ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Informatica

On the Implementability of Global Types

Relatore:
Prof. Ivan Lanese

Presentata da:
Gabriele Genovese

Correlatori:
Prof. Cinzia Di Giusto
Prof. Étienne Lozes

Contro-relatore:
Luca Padovani

Sessione II Ottobre 2025
Anno Accademico 2024/2025

Qualcosa

Abstract

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 implementability 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 implementability 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 Tables

List of Figures

Listings

Chapter 1

Introduction

Informally, a *distributed system* is a collection of independent computing entities (interchangeably called processes, actors, nodes, or participants) 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 difficulties that computer scientists have long sought to overcome-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**.

One recurring difficulty is writing **correct programs** in this context. 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 developers with guarantees about their programs. Formal methods for distributed systems offer mathematically rigorous techniques to specify, design, and verify such systems. They are valuable during development, helping detect errors early, and during analysis, enabling the study of critical properties such as **safety**, **liveness**, and **deadlock-freedom**. Two primary verification approaches are *model checking* and *by-construction* verification. Model checking systematically explores a system's

state space to confirm properties, while by-construction verification guarantees correctness through the design process itself, preventing errors from being introduced.

There exists several models to reason about distributed systems. Different model are specialized in different aspects of a system, and we are interested in the ones about the exchange of information, such as Calculus of Communicating Systems (CCS), the π-calculus, and Petri nets. In this work, however, we focus on *Multiparty Session Types* (MPST) [17] and *choreographies* [24], since these formalisms place particular emphasis on structured and verifiable communication protocols, making them especially well suited for protocol design. In MPST, communication is specified by a *global type*, which describes the entire interaction among participants. This global type is then *projected* into *local types*, one for each participant. Local types serve as contracts that guarantee each component is compliant to the described protocol, therefore ensuring certain properties, such as deadlock-freedom, at compile time. The implementability problem in MPST is comparable to verifying whether a given global type can be correctly projected into local types, preserving the intended behaviour.

1.1 Goal

The goal of this work is to study the **implementability problem**, which concerns whether a global specification can be faithfully realised by a set of *local processes* in a distributed system. In essence, it asks: does an implementation really **respect** the behaviour described by a given specification model?

To illustrate the relevance of this problem, consider the following example.
Example 1. Given four processes A, B, C, D distributed over a network, and four messages x, y, z, w to be exchanged according to the description in Listing 1.1, is it possible to implement it in a real world 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, only some 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.

This problem is examined from a theoretical perspective to provide a more formal and precise understanding of the fundamental limits that exist and why syntactical constrains of certain models work. In this work, we use an *automata-based* approach to Global Types. This formalism is designed to be highly modular, incorporating various *network semantics* (such as asynchronous, peer-to-peer, causal ordering, and synchronous semantics) as explicit parameters of the framework. This parameterization allows flexible analysis of different communication models within a unified setting.

The main contributions of this work are:

- a proof of the **undecidability** of the *weak implementability* problem under the synchronous semantics of our framework;
- an extension and improvement of the model-checking tool ReSCu [14], enabling the verification of *deadlock-freedom* and *progress* for synchronous systems;
- a related-work overview, highlighting existing research and results in this particular domain, providing a structured summary and comparison with our approach, and setting the perspective for the contributions that follow.

These two contributions are closely connected: they both address the implementability problem, but from two complementary angles. The first contribution establishes undecidability, showing that in the general case the weak implementability 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 is designed as 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. Chapter 1 (the present chapter) gives a high-level description of the frameworks used, avoiding formal definitions and proofs for accessibility. 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.

The *implementability problem* was originally introduced for languages of Message Sequence Charts (MSCs). Before presenting it, we first recall the necessary informal background on MSCs.

1.1.1 Message Sequence Charts

Message Sequence Charts (MSCs) are a standardised graphical formalism, introduced in 1992 [18], 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. An extension of this formalism, known as High-Level Message Sequence Charts (HMSCs), was later introduced [19]. 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.

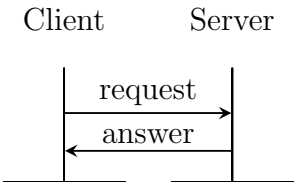

Figure 1.1: Simple example of a client-server architecture.

With MSCs, [10] presents some interesting communication semantics. I will describe a few them informally, using examples to highlight the differences from the main semantics considered in this work, which is `synch`, that is also the only one formally defined in Definition 2.2.3. In Chapter 5, the discussion continue presenting other communication semantics, and summarizing the relevance of the work by Di Giusto, et al. [10]. Some examples are shown in Figure 1.2, 1.3 and 1.4, whose *membership* can be verified with an online tool for MSCs [13].

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.

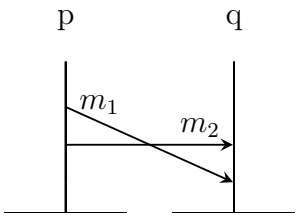

Figure 1.2: Asynchronous semantic example.

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.

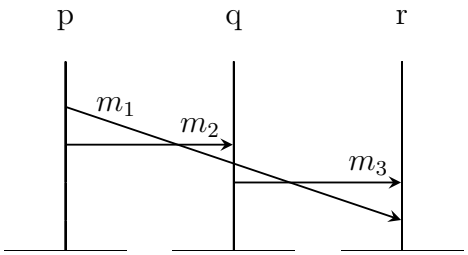

Figure 1.3: Peer-to-peer semantic example.

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 implementability problem for MSCs

The *implementability problem* was first introduced for MSC languages in [1, 2]. It 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 implementability*, requires the implementation to also be **deadlock-free**. This problem has some synonyms in the term, with slightly different definition, but it can be called also realisability and projectability.

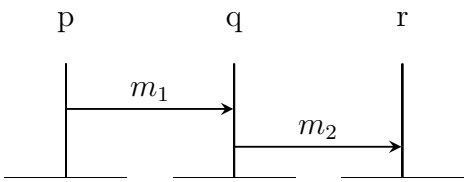

Figure 1.4: Synchronous semantic example.

As already mentioned, the implementability problem in MPST is analogous to checking whether a given global type can be soundly projected into local types while preserving the intended behaviour.

1.1.2 Multiparty Session Types

Multiparty Session Types (MPST) [17] 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 1.5 show a schema summarizing the principal parts of the framework.

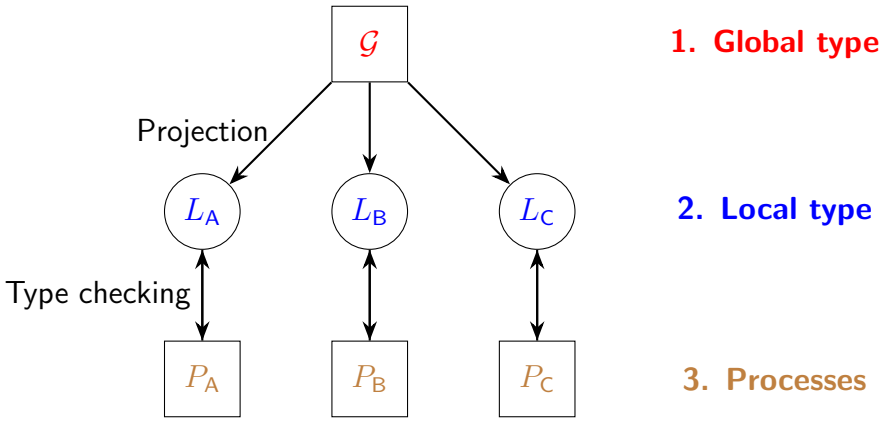

Figure 1.5: Intuitive schema of MPST framework

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 [17]. Projectability is, therefore, comparable to the implementability 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 [7] being the only algorithm aiming for full completeness.

A key restriction in the definition of MPST appears in branching. In the original framework [17, 6], choice is **sender-driven**: the first sender dictates the branch, ensuring safety but excluding many common patterns where multiple participants influence the decision [6]. Allowing **mixed choice** increases expressivity by permitting several initiators, but it also makes the implementability problem undecidable in general [27].

1.1.3 Reduction to synchronous semantic

The main idea of this work is that reasoning about implementability becomes more tractable under *synchronous* semantics for automata-based solutions to the implementability problem. In synchronous communication, send and receive actions are tightly coupled, effectively removing nondeterminism caused by asynchronous message buffering. Several results exploit this observation by reducing the implementability problem under richer communication models (e.g. asynchronous or peer-to-peer FIFO) to the simpler synchronous case [3, 10].

Formally, one can show that if a global type is implementable in synchronous semantics, then under certain conditions it is also implementable in more general models such as peer-to-peer or mailbox semantics. This reduction requires constraints such as *orphan-freedom* (no message is left unmatched) and *deadlock-freedom*.

The following theorem, currently a work in progress by my supervisors [12, Theorem 5.3], provides a characterization of a connection between peer-to-peer semantics and synchronous semantics: a global type G is deadlock-free realisable in p2p iff the following four conditions hold

- the language of G's local type is in 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 implementable 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 implementability checks, complementing the undecidability findings of the theoretical contribution.

Chapter 2

Preliminaries

In this section, the fundamental concepts and definitions necessary to contextualize the main contributions of this work are presented. I, 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

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 two communication models: the peer-to-peer model ($\mathsf{p2p}$) and 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 $\mathsf{src}(s+1) = s$.

In the peer-to-peer communication model $\mathsf{p2p}$, every pair of processes (p, q) is connected by a dedicated FIFO channel. Messages sent by p to q are delivered in the same order in which they were issued: if p first sends m_1 and then m_2, the channel guarantees that m_2 cannot overtake m_1. Concretely:

- if m_1 is never received, then m_2 cannot be received either;
- if both are received, then m_1 is delivered before m_2.

Definition 2.2.4 (Peer-to-peer model $\mathsf{p2p}$)**.** The set $\mathcal{E}_{\mathsf{p2p}}$ consists of all executions e such that for any two send events $s_1 = !m_1^{p \to q}$ and $s_2 = !m_2^{p \to q}$ in $\mathsf{events}_S(e)$, with $s_1 < s_2$, one of the following holds:

- s_2 remains unmatched; or
- there exist receive events r_1, r_2 such that $r_1 < r_2$, $\mathsf{src}(r_1) = s_1$, and $\mathsf{src}(r_2) = s_2$.

It is easy to observe that $\mathcal{E}_{\mathsf{synch}} \subseteq \mathcal{E}_{\mathsf{p2p}}$. Furthermore, the *source function* src_e is defined as follows. If e is an execution in $\mathsf{p2p}$, the source of the i-th receive event of q from p is the i-th corresponding send event of p to q. If instead e belongs to 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. 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 $\mathsf{src}((r,1)) = (q,3)$.

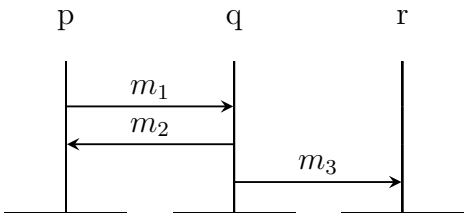

Figure 2.1: Simple example with an exchange of three messages.

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 $M_1 = ((w_p^1)_{p\in\mathbb{P}}, \mathsf{src}_1)$ and $M_2 = ((w_p^2)_{p\in\mathbb{P}}, \mathsf{src}_2)$.

Then: (i) for each process p, the sequence is $w_p = w_p^1 \cdot w_p^2$; (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'
- 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 3. Consider the Example 2. 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 4. Considering the Example 2, 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 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 5. Consider the Example 2 and the respective linearisation listed in Example 4. 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 [5]. These definitions will later be used also for Chapter 4.

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 a model)**.** 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 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.

All synchronous executions are orphan-free by definition.

A system is deadlock-free if, any *partial* execution can be extended/completed to an accepting execution.

Definition 2.2.12 (Deadlock-free)**.** 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' with $e \leq_{\mathsf{pref}} e'$ and $e' \in \mathcal{L}^{\mathsf{com}}_{\mathsf{exec}}(\mathcal{S})$.

The following result shows that, for either $\mathsf{p2p}$ or synch communication models, the deadlock-freedom property of a system of CFSMs can be expressed as a property on the MSCs of the system.

Lemma 1 (Deadlock-freedom as an MSC property)**.** *Assume* com *is* $\mathsf{p2p}$ *(respectively,* com *is* synch*). Then a system of CFSMs* $\mathcal{S}$ *is deadlock-free for* com *if and only if*

$$\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\widehat{\mathcal{S}}) \subseteq \mathsf{Prefixes}(\mathcal{L}^{\mathsf{com}}_{\mathsf{msc}}(\mathcal{S})).$$

The proof for Lemma 1 can be found in [12, Proposition 2.1].

2.3 Global Types

This part will further highlight the basic notions to understand the formal proof for the theorem presented in Chapter 3 and, in particular, Global Type and Weakly-realisable. We begin by extending the definition of linearisability so that it applies to all communication models. In our setting, Global Types are automata that describe a language of MSCs, as considered in this recent work by Di Giusto, et al. [12].

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 write Arr to denote the finite set of arrows. A Global Type G is a deterministic finite state automaton over the alphabet Arr.

We use the notation $p \overset{m}{\leftrightarrow} q$ 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 will serve as an acknowledgment message for p.

Example 6. An example of a Global Type expressed as an automaton is the following. Consider the not-implementable specification stated in Listing 1.1. The protocol can be modelled with the Global Type in Figure 2.2.

I 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. Note that for $w \in \mathsf{Arr}^*$, the function $w \mapsto \mathsf{msc}(w)$ with $\mathsf{msc}(w) \in \mathcal{M}_{\mathsf{synch}}$ is not

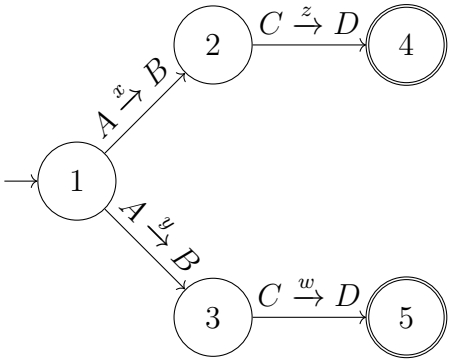

Figure 2.2: An automaton representing the specification's global type given in Listing 1.1.

injective, as two arrows with disjoint pairs of processes commute. We write $w_1 \sim w_2$ if $\mathsf{msc}(w_1) = \mathsf{msc}(w_2)$, and $[w]$ for the equivalence class of w with respect to $\sim$.

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 $\mathcal{L}_{\mathsf{words}}(\mathsf{G})$:

Definition 2.3.2 ($\mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$)**.**

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

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

$$\mathcal{L}^{\forall}_{\mathsf{msc}}(\mathsf{G}) \stackrel{\mathsf{def}}{=} \{\mathsf{msc}(w) \mid [w] \subseteq \mathcal{L}_{\mathsf{words}}(\mathsf{G})\}.$$

Definition 2.3.4 (Commutation-closed)**.** A global type G is *commutation-closed* if

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

I now introduce the definition of implementability following the one given by Alur, et al. [3], referred to as *Weak-realisability*. To formalize it, we first define the notions of *weak implication* and *weak closure*.

Definition 2.3.5 (Weakly-imply)**.** Let $\mathcal{M}$ be a set of MSCs and M an MSC. $\mathcal{M}$ *weakly implies* M, if for any sequence of automata $\langle A_i \mid 1 \leq i \leq n \rangle$, if every MSC in $\mathcal{M}$ is in $L(\prod_i A_i)$ then $M \in L(\prod_i A_i)$.

In order to understand the meaning of *weak implication*, consider the following example.

Example 7. Define two MSCs, MSC1 and MSC2. Both perform the same four communications, but in different orders. MSC1 first sends message a from P1 to P2, then from P1 to P3, then sends b from P4 to P2, and finally from P4 to P3. MSC2 instead starts with P4 sending b to P2, then to P3, followed by P1 sending a to P2 and then to P3.

Now define a third MSC M with the same four messages but in a different order: P1 sends a to P2, P4 sends b to P2, P4 sends b to P3, and finally P1 sends a to P3.

We say M is weakly implied by MSC1 and MSC2. Indeed, by looking at each process projection we recover the same behaviour in one of them: for P1 and P4 in both, for P2 in MSC1, and for P3 in MSC2. Figure 2.3 illustrates the three MSCs.

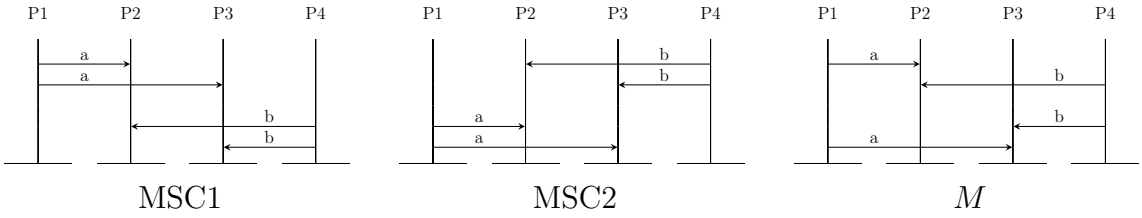

Figure 2.3: The MSC M is weakly implied by MSC1 and MSC2

Definition 2.3.6 (Weakly-closure $\mathcal{M}^w$)**.** The weak-closure $\mathcal{M}^w$ of a set $\mathcal{M}$ of MSCs contains all the MSCs weakly implied by $\mathcal{M}$.

Definition 2.3.7 (Weakly-realisable)**.** An MSC M is said to be weakly-realisable if the set of MSCs $L(M)$ is weakly realisable. A set of MSCs $\mathcal{M}$ is said to be weakly realisable if $\mathcal{M} = \mathcal{M}^w$.

It is important to note that this definition does not include the property of deadlock-freedom.

We are now ready to present the main contributions of this work.

Chapter 3

Weak-Realisability is Undecidable for Synch Global Types

As already mentioned, the first contribution is Theorem 1, which states that *Weak-realisability is undecidable for synchronous global types*. To understand this result, I have already covered the basic notions in Chapter 2 on MSCs, Global Types, and Weak-realisability. These concepts are general and align closely with definitions used in previously established works. We now introduce the main objects employed in the proof of Theorem 1. The proof itself is adapted from the work of Alur et al. [3]. In between the new definition's declaration, I will also highlight the differences with the original proof.

3.1 Definitions

The proof is carried out by 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 a *global type* (called L^*) built from synchronous MSCs. Therefore, we need to prove:

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

In the original proof, L^* is not a global type, but an MSC. Then, we need to change some base definitions of the original paper to fit the model.

Definition 3.1.1 (Relaxed Post Correspondence Problem)**.** Given a set of tiles $\{(v_1, w_1), (v_2, w_2), ..., (v_r, w_r)\}$, determine 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 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 8 (Simple RPCP instance)**.** Consider the tile set

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

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

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

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 $x_1 \cdots x_j$:
  - $j = 1$: $y_1 = \texttt{a}$ is a prefix of $x_1 = \texttt{ab}$;
  - $j = 2$: $y_1y_2 = \texttt{abb}$ is a prefix of $x_1x_2 = \texttt{abb}$;
  - $j = 3$: $y_1y_2y_3 = \texttt{abbc}$ is a prefix of $x_1x_2x_3 = \texttt{abbc}$.

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

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 $x_i = w_i$;

let's define the behavior of the MSC M_i^n. Firstly, Process 1 synchronously sends the 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,

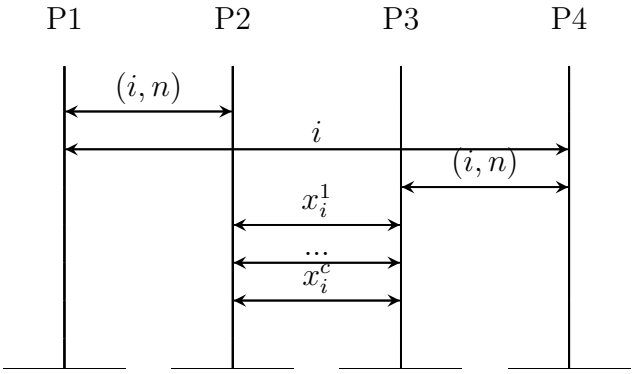

Figure 3.1: The M_i^n MSC.

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 2). 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.

Lemma 2. *The MSC M_i^n belongs to $\mathcal{M}_{\mathsf{synch}}$.*

Proof. By Definition 2.2.8 and Definition 2.2.3, we need to show a linearization with all send operations followed by their corresponding receive operations:

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

Such a linearization exists by construction, hence M_i^n is synchronous. □

Before establishing the connection between MSCs and Global Types, we briefly summarize the rationale behind the design of M_i^n.

Suppose that $\Delta = (i_1, a_1, b_1, \ldots, i_m, a_m, b_m)$ is a solution to the RPCP instance. From this solution we construct two MSC 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 synchronous concatenations of synchronous MSCs. We then define a third MSC $M_{\texttt{sol}}$, 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 $M_{\texttt{sol}}$ is acyclic and *synchronous*. Establishing the synchrony of $M_{\texttt{sol}}$ is non-trivial, and this step is an addition to the original proof. By construction, L^* weakly implies $M_{\texttt{sol}}$, but $M_{\texttt{sol}} \notin L^*$, since at least one tile differs. Consequently, L^* is *not realisable*.

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.

Definition 3.1.3 (G_M)**.** Given an MSC $M \in \mathcal{M}_{\mathsf{synch}}$, there exists a global type G_M such that $M \in \mathcal{L}^{\exists}_{\mathsf{msc}}(\mathsf{G})$.

Definition 3.1.3 establishes a direct correspondence between a single synchronous MSC and a global type. In particular, every synchronous MSC can be captured precisely by a global type whose language contains 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.

Definition 3.1.4 (G_S)**.** Given a string $S \in \Sigma^*$, and two integers i, n, the global type G_S is the global type composed of:

- $\mathbb{P} = \{p, q, r, s\}$;
- $\mathbb{M} = \{m_1, m_2, m_3, m_{S_1}, ..., m_{S_c}\}$ where $m_1 = (i, n), m_2 = i, m_3 = (i, n)$, $m_{S_1} = S_1, ..., m_{S_c} = S_c$ with $c = |S|$;
- Arr $= \{p \overset{m_1}{\longleftrightarrow} q \ , \ p \overset{m_2}{\longleftrightarrow} s \ , \ s \overset{m_3}{\longleftrightarrow} r \ , \ q \overset{m_{S_1}}{\longleftrightarrow} r \ , \ ..., \ q \overset{m_{S_c}}{\longleftrightarrow} r \ \}$ where each arrow denotes a synchronous message accompanied by an acknowledgment.

G_S's automaton is depicted in Figure 3.2.

Intuitively, G_S specifies the same communication pattern as the MSC M_i^n introduced in Definition 3.1.2, with the string S representing the transmitted sequence

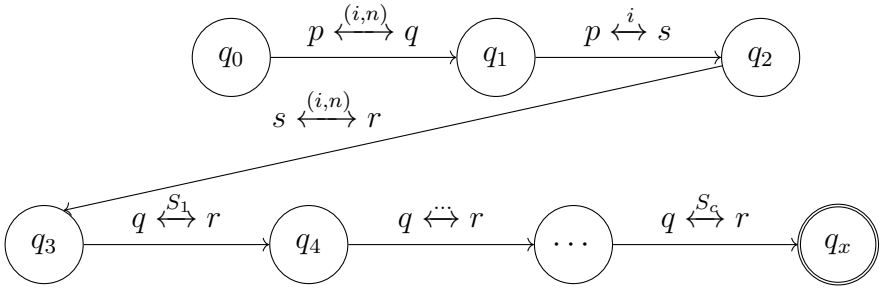

Figure 3.2: The global type G_S.

of characters. This structural correspondence will be made precise in the next lemma (Lemma 3).

Lemma 3. *The MSC M_i^n (Definition 3.1.2) is included in $\mathcal{L}(G_S)$ (Definition 3.1.4).*

Proof. Both M_i^n and G_S describe the same communication structure: process p sends (i, n) to q and i to s; process s relays (i, n) to r; process q then sends the characters of S (here matching x_i) to r. If i, n and S are the same, the sequence of messages is identical in both M_i^n and G_S. Since both models enforce synchronous communication, their linearisations coincide. Hence, $M_i^n \in L(G_S)$. □

Having established the correspondence between an individual MSC and a global type, we can now extend Definition 3.1.3 to encompass sets of MSCs, thereby capturing entire families of synchronous behaviours within a global type. This definition is used to easily construct the global type L^*.

Definition 3.1.5 (G^*)**.** Given a set of MSCs $\mathcal{M} = \{M \mid M \in \mathcal{M}_{\mathsf{synch}}\}$, G^* is the set of global types such that $G^* = \{G_M \mid M \in \mathcal{M}\}$, where G_M is built using Definition 3.1.3.

Informally, for every MSC $M \in \mathcal{M}$ there exists a global type $G \in G^*$ that captures the language of M. Now, let's build the automaton that represents all the possible combinations and constructions of a set of MSCs.

Definition 3.1.6 (The L^* global type)**.** Assume a finite set $\mathcal{M}$ of MSCs, where $\mathcal{M} = \{M \mid M \in \mathcal{M}_{\mathsf{synch}}\}$. Let G^* be defined as in Definition 3.1.5. 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, L^* is obtained as the determinisation of L_N^*.

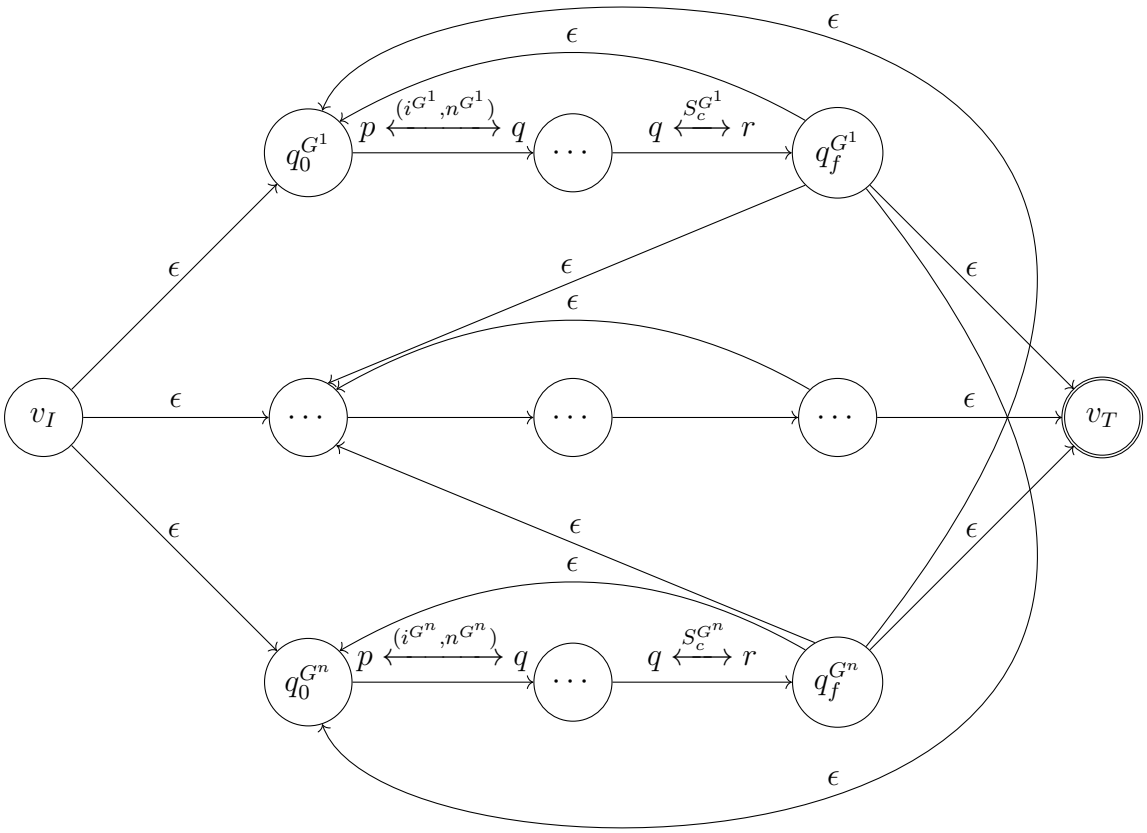

Figure 3.3: The automaton of the global type L_N^*.

Informally, L^* denotes the set of all possible executions arising from a family of MSCs. This construction constitutes the final component required for the proof, as

it provides the key structure used to demonstrate non-realisability.

3.2 Undecidability proof

Given the definitions and lemmas state in the last section, I am 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.*

Proof. The proof proceeds via a reduction from the RPCP problem. Let's define some useful elements for the proof.

Given an instance $\Delta = \{(v_1, w_1), \ldots, (v_m, w_m)\}$ of RPCP, we construct a set L of MSCs over four processes as follows. For each pair (v_i, w_i), we define two MSCs, M_i^0 and M_i^1, as illustrated in Figure 3.1. Observe that the communication graph of each MSC is strongly connected and involves all four processes. Therefore, the MSC represented from L^* and derived from L is bounded. With the set L, and following the Definition 3.1.6, we can construct the global type L^*.

We need to prove:

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

$\Rightarrow$ Let's 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. Consider the new MSCs M_x and M_y obtained from the sequences $M_x = M_{i_1}^{a_1} \cdots M_{i_m}^{a_m}$ and $M_y = M_{i_1}^{b_1} \cdots M_{i_m}^{b_m}$. Executions of both of these (sequences of) MSCs must exist in any realisation of L^*. Additionally, these MSCs are in $\mathcal{M}_{\mathsf{synch}}$ because they are sequence of $\mathcal{M}_{\mathsf{synch}}$ MSC (Lemma 2). 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 $M_x|_1$, $M_x|_2$, $M_x|_3$, and $M_x|_4$ of M_x, and $M_y|_1$, $M_y|_2$, $M_y|_3$, $M_y|_4$ of M_y onto the 4 processes. Now consider an MSC $M_{\mathtt{sol}}$ formed from $M_y|_1$, $M_y|_2$, $M_x|_3$, and $M_x|_4$. This MSC represent 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 weakly implied by L^*. By definition, the only thing to establish is that $M_{\mathtt{sol}}$ is indeed an MSC, in the sense that it is acyclic, well-formed, complete and synchronous. 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 $M_y|_1$ and $M_x|_4$ (i.e., the sequence of messages sent from P_1 to P_4 in $M_y|_1$ is equal to the sequence of messages received in $M_x|_4$), and the communication between P_2 and P_3 is consistent in $M_y|_2$ and $M_x|_3$ 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.

Finally, we prove that $M_{\mathtt{sol}} \in \mathcal{M}_{\mathsf{synch}}$. Assume, for 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 we now that can cause a cycle are the communication between P_1 and P_4, and between P_2 and P_3. We don't need to analyse the new communication between P_1 and P_4 because it's 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.

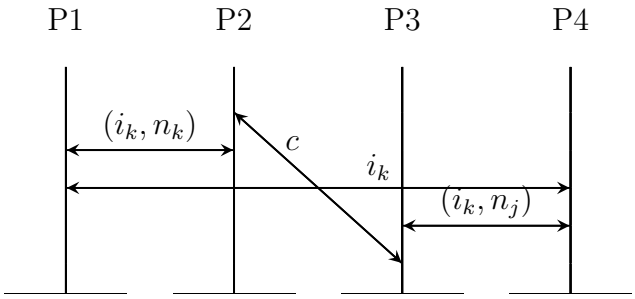

Figure 3.4: MSC communication that breaks synchrony.

For the communication between P_2 and P_3, the only possible cycle pattern is depicted in Fig. 3.4. 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 $M_{\mathtt{sol}} \in \mathcal{M}_{\mathsf{synch}}$.

Note that $M_{\mathtt{sol}}$ cannot itself be in L^* because there must be some index i_j where $a_j \neq b_j$, and no MSC exists in L where, after P_1 announces the index, what P_2 sends is not identical to what P_3 receives.

$\Leftarrow$ Suppose there is some MSC $M^{@}$ which exists in any realisation of L^*, but is not in L^* itself. We want to derive a solution to Δ from $M^{@}$. First, it is clear that the projection $M^{@}|_1$ must consist 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, it is clear that, in order for process 2 to receive those messages, $M^{@}|_2$ must consist 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, $M^{@}|_4$ 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 $M^{@}|_3$ consists of a sequence of receipt of $(i, 0)$ or $(i, 1)$ followed by receipt of v_i or w_i, respectively. Now, since $M^{@}$ is not in 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 $x_{i_1} \cdots x_{i_m}$.

□

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

Example 9 ($M_{\mathtt{sol}}$ Example of Theorem 1)**.** Consider the tiles and the solution of the RPCP instance in Example 8, 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 $x_1x_2x_3 = \mathtt{ab\,b\,c} = \mathtt{abbc}$ and $y_1y_2y_3 = \mathtt{a\,bb\,c} = \mathtt{abbc}$. 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 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 stream. Using the concrete index sequence $(2, 1, 3)$ we form two concatenated MSCs:

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

Here M 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) $(y_1, y_2, y_3) = (v_2, w_1, w_3)$.

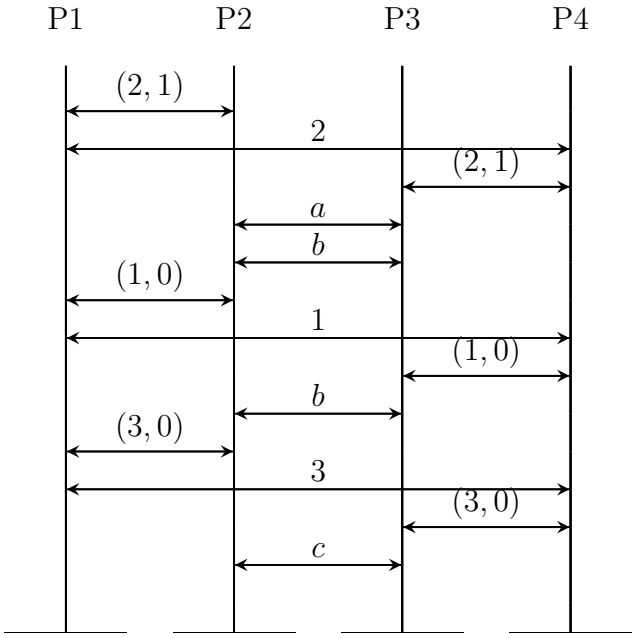

Figure 3.5: The MSC M_x

Recall that $M|_p$ denote the projection of M onto process p. We construct the MSC

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

i.e. processes $1, 2$ follow M_y while $3, 4$ follow M_x. Intuitively, $M_{\texttt{sol}}$ pairs the right-side construction (from M_y) with the left-side construction (from M_x). Figure 3.7

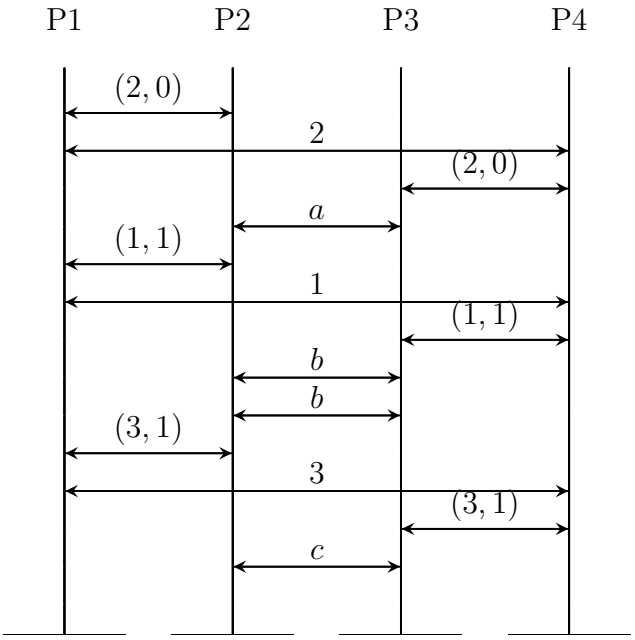

Figure 3.6: The MSC M_y

illustrates the behaviour of the MSC $M_{\texttt{sol}}$. 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.

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.

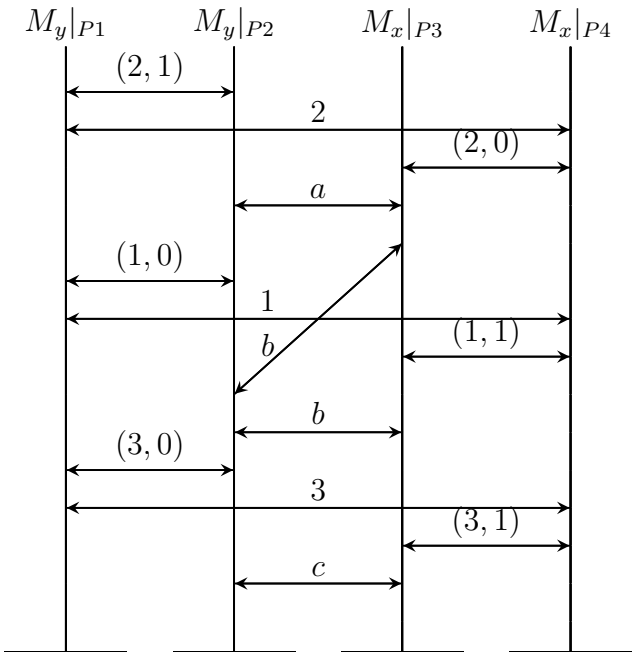

Figure 3.7: The MSC $M_{\texttt{sol}}$

Chapter 4

ReSCu

I now present ReSCu (first introduced in [8, 11, 15]), describing its features, the input language it uses, its implementation, and the modifications I introduced to extend its functionality [9]. The updated public repository with the new features and examples is available at

`https://github.com/gabrielegenovese/rescu` [14].

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 both FIFO and bag buffers. 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 borderline 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 [16], that uses a framework with for different verification techniques. Symbolic Communicating Machines (SCM), defined and used in [21, Definition 5.1] serve as the input format of the tool. SCMs are Communicating Finite-State Machines (CFSM, Definition 2.2.9) extended with a finite set of variables. The grammar has been updated to provide greater flexibility and clar-

ity. 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

Let's make a simple example.

Example 10 (Ping-Pong Example)**.** Let the set of processes be $\mathbb{P} = \{A, B\}$, the set of messages $\mathbb{M} = \{\texttt{ping}, \texttt{pong}\}$, 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

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

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: $0 \xrightarrow{(A,B,!,\texttt{ping})} 1 \xrightarrow{(B,A,?,\texttt{pong})} 2$.

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

with

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

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

```
1  scm ping_pong :
2
3  nb_channels = 1 ;
4  parameters :
5  unit ping ;
6  unit pong ;
7
8  automaton A :
9  initial : 0
10 final : 2
11
12 state 0 :
13 to 1 : 0 ! ping ;
14 state 1 :
15 to 2 : 0 ? pong ;
16 state 2 :
17
18 automaton B :
19 initial : 0
20 final : 2
21
22 state 0 :
23 to 1 : 0 ? ping ;
24 state 1 :
25 to 2 : 0 ! pong ;
26 state 2 :
```

Listing 4.2: Tool's input for Example 10

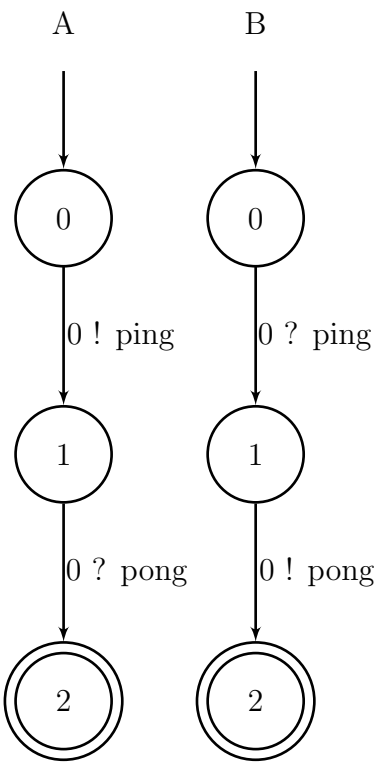

Figure 4.1: Simple Ping-Pong example.

4.2 Progress and Deadlock-Freedom

I 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 whenever the input SCM is recognized as an RSC. These two verification routines are triggered only after another check: once the system is proven to be RSC, we can safely construct a well-formed synchronous product from it. We consider the definitions given in Chapter 2. Definition 2.2.9 corresponds to the concept of an SCM. We now present the definition of the Synchronous Product for CFSMs, which has been implemented in the tool and serves as a key component for analysing.

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 global type $P = \text{prod}_s(\mathcal{S}) = (L, Arr, \delta, l_0, F)$, 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 11 (Synