M_{\mathtt{sol}}$ is the communication between P_1 and P_4, and between P_2 and P_3. But the communication between P_1 and P_4 is consistent in $M_y|_{P1}$ and $M_x|_{P4}$ (i.e., the sequence of messages sent from P_1 to P_4 in $M_y|_{P1}$ is equal to the sequence of messages received in $M_x|_{P4}$), and the communication between P_2 and P_3 is consistent in $M_y|_{P2}$ and $M_x|_{P3}$ because R is a solution to the RPCP. Furthermore, the acyclicity of $M_{\mathtt{sol}}$ follows from the property of the solution that the string formed by the first j words on processes 1 and 2 is always a prefix of the string formed by the first j words on processes 3 and 4. Consequently, each message from P_1 to P_4 is sent before it needs to be received. Therefore, the MSC $M_{\mathtt{sol}}$ is well-formed.

We now prove that the MSC $M_{\mathtt{sol}}$ is synchronous, that is $M_{\mathtt{sol}} \in \mathcal{M}_{\mathsf{synch}}$. Assume, by contradiction, that $M_{\mathtt{sol}} \notin \mathcal{M}_{\mathsf{synch}}$. Then, there should be a cycle of dependencies in the communication pattern. There are no communication between P_2 and P_4, and between P_1 and P_3. Therefore, this cycle must involve all processes, starting for example from P_1 and having this dependency graph $P_1 \leftrightarrow P_2 \leftrightarrow P_3 \leftrightarrow P_4 \leftrightarrow P_1$. The only new situation that can cause a cycle are the communication between P_1 and P_4, and between P_2 and P_3. We do not need to analyse the new communication between P_1 and P_4 because it is not feasible in any communication model, but we need to analyse the one between P_2 and P_3 because it's feasible in FIFO.

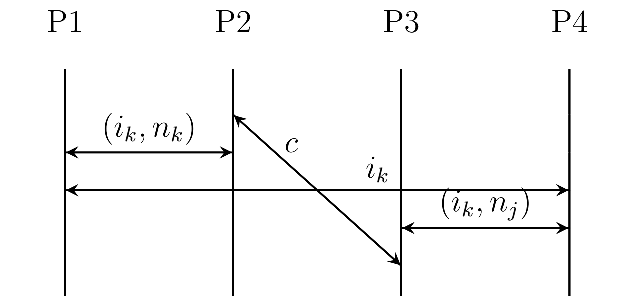

Figure 3.4: MSC communication that breaks synchrony.

For the communication between P_2 and P_3, the only possible cycle pattern is depicted in Figure 3.4 showed as an MSC. Suppose P_2 wants to send a character c, but P_3 is not expecting any further characters. In order for P_3 to resume receiving, it must first receive an index from P_4. However, P_4 can only send this index after receiving it from P_1, which in turn must first communicate the index to P_2. At this point, P_2 needs to receive the index from P_1, but it cannot do so until it finishes sending character c. This creates a circular dependency among the processes, making the communication pattern impossible. This cycle would break the prefix property as $x_1...x_{k-1}...x_m = y_1...y_{k-1}...y_m$, but the character c appears in $y_1...y_{k-1}$ but not in $x_1...y_{k-1}$ contradicting the assumption that $y_1...y_{k-1} \leq x_1...x_{k-1}$. Therefore, we conclude that $M_{\mathtt{sol}} \in \mathcal{M}_{\mathsf{synch}}$.

We now prove the non-realisability of L^*, thanks to $M_{\mathtt{sol}}$. Consider the system of CFSM $\mathsf{proj}(L^*)$ and a linearisation $w_{\mathtt{sol}} \in \mathsf{lin}(M_{\mathtt{sol}})$. We need to prove that $\mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathsf{proj}(L^*)) \neq \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(L^*)$. Given that $M_{\mathtt{sol}}$ is composed by projections of MSCs used to build L^*, we can establish $M_{\mathtt{sol}} \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(\mathsf{proj}(L^*))$. Thanks to $M_{\mathtt{sol}}$, we can notice that $\mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathsf{proj}(L^*))$ cannot itself be in $\mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(L^*)$ because there must be some index i_j where $a_j \neq b_j$, and no execution of the Global Type exists in L^* where, after P_1 announces the index, what P_2 sends is not identical to what P_3 receives. $M_{\mathtt{sol}}$ rapresents the possible execution that establish the inequality. More formally, $w_{\mathtt{sol}} \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathsf{proj}(L^*))$, but $w_{\mathtt{sol}} \notin \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(L^*)$. This generally establish the non-realisability of L^*. Example 3.2.1 shows an instance of the construction of $M_{\mathtt{sol}}$.

$\Leftarrow$ Suppose there is some MSC $M^{@}$ that can be built from possible projections of L^*, but is not part of L^*'s language of executions. More precisely, we want to derive a solution to Δ from $M^{@}$. First, it is clear that the projection $M^{@}|_{P1}$ consists of a sequence of pairs of messages (the first of each pair acknowledged), sent from process 1 to processes 2 and 4, respectively, with messages (i, b) and i. Likewise, in order for process 2 to receive those messages, $M^{@}|_{P2}$ consists of a sequence of receipts of (i, b) pairs, and after each (i, b), either v_i or w_i is sent to process 3, based on whether $b = 0$ or $b = 1$, before the next index pair is received. Likewise, $M^{@}|_{P4}$ consists of a sequence of receipts of index i from process 1, followed by sending of $(i, 0)$ or $(i, 1)$ to process 3, and $M^{@}|_{P3}$ consists of a sequence of receipt of $(i, 0)$ or $(i, 1)$ followed by receipt of v_i or w_i, respectively. Now, since $M^{@} \notin \mathcal{L}^{\mathsf{synch}}_{\mathsf{msc}}(L^*)$, for some index i the choice of v_i or w_i must differ on process 2 and process 3. (Note, we are assuming that the buffers between processes are FIFO.) Furthermore, because of the precedences, the prefix formed by the first j words on process 2 must precede the $(j+1)$-th message from process 1 to process 4, which in turn precedes the $(j+1)$-th message from 4 to 3, and hence the $(j+1)$-th word on process 3. That is, the string formed by the first j words on process 2 is a prefix of the string formed by the first j words on process 3. Therefore, we can readily build a solution for Δ from $M^{@}$ by building the strings of the solution taking the projections of P_1 and P_4. In fact, P_1 builds $y_{i_1} \cdots y_{i_m}$, and P_4 builds $x_{i_1} \cdots x_{i_m}$.

$\square$

In this example, we will show the step-by-step construction of $M_{\mathtt{sol}}$ from Theorem 1.

Example 3.2.1 ($M_{\mathtt{sol}}$ Example of Theorem 1)**.** Consider the tiles and the solution of the RPCP instance in Example 3.1.1, with the tile set and the solution with index sequence $(2, 1, 3)$

$$(v_1, w_1) = (\mathtt{b}, \mathtt{bb}),\ (v_2, w_2) = (\mathtt{a}, \mathtt{ab}),\ (v_3, w_3) = (\mathtt{c}, \mathtt{c}).$$

$$x_1 = w_2,\ y_1 = v_2; \quad x_2 = v_1,\ y_2 = w_1; \quad x_3 = v_3,\ y_3 = w_3$$

This sequence is a solution because $x_1x_2x_3 = \mathtt{ab}\,\mathtt{b}\,\mathtt{c} = \mathtt{abbc}$ and $y_1y_2y_3 = \mathtt{a}\,\mathtt{bb}\,\mathtt{c} = \mathtt{abbc}$. The prefix property and the "some index differs" condition are satisfied.

Therefore, the encoding of the solution is

$$\Delta = (i_1 = 2, a_1 = 1, b_1 = 0, i_2 = 1, a_2 = 0, b_2 = 1, i_3 = 3, a_3 = 0, b_3 = 1)$$

Recall that for each tile of index i we have two synchronous MSCs M_i^0 and M_i^1 (see Definition 3.1.2), where the bit indicates choosing v_i (0) or w_i (1) for the character comunication. Using the concrete index sequence $(2, 1, 3)$ we form two concatenated MSCs:

$$M_x = M_2^1 \cdot M_1^0 \cdot M_3^0, \qquad M_y = M_2^0 \cdot M_1^1 \cdot M_3^1.$$

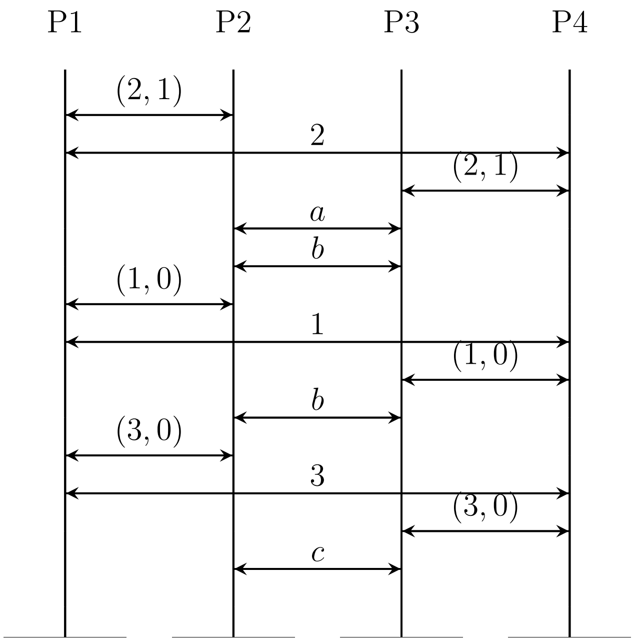

Figure 3.5: The MSC M_x.

Here M_x encodes the **x**-concatenation $(x_1, x_2, x_3) = (w_2, v_1, v_3)$ (depicted in Figure 3.5) and M_y encodes the **y**-concatenation (depicted in Figure 3.6) $(y_1, y_2, y_3) = (v_2, w_1, w_3)$.

Recall that $M|_p$ denotes the projection of M onto process p.

We construct the MSC $M_{\mathtt{sol}}$, using M_x and M_y projections, as follows:

$$M_{\mathtt{sol}} = (M_y|_{P1},\ M_y|_{P2},\ M_x|_{P3},\ M_x|_{P4}),$$

i.e. processes $1, 2$ follow M_y while $3, 4$ follow M_x. Intuitively, $M_{\mathtt{sol}}$ pairs the right-side construction (from M_y) with the left-side construction (from M_x). Figure 3.7 illustrates the behaviour of the MSC $M_{\mathtt{sol}}$. Observe that when process 3 expects to receive the second character `b` right after a, but process 2 cannot send it immediately: it must first obtain the corresponding index and bit from process 1. The prefix property guarantees that every partial construction of the right-hand side is aligned with a prefix of the left-hand side, therefore preserving synchronous semantics throughout the execution.

From Figure 3.7, it is evident that this kind of execution cannot occur in any execution of L^*, even though the MSC is constructed using valid projections included

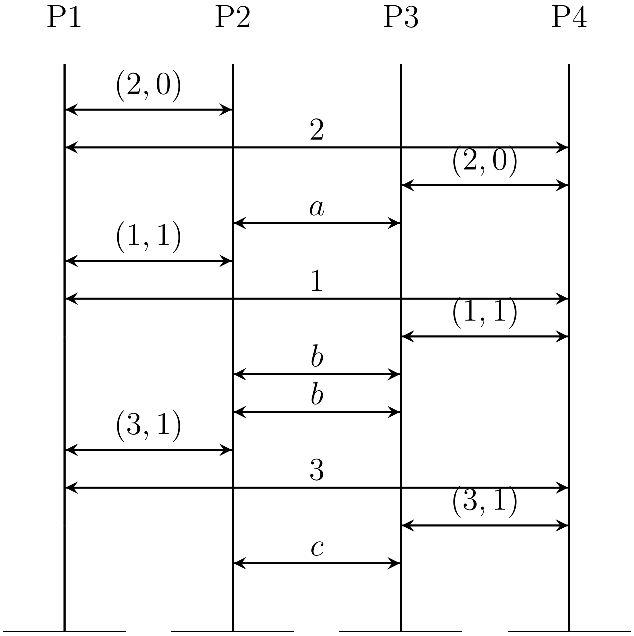

Figure 3.6: The MSC M_y.

in its MSC language. More precisely, the language of L^* contains only communications of the type represented by M_x and M_y. The exchange involving the first b character in Figure 3.7 does not belong to the execution language of L^*, but it belongs to L^*'s projections. This particular communication pattern demonstrates the non-realisability of L^*.

The sequence of lemmas and the main theorem collectively establish the undecidability of weak-realisability for global types. Having developed the theoretical foundation, we now move to the next section, where we focus on the practical aspects of analysing realisability, and introduce the ReSCu tool.

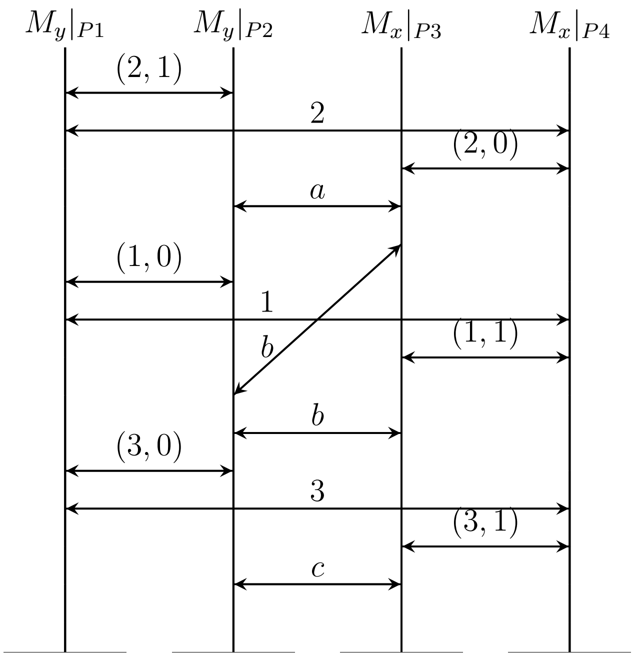

Figure 3.7: The MSC $M_{\texttt{sol}}$.

Chapter 4

ReSCu

In the previous chapter, we examined the theoretical aspects of the realisability problem for MPST, culminating in the main result: the **undecidability** of weak realisability under synchronous semantics. This analysis not only establishes a fundamental limitation but also motivates the exploration of complementary directions, such as identifying restricted, decidable subclasses or developing practical verification techniques that can be effectively applied in real-world settings.

In this chapter, we shift our focus from *undecidability* to *decidability*. As introduced in Section 1.2, our goal is to analyse the *realisability problem* through a reduction approach: specifically, we reduce the study of realisability in the peer-to-peer (*p2p*) communication model to the study of realisability under the *synchronous semantics*. This result was showed before without a formal definition, but now, building upon the theoretical framework developed in Chapters 2 and 3, we can state this result rigorously.

Theorem 2 (Reduction to synch-implementability, Theorem 5.3 in [14]). *A global type* G *is* deadlock-free realisable *in* $\mathsf{p2p}$ *if and only if the following conditions hold:*

1. $\mathcal{L}^{\mathsf{p2p}}_{\mathsf{msc}}(\mathsf{proj}(\mathsf{G})) \subseteq \mathsf{Prefixes}(\mathcal{M}_{\mathsf{synch}})$;
2. $\mathsf{proj}(\mathsf{G})$ *is orphan-free in* $\mathsf{p2p}$;
3. $\mathcal{L}^{\mathsf{p2p}}_{\mathsf{msc}}(\mathsf{proj}(\widehat{\mathsf{G}})) \subseteq \mathsf{Prefixes}(\mathcal{L}^{\mathsf{p2p}}_{\mathsf{msc}}(\mathsf{proj}(\mathsf{G})))$;
4. G *is deadlock-free realisable in* synch.

An informal explanation of the four conditions is as follows:

- the language generated by the local projections of G is compatible with synchronous semantics;

- all projections of G are *orphan-free*;
- every trace in the MSC language of G corresponds to a *deadlock-free* execution in the `p2p` model;
- G is *deadlock-free realisable* under synchronous semantics.

A detailed proof of this theorem is provided in the referenced paper. Notably, the first three conditions have already been proven *decidable* [18, 14]. The remaining challenge, addressed in this thesis, concerns the *decidability of deadlock-free realisability* in the synchronous model, which constitutes the central focus of our analysis.

This theoretical result directly motivates the practical contribution of this work: the extension of the ReSCu tool toward fully automated verification within our modular framework. ReSCu (first introduced in [10, 13, 18]) is a verification tool for reasoning about communication models and protocol design. Originally, ReSCu was developed to verify membership in the class of synchronous systems and to check reachability of specific configurations under synchronous executions. The tool already supports verification of the first and second conditions of the theorem. Consequently, the next natural step was to extend its capabilities toward analysing the third and fourth conditions, both related to *deadlock-freedom*. Building on this foundation, we enhanced ReSCu to reason about key behavioural properties, specifically, *deadlock-freedom* and *progress*, which are central to the broader study of realisability. Examining deadlock-freedom in this context is particularly valuable, as it restricts the space of possible executions and often yields a *decidable* verification problem. Through these extensions, ReSCu bridges the gap between the theoretical foundations of realisability and practical, automated analysis, serving as a *building block* toward decidable realisability checks in distributed systems.

We describe the features of ReSCu, the input language it adopts, and its implementation details, with particular emphasis on the extensions and modifications we introduced to improve its capabilities [11]. The updated public repository, which includes the new features and illustrative examples, is available at:

`https://github.com/gabrielegenovese/rescu` [16].

4.1 Characteristics

ReSCu is a command-line tool that can check both membership in the class of `synch` systems (called Realisable with Synchronous Communication or, in brief, RSC from now on) and reachability of regular sets of configurations. It accepts input systems with arbitrary topologies and supports FIFO and bag buffers among others. The tool provides several options: `-isrsc` checks whether the system is RSC, and `-mc` checks reachability of bad configurations. Both checks can be combined in a single run. The `-fifo` option overrides buffer types by treating all as FIFO. When a system is unsafe, the `-counter` option (used with `-mc`) produces an RSC execution that leads to the bad configuration, while the same option used with `-isrsc` outputs the violation execution if the system is not RSC. Additional features include a progress display to estimate remaining runtime during long computations, and `-to_dot`, which exports the system to DOT format for visualization. One of the most similar tools is McScm [19], that uses a framework with different verification techniques. Symbolic Communicating Machines (SCM), defined and used in [25, Definition 5.1] serve as the input format of the tool. SCMs are Communicating Finite-State Machines (CFSM, Definition 2.2.9) extended with the use of channels and a finite set of variables (that corresponds to message). The grammar has been updated to provide greater flexibility and clarity. In particular, transition guards have been made optional (with a default value `: when true`), and a new `final` keyword has been introduced to explicitly specify final states. The updated grammar is shown in Listing 4.1.

```
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>;<final>; <state_list>
initial       ::= initial : <int_list>;
final         ::= final : <int_list>;
state_list    ::= <state>
                | <state_list> <state>
state         ::= state <int> : <trans_list>
trans_list    ::= <transition>
                | <trans_list> <transition>
guard         ::= : when true | <nothing>
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>
```

```
nothing        ::=
```

Listing 4.1: Modified SCM grammar

Given the definition of SCM and the newly introduced input grammar, We now present an example to illustrate how these concepts are applied in practice with the tool. For clarity, the example is expressed in the CFSM notation rather than in the SCM formalism. Consequently, channels and variables are omitted and replaced directly by messages. However, the figures are displayed in SCM format, as they are automatically generated by the tool.

Example 4.1.1 (Ping-Pong Example)**.** Let the set of processes be $\mathbb{P} = \{A, B\}$, the set of messages $\mathbb{M} = \{\mathtt{ping}, \mathtt{pong}\}$, and the set of channels consist of a single FIFO channel 0 from A to B and from B to A. The corresponding actions are

$$\mathsf{Act} = \{\, (A, B, !, \mathtt{ping}), (B, A, ?, \mathtt{ping}), (B, A, !, \mathtt{pong}), (A, B, ?, \mathtt{pong}) \,\}.$$

The system of CFSMs is $\mathcal{S} = (\mathcal{A}_A, \mathcal{A}_B)$, where:

$$\mathcal{A}_A = (Q_A, \Sigma, \delta_A, q_{0,A}, F_A)$$

with

- $Q_A = \{0, 1, 2\}$, initial state $q_{0,A} = 0$, final state $F_A = \{2\}$,
- transitions: $0 \xrightarrow{(A,B,!,\mathtt{ping})} 1 \xrightarrow{(B,A,?,\mathtt{pong})} 2$.

$$\mathcal{A}_B = (Q_B, \Sigma, \delta_B, q_{0,B}, F_B)$$

with

- $Q_B = \{0, 1, 2\}$, initial state $q_{0,B} = 0$, final state $F_B = \{2\}$,
- transitions: $0 \xrightarrow{(B,A,?,\mathtt{ping})} 1 \xrightarrow{(A,B,!,\mathtt{pong})} 2$.

This CFSM system $\mathcal{S}$ is showed in Figure 4.1. The corresponding input as SCM format for the tool is showed in Listing 4.2.

```
scm ping_pong :

nb_channels = 1 ;
parameters :
unit ping ;
unit pong ;

automaton A :
```

```
initial : 0
final : 2

state 0 :
to 1 : 0 ! ping ;
state 1 :
to 2 : 0 ? pong ;
state 2 :

automaton B :
initial : 0
final : 2

state 0 :
to 1 : 0 ? ping ;
state 1 :
to 2 : 0 ! pong ;
state 2 :
```

Listing 4.2: Tool's input for Example 4.1.1

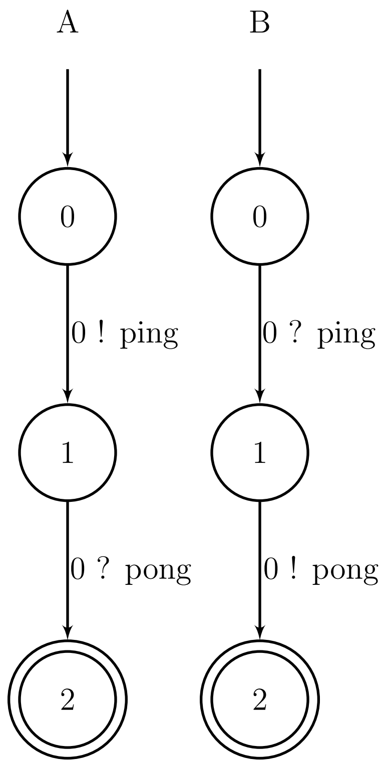

Figure 4.1: Simple Ping-Pong example.

4.2 Progress and Deadlock-Freedom

We extended ReSCu with verification routines that focus on two fundamental correctness properties of distributed systems: *progress* and *deadlock-freedom*. To enable this, the tool constructs the synchronous system using the synchronous product operation whenever the input SCM is recognized as realisable in synchronous communication semantic (RSC). Once the system is proven to be RSC, we can safely construct a well-formed synchronous product from it, and, given the synchronous product, the tool elaborates the other two additional checks.

Remark. The discussion in this chapter assumes *complete nondeterministic fairness* over choices. In other words, whenever the system encounters a nondeterministic branching, all possible continuations are treated equally and none of them can be ignored. This assumption ensures that the verification does not overlook executions simply because they are less probable, and it avoids trivial counterexamples where a branch is never explored. In practice, relaxing fairness assumptions can yield more realistic analyses (e.g. prioritising certain branches or modelling schedulers with biases), but at the cost of complicating the verification procedures. Exploring weaker or alternative fairness models is therefore an interesting direction for future work, especially for applications where nondeterminism is influenced by external constraints such as message delays or resource contention.

We now present the definition of the Synchronous Product for CFSMs, which I have implemented in the tool, and it serves as a key component for the analysis.

Definition 4.2.1 (Synchronous Product)**.** Let $\mathcal{S} = (\mathcal{A}_p)_{p\in\mathbb{P}}$ be a system of CFSMs, where $\mathcal{A}_p = (L_p, Act_p, \delta_p, l_{0,p}, F_p)$ is the CFSM associated to process p.

The *synchronous product* of $\mathcal{S}$ is the FSA $P = \mathrm{prod}_s(\mathcal{S}) = (L, Arr, \delta, l_0, F)$, where

- $L = \prod_{p\in\mathbb{P}} L_p$ is the set of global locations,
- $l_0 = (l_{0,p})_{p\in\mathbb{P}}$ is the initial global state,
- $F = \prod_{p\in\mathbb{P}} F_p$ is the set of global final states,
- δ is the transition relation defined as follows: $(\vec{l},\ p \xrightarrow{m} q,\ \vec{l}\,') \in \delta$ if

$$(l_p,\ !m^{p\to q},\ l'_p) \in \delta_p, \quad (l_q,\ ?m^{p\to q},\ l'_q) \in \delta_q, \quad l'_r = l_r \text{ for all } r \notin \{p, q\}.$$

Example 4.2.1 (Synchronous Product Example)**.** Consider the system of CFSMs $\mathcal{S} = (\mathcal{A}_A, \mathcal{A}_B)$ from the Example 4.1.1. Its synchronous product is $P = \mathrm{prod}_s(\mathcal{S}) = (L, Arr, \delta, l_0, F)$, where

- $L = Q_A \times Q_B = \{0, 1, 2\} \times \{0, 1, 2\}$,

- $l_0 = (0, 0)$,
- $F = \{(2, 2)\}$,
- δ consists of the following transitions: $(0, 0) \xrightarrow{A \xrightarrow{\texttt{ping}} B} (1, 1) \xrightarrow{B \xrightarrow{\texttt{pong}} A} (2, 2)$.

Thus, the synchronous product captures the joint behaviour: process A sends `ping` to B, then B responds with `pong` to A, and both processes reach their final states simultaneously. Figure 4.2 illustrates the product's automaton $\text{prod}_s(\mathcal{S})$.

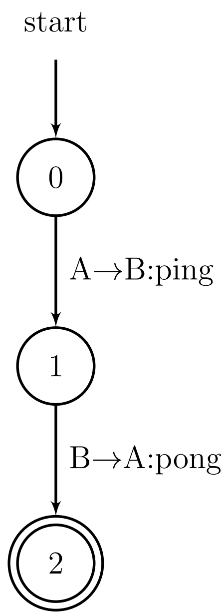

Figure 4.2: Synchronous Product of the CFSM system in Example 4.2.1.

After constructing the synchronous product, the tool performs several important post-processing operations. In particular, it removes any unreachable nodes from the resulting product, simplifying the structure and ensuring that only relevant states are retained for further analysis. We can now define the two SCM properties implemented as verification routines in the tool.

Consider the definition of deadlock-freedom for CFSMs (Definition 2.2.12). We will report the same definition focusing on the synch communication semantic. This implies that the system uses the synchronous product when analysing the executions of the system (Definition 4.2.1).

Definition 4.2.2 (Deadlock-freedom in synch)**.** A system $\mathcal{S}$ is *deadlock-free* in synch if for every execution $e \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\widehat{\mathcal{S}})$, there exists a completion e' with $e \leq_{\mathsf{pref}} e'$ and $e' \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathcal{S})$.

Remark. The notation $\widehat{\mathcal{S}}$ denotes the system obtained by treating every state as an accepting (or final) state. This way, all possible partial executions of the system are taken into account. The deadlock-freedom condition then requires that each such partial execution can be extended to a complete execution of the original system $\mathcal{S}$. Intuitively, this ensures that the system cannot "get stuck" in the middle of a computation, i.e. every execution fragment can always be continued to reach a final state.

More precisely, a system that can reach, from its initial states, some state that does not lead to a final state is not deadlock-free. Under this definition, even a loop that never reaches a final state is considered a deadlock, making the property more restrictive. This check is implemented using a reverse search algorithm starting from the final states.

Definition 4.2.3 (Progress)**.** A system of CFSMs $\mathcal{S}$ satisfies the *progress* property in synch if for every execution $e \in \mathcal{L}_{\mathsf{exec}}(\widehat{P})$, with $P = \mathrm{prod}_s(\mathcal{S})$,

- the execution e is also a valid execution of $e \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(P)$, or
- there exists another execution $e' \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\widehat{P})$ such that $e \leq_{\mathsf{pref}} e'$, with $e \neq e'$.

Intuitively, progress ensures that the system never reaches a state where it is permanently stuck, except in the case of successful termination. This is weaker than deadlock-freedom, since infinite executions are allowed as long as they can always perform a new step. In particular, livelocks (loops without termination) are considered to satisfy progress, but would violate deadlock-freedom.

Lastly, the synchronized system can be exported in DOT format (with a default filename of `sync.dot`), which allows for graphical visualization of its structure and behaviour. Some illustrative examples demonstrating these new features are included in the `examples/deadlock` folder from the online repository [16]. Two of them are showed and explained with details in the next section.

4.3 Examples

To illustrate these notions, we present two examples. The first is the classical *Dining Philosophers* problem, which shows how resource contention can lead to deadlock. The second is a minimal looping system that demonstrates how a process may satisfy the progress property while still failing to be deadlock-free.

4.3.1 The Dining Philosophers

Example 4.3.1. Consider two philosophers P_0, P_1 and two forks F_1, F_2, arranged so that each philosopher needs both forks to eat. If both philosophers pick up their left fork simultaneously, each waits indefinitely for the other fork, producing a deadlock. This captures the essence of the Dining Philosophers problem: concurrent processes blocking one another when competing for shared resources.

```
This system is RSC.
There are some sink states:
Sink: Id=11 Configuration={{ F0:4; F1:3; P1:2; P2:2 }}
There are some deadlock states:
Deadlock: Id=4 Configuration={{ F0:2; F1:1; P1:1; P2:1 }}
Deadlock: Id=11 Configuration={{ F0:4; F1:3; P1:2; P2:2 }}
Deadlock: Id=8 Configuration={{ F0:4; F1:1; P1:1; P2:2 }}
Deadlock: Id=7 Configuration={{ F0:2; F1:3; P1:2; P2:1 }}
```

Listing 4.3: Output of Example 4.3.1

The behaviour of the four participants is shown in Figure 4.3. Running the tool on this input produces the terminal output in Listing 4.3 and the corresponding synchronous system in Figure 4.4. In the generated figure, the red state marks a configuration where no further actions are possible, while the three yellow states correspond to deadlocks, i.e. executions where both philosophers wait for each other indefinitely. The terminal output also lists the precise configurations of these problematic states.

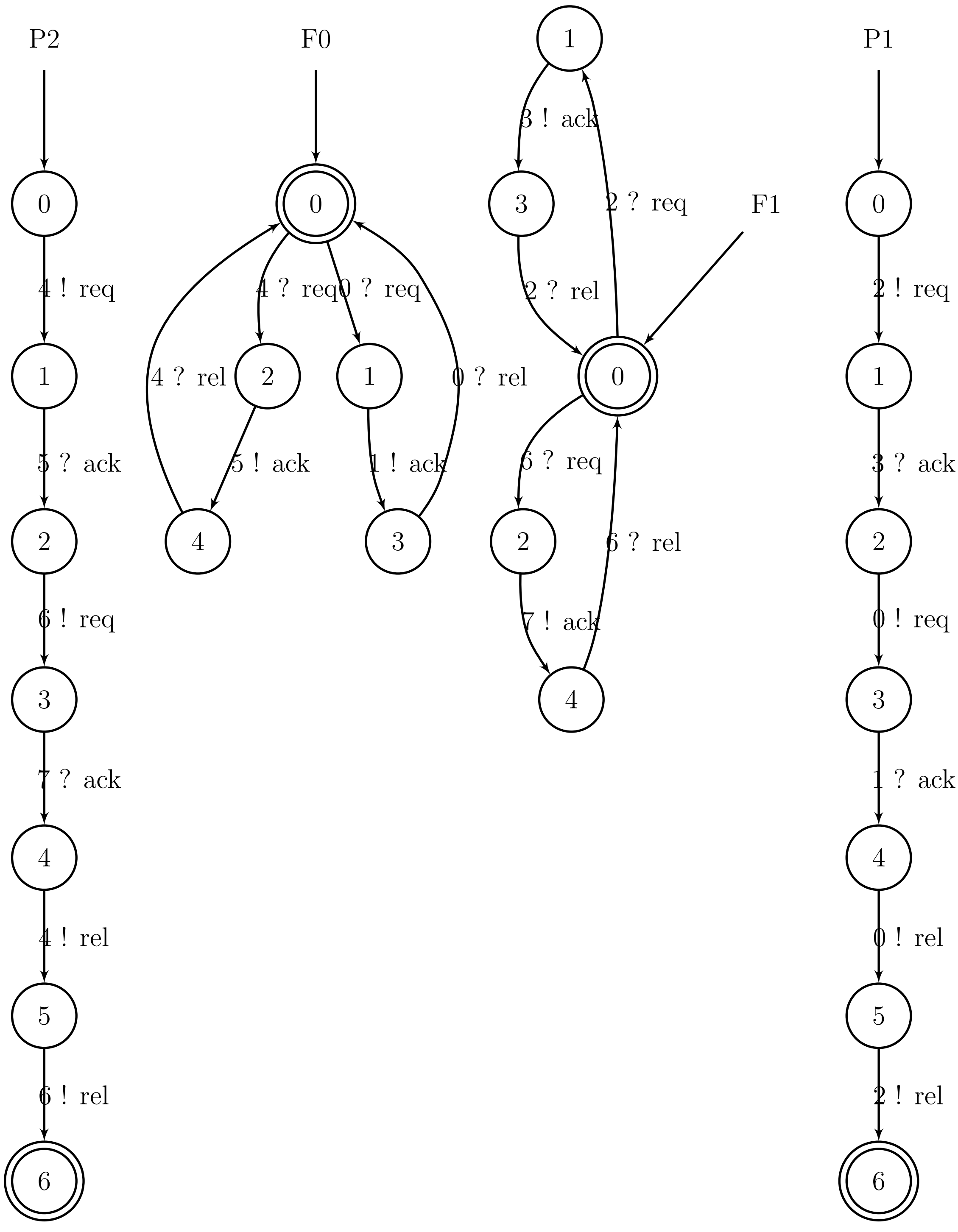

Figure 4.3: SCM automata representation of the Example 4.3.1.

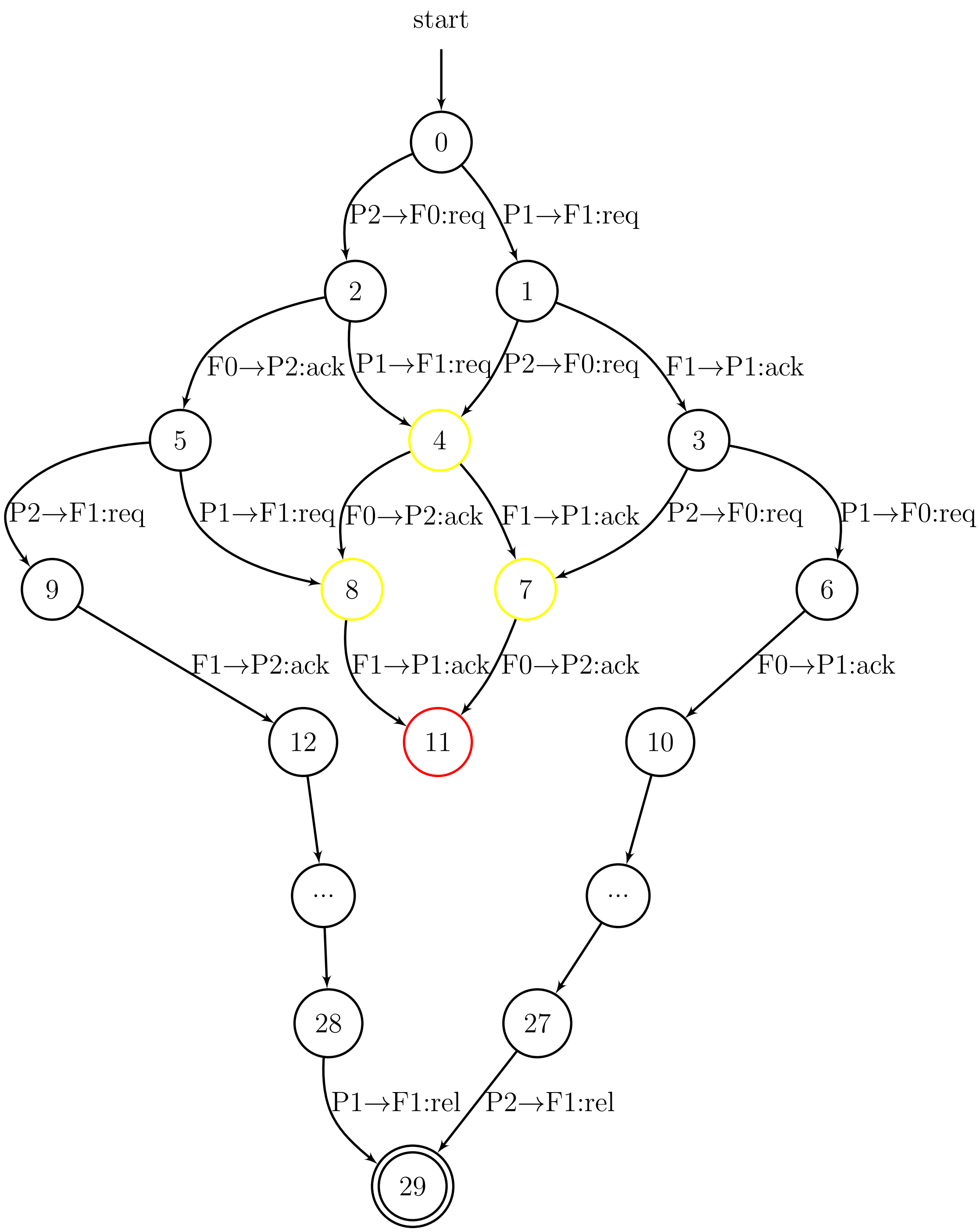

Figure 4.4: Synchronous Product of the Example 4.3.1.

4.3.2 Example with a loop

Example 4.3.2. Now consider two processes A and B that exchange data. At some point, each makes a nondeterministic choice: one branch continues sending messages indefinitely, while the other leads to termination. Once the choice to continue is taken, however, there is no way to return to the terminating branch. As a result, the system may remain stuck in an infinite loop, never reaching a final state. Although both processes remain active, the system is effectively deadlocked.

```
This system is RSC.
The system has the progress property.
There are some deadlock states:
Deadlock: Id=17 Configuration={{ A:1; B:4 }}
Deadlock: Id=15 Configuration={{ A:3; B:3 }}
```

Listing 4.4: Output of Example 4.3.2.

The behaviour of this system is shown in Figure 4.5. Executing the tool produces the output in Listing 4.4 and the synchronous system in Figure 4.6. In the generated figure, yellow states highlight the deadlocked executions, while the terminal output provides the configuration of each detected deadlock.

Remark. If the system contains a loop with at least one possible way out, this execution is still considered without a deadlock thanks to the *fairness* assumption. Fairness ensures that the exit path will eventually be taken.

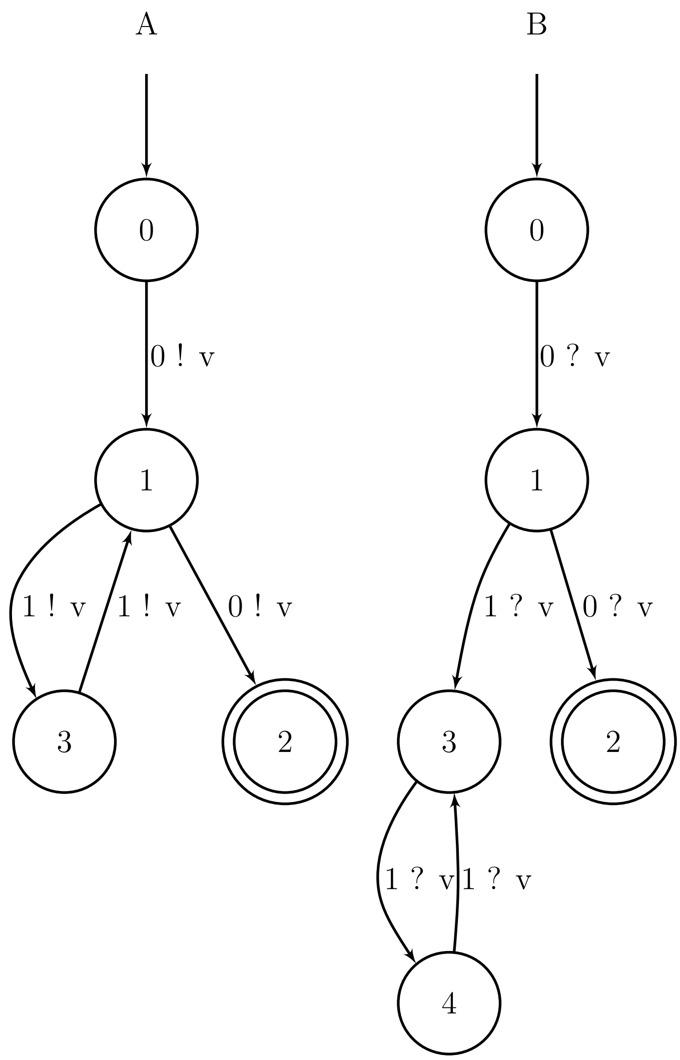

Figure 4.5: SCM automata representation of the Example 4.3.2.

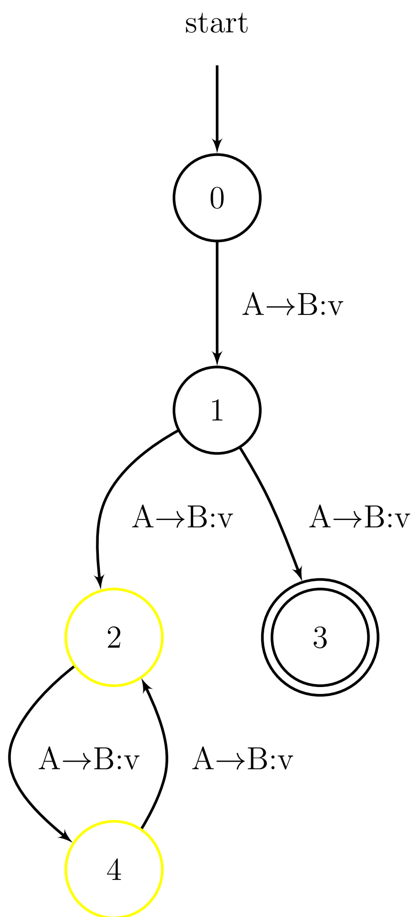

Figure 4.6: Synchronous Product of the Loop Example 4.3.2

Chapter 5

Related work

This thesis is centred around the study of the *realisability problem* for *global types*. In this chapter, We review related works addressing this problem, both within the same formal framework and in comparable models.

In particular, our work uses a definition of global types that rapresents a set of MSCs. This thesis forms part of a broader line of research originating from [12] and later expanded in [14], which aims to develop a general framework for communication models. We first present and discuss the results of these works, positioning my own contributions in relation to them.

Subsequently, We analyse recent results by Stutz et al. [30], who extensively study the realisability problem, while also tracing the line of research back to early contributions such as Alur et al. [1] and Lohrey et al. [27].

Finally, We briefly review related approaches in comparable formal models, such as Multiparty Session Types (MPST) and Choreography Automata [4], highlighting similarities and differences with respect to the problem addressed in this thesis.

5.1 Hierarchy of communication model's semantics

We defined early some communication semantics of out interest, informally, in Chapter 1 and, formally, the synch in Definition 2.2.3. Furthermore, [12] show some other interesting semantics. It also introduces a hierarchy of communication semantics, illustrated in Figure 5.1. The main objective of this work was to establish a hierarchy that preserves *monotonic* properties: if a property holds for a given communication semantic, it should also hold for all semantics contained within it. However, it was shown that this monotonicity only applies to specific properties, such as *weak-k-synchronizability*. In contrast, it does not generally extend to the realisability problem, that is why we focused in particular on certain semantics. We define subsequently some other useful and well known communication semantics.

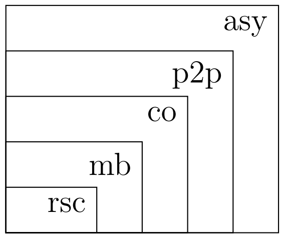

Figure 5.1: Hierarchy of communication model semantics.

Causally ordered In the causally ordered (`co`) communication model, messages are delivered to a process in accordance with the causal dependencies of their emissions. In other words, if there are two messages m_1 and m_2 with the same recipient, such that there exists a causal path from m_1 to m_2, then m_1 must be received before m_2. This notion of causal ordering was first introduced by Lamport under the name "happened-before" relation. In Figure 1.3, this causality is violated: m_1 should be received before m_3. Causal delivery is commonly implemented using Lamport's logical clock algorithm [23].

Mailbox In this model, any two messages sent to the same process, regardless of the sender, must be received in the same order as they are sent. If a process receives m_1 before m_2, then m_1 must have been sent before m_2. `mb` coordinates all the senders of a single receiver. This model is also called FIFO $n-1$. In Figure 5.2, an example for this communication model is shown.

RSC Figure 5.1 shows `rsc` as the last block of the hierarchy. `rsc` stands for *Realisable in Synchronous Communication*, therefore, is comparable to our definition of synch model, but there are some differences. For example, it does not accept *orphan messages*, which are instead accepted for the definition of this thesis.

5.2 Realisability for Alur

In this section, we compare our framework with one of the earliest and most influential works on realisability, the one of Alur et al. [3], which also inspired part of our approach. Their notion of *Weak Realisability* captures the idea that a specification

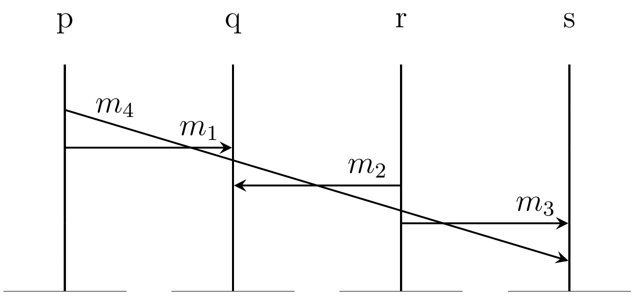

Figure 5.2: An example of mailbox semantic.

of Message Sequence Charts (MSCs) should already include all behaviours that are consistent with the local views of processes. Intuitively, a set of MSCs is weakly realisable when, for every process, the events it observes in any MSC of the specification are compatible with those in some MSC already in the set. This closure condition ensures that the global behaviour can be reconstructed from the projections of individual processes, so that every implied MSC is already part of the language. Our own definition of weak realisability coincides with theirs, as it expresses the same fidelity concept over the local behaviour and abstracts from any deadlock-related concern. In both cases, weak realisability focuses on the alignment between local and global behaviours rather than on safety properties such as deadlock-freedom. For safe realisability, we recall an informal definition of Alur et al. [3] and discuss the differences.

Intuitively, let L be a set of MSCs. Then L is said to be *safely realisable* if there exists a family of concurrent automata $\langle A_i \mid 1 \leq i \leq n \rangle$ such that $L = L(\prod_i A_i)$ and the product automaton $\prod_i A_i$ is *deadlock-free.* In this setting, a *deadlock state* is a configuration of the global system from which no accepting state can be reached. This corresponds to a situation where all processes are waiting to receive messages that are no longer available in their communication buffers, preventing further progress. Hence, a system is deadlock-free if no such state is reachable from its initial configuration. This notion captures the safety aspect of realisability by ensuring that the system never reaches a globally stalled state during execution. This definition of safe realisability correspond to ours in $\mathsf{p2p}$ or synch.

The work of Alur et al. went on further, defining specific complexity classes for different kind of assumptions. For finite sets of MSCs, weakly realisability is shown to be $\mathtt{coNP}$-complete and safe realisability is shown to be decidable in $\mathtt{P}$-time. The problem was subsequently studied for HMSCs. For *bounded* HMSCs, safe realisability remains decidable, and it is `EXPSPACE`-cpmplete, but weak realisability becomes undecidable. For *unbounded* HMSCs, safe realisability and weak realisability are undecidable remains decidable, and it is `EXPSPACE`-cpmplete, but weak realisability becomes undecidable [3]. Later, Lohrey et al. [27] proved, with a technique that involves five processes, that in the general case, safe realisability is undecidable, though it is decidable (and `EXPSPACE`-complete) for a specific kind of HMSCs, called globally cooperative. Most positive results assume bounded channels, but [6] introduces a new class of HMSCs that allows unbounded channels while maintaining realisability. A summary of the main complexity results is given in Table 5.1.

	Finite set	**Bounded graphs**	**Unbounded**
Weak	`coNp`-complete	undecidable	undecidable
Safe	P-time	`EXPSPACE`-complete	undecidable

Table 5.1: Summary of results on realisability in [3].

5.3 Multiparty Session Types

Multiparty Session Types (MPST) [20] provide a type-theoretic framework to specify and verify communication protocols among multiple participants. They ensure that communication follows a predefined structure, preventing errors such as deadlocks, orphan messages, and unspecified receptions. The **global specification** describes the overall communication protocol. From this, one derives the **local behaviours** of each participant via a *projection* operation. The system's **processes** form the *implementation*, defining how participants interact. With the definition of a *typing system* and suitable *type-checking rules*, one ensures that the implementation conforms to the local specification, thereby guaranteeing properties such as *well-formedness*. Figure 5.3 show a schema summarizing the principal parts of the framework.

5.3.1 Projectability

A central notion in MPST is *projectability*, which asks whether a global type can be faithfully projected into local specifications for each participant. If projection succeeds, the resulting local types interact without mismatches or unintended behaviours, effectively bridging global specifications and distributed implementations [20]. Projectability is, therefore, comparable to the realisability problem as they have the same aim. Projection algorithms, however, often reject natural protocols that fail to meet restrictive syntactic conditions. This difference between expressivity and safety has motivated extensions of the theory, with [9] being the only algorithm aiming for full completeness.

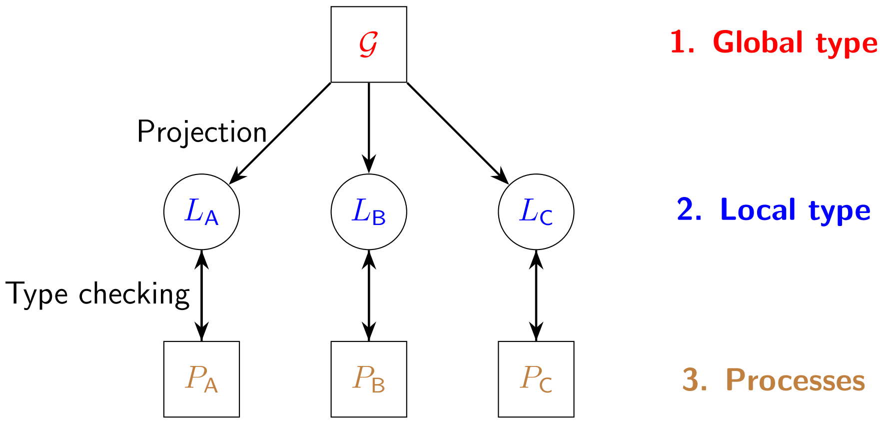

Figure 5.3: Intuitive schema of MPST framework

A key restriction in the definition of MPST appears in branching. In the original framework [20, 8], choice is **sender-driven**: the first sender dictates the branch, ensuring safety but excluding many common patterns where multiple participants influence the decision [8]. Allowing **mixed choice** increases expressivity by permitting several initiators, but it also makes the realisability problem undecidable in general [30].

5.3.2 Realisability for MPST

Recent work has focused on addressing the connection between MPST and automata-theoretic formalisms. Stutz and Zufferey showed that realisability is decidable by encoding global types into HMSCs that are globally cooperative [32, 29]. Building on this, Li et al. [26] proposed a complete projection function for MPST, guaranteeing that every implementable global type admits a correct distributed implementation.

Stutz's thesis [30] connects MPST to High-level MSCs (HMSCs), introducing a generalized projection operator for sender-driven choice where a sender may branch towards different receivers. This captures patterns beyond classical MPST projection. He also proves that while syntactic projection is incomplete, an automata-theoretic encoding into HMSCs yields decidability for sender-driven choice, with realisability shown to be in `PSPACE`-the first precise complexity bound for this fragment.

We recall the definition of the *Realisability Problem* as introduced by Stutz et al. [30].

Definition 5.3.1 (Implementability Problem [30])**.** A language $L \subseteq \Gamma^\omega$ is said to be *implementable* if there exists a CSM $\{A_p\}_{p\in\mathbb{P}}$ such that

- **deadlock freedom:** $\{A_p\}_{p\in\mathbb{P}}$ is deadlock-free, and

- **protocol fidelity:** L is the language of $\{A_p\}_{p\in\mathbb{P}}$.

where the definition of deadlock freedom is:

Definition 5.3.2 (Deadlock freedom [30])**.** A configuration for an automaton is a deadlock if it is not final and has no outgoing transitions while it is reachable if it occurs on some run of A. A CSM is deadlock-free if no reachable configuration is a deadlock. We say a CSM is sink-final if all its state machines are.

5.4 Choreographies

Choreographies [28] are another formalism to describe distributed communication protocols. Choreographies emphasize the global specification of interactions as a high-level description of the intended message exchanges. Similarly to MPST, their goal is to ensure that a distributed implementation can be derived in which each participant follows a local behaviour consistent with the global description, called respectively *local* and *global*-view. This setting naturally connects to the realisability problem, since the key question is whether a choreography can be faithfully implemented by a system of local processes. In choreographies, the local-view is called **End-Point Projection** (EPP), and it is derived via a projection operation from the global-view. In particular, *Choreography Automata* [4] share many conceptual similarities with our notion of Global Types. Both formalisms model global interaction structures through automata over communication actions, capturing the causal dependencies among participants. The main difference lies in the underlying semantics and the intended use: Choreography Automata focus on synthesis and verification within choreographic frameworks, while our Global Types are tailored to the study of realisability under different communication semantics.

One important challenge studied in choreographic design is the **knowledge of choice** problem, introduced in the context of MPST [9]. This problem can be seen as a specific instance of the general projection problem: it arises when translating a global description into consistent local behaviours. In particular, it captures the difficulty of maintaining coherence when decisions made by one participant must be known by others.

Informally, a choreography has knowledge of choice if, whenever a branching (conditional) decision is made by one participant, all other affected participants are made aware of that decision. Without proper communication of the choice, a participant may behave inconsistently because it lacks information to distinguish which branch was taken. For example, if process A chooses between two branches that lead to different sub-protocols with process B, then B must receive a signal (a "selection") that lets it synchronize on the correct continuation.

If the choreography lacks such a mechanism, it becomes *unprojectable*: EPP cannot generate local behaviours that correctly coordinate the branching. This issue is addressed, typically, by adding explicit selection messages to propagate the choice, and how this can be automated via *amendment* or *repair* algorithms [24, 5], which insert minimal extra communications to guarantee knowledge of choice.

Conceptually, this problem is closely related to the *sender-driven choice* policy highlighted before in the MPST framework, where multiple senders make independent choices that must be reconciled to ensure a coherent global behaviour. In both cases, the challenge lies in ensuring that all participants have sufficient knowledge to follow the same branch, preserving consistency across local projections.

5.5 Other works

Stutz et al. [31] introduce *Protocol State Machines* (PSMs), an automata-based formalism that generalises both MPST and HMSC. Their approach bears resemblance to choreography-automata models, specifying interactions via state machines over communication labels. Crucially, they investigate the computational complexity of type checking and realisability: when restricted to choice-free or single-choice fragments, checking realisability or well-formedness is decidable (often in PTIME), but once *mixed choice* is permitted, the realisability problem becomes undecidable. This result lends further evidence to the hypothesis that unrestricted mixed-choice global types are intrinsically hard to implement in distributed systems.

Another relevant contribution is by Guanciale et al. [17], who study the *realisability of pomsets* via communicating automata. Pomsets, or partially ordered multisets, generalise MSCs by capturing causal dependencies among events rather than total orders. Their work defines realisability conditions ensuring both communication correctness and termination soundness, supporting participants with internal concurrency.

Chapter 6

Conclusion

This work addressed the *realisability problem* for Global Types, a central concern in the verification of distributed systems. After a brief overview of the problem, we positioned our contribution within an ongoing research effort, bridging well-established theoretical foundations with practical tool development.

On the theoretical side, we introduced the necessary background notions (i.e. CFSMs, Global Types, MSCs, and communication models) and formalised weak realisability. The main contribution was to connect the realisability problem to classical undecidability results, in particular through a reduction to the Relaxed Post Correspondence Problem (RPCP). This result, presented in Chapter 3, establishes a fundamental limitation of realisability under synchronous semantics and lays the groundwork for exploring decidable subclasses and practical approximations.

On the practical side, detailed in Chapter 4, we improved and extended the ReSCu tool, used for checking realisability and other semantic properties of Symbolic Communicating Machines (SCMs). The input grammar was refined for greater usability, and new verification routines were implemented, including checks for progress and deadlock-freedom. The tool now also generates visual representations of synchronous systems, along with illustrative examples. These extensions strengthen ReSCu both as a research prototype and as a practical aid for automated verification.ù

In Chapter 5, we analysed part of the state of the art on realisability, comparing our definitions and results with existing approaches in the literature. This analysis helped clarify how our formalisation of weak and safe realisability fits within, and extends, previous frameworks, particularly those by Alur et al. [3], Lohrey et al. [27], and Stutz et al. [30]. The comparison also highlighted key conceptual differences, especially in how communication semantics and closure properties are handled, further motivating our formal treatment.

6.1 Future Work

Future research directions include extending the theoretical results beyond weak realisability toward a decidability result for *safe realisability*, thereby incorporating properties such as deadlock-freedom directly into the analysis of Global Types. This line of investigation will build upon the techniques developed in this work and extend the existing results of Lohrey et al. [27].

Another important direction is the exploration of alternative communication semantics, such as causal-order and mailbox-based models, which were mentioned throughout this work. Investigating the realisability problem under these semantics could shed light on how communication constraints and buffering behaviours affect implementability and decidability. Establishing precise connections between synchronous semantics and these more general models may also lead to new transfer results, showing under which conditions realisability in one model implies realisability in another.

On the practical side, a natural objective is to further enhance ReSCu to support these theoretical extensions, ultimately aiming for a complete and automated framework to decide realisability for restricted classes of Global Types. This would enable systematic benchmarking against existing tools and the validation of the approach on real-world communication protocols.

Finally, it would be valuable to investigate the role of *fairness* and other semantic variants, within the verification process. These extensions could help model more realistic distributed environments and provide a deeper understanding of how fairness assumptions influence the decidability and correctness of realisable systems.

Bibliography

[1] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In *Proceedings of the 22nd international conference on Software engineering*, pages 304–313, 2000.

[2] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623, 2003.

[3] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of msc graphs. *Theoretical Computer Science*, 331(1):97–114, 2005.

[4] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In *International Conference on Coordination Languages and Models*, pages 86–106. Springer, 2020.

[5] Samik Basu and Tevfik Bultan. Automated choreography repair. In *International Conference on Fundamental Approaches to Software Engineering*, pages 13–30. Springer, 2016.

[6] Benedikt Bollig, Marie Fortin, and Paul Gastin. High-level message sequence charts: Satisfiability and realizability revisited. In *International Conference on Applications and Theory of Petri Nets and Concurrency*, pages 109–129. Springer, 2025.

[7] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, apr 1983.

[8] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(2):1–78, 2012.

[9] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8, 2012.

[10] Loïc Desgeorges and Loïc Germerie Guizouarn. Rsc to the rescu: Automated verification of systems of communicating automata. In *International Conference on Coordination Languages and Models*, pages 135–143. Springer, 2023.

[11] Loïc Desgeorges and Loïc Germerie Guizouarn. The original ReSCu repository. `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://src.koda.cnrs.fr/loic.germerie.guizouarn/rescu`, 2025. [Online; accessed 20-August-2025].

[12] Cinzia Di Giusto, Davide Ferré, Laetitia Laversa, and Etienne Lozes. A partial order view of message-passing communication models. *Proceedings of the ACM on Programming Languages*, 7(POPL):1601–1627, 2023.

[13] Cinzia Di Giusto, Loïc Germerie Guizouarn, and Etienne Lozes. Multiparty half-duplex systems and synchronous communications. *Journal of Logical and Algebraic Methods in Programming*, 131:100843, 2023.

[14] Cinzia Di Giusto, Etienne Lozes, and Pascal Urso. Realisability and complementability of multiparty session types. *arXiv preprint arXiv:2507.17354*, 2025.

[15] Belot Florent. Drawing and Analyzing an MSC. `https://belotflorent.github.io/MSC-Tool-SWI-Prolog/`, 2024. [Online; accessed 15-August-2025].

[16] Loïc Germerie Guizouarn and Gabriele Genovese. The updated ReSCu repository. `https://github.com/gabrielegenovese/rescu`, 2025. [Online; accessed 20-August-2025].

[17] Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.

[18] Loïc Germerie Guizouarn. *Communicating automata and quasi-synchronous communications*. PhD thesis, Université Côte d'Azur, 2023.

[19] Alexander Heußner, Tristan Le Gall, and Grégoire Sutre. Mcscm: a general framework for the verification of communicating machines. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 478–484. Springer, 2012.

[20] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008.

[21] International Telecommunication Union. Z.120: Message Sequence Chart. Technical report, International Telecommunication Union, 1992.

[22] International Telecommunication Union. Z.120: Message Sequence Chart. Technical report, International Telecommunication Union, 1996.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. Communications of the ACM, 2019.

[24] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013*, volume 123 of *EPTCS*, pages 34–48, 2013.

[25] Tristan Le Gall. *Abstract lattices for the verification of systemes with stacks and queues*. PhD thesis, Université Rennes 1, 2008.

[26] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. Complete multiparty session type projection with automata. In *International Conference on Computer Aided Verification*, pages 350–373. Springer, 2023.

[27] Markus Lohrey. Realizability of high-level message sequence charts: closing the gaps. *Theoretical Computer Science*, 309(1-3):529–554, 2003.

[28] Fabrizio Montesi. *Choreographic programming*. IT-Universitetet i København, 2014.

[29] Felix Stutz. Asynchronous multiparty session type implementability is decidable - lessons learned from message sequence charts. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 32:1–32:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[30] Felix Stutz. *Implementability of Asynchronous Communication Protocols–The Power of Choice*. PhD thesis, Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau, 2024.

[31] Felix Stutz and Emanuele D'Osualdo. An automata-theoretic basis for specification and type checking of multiparty protocols. In *European Symposium on Programming*, pages 314–346. Springer, 2025.

[32] Felix Stutz and Damien Zufferey. Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. In Pierre Ganty and Dario Della Monica, editors, *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal*

Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022, volume 370 of *EPTCS*, pages 194–212, 2022.

Acknowledgments

Thanks for…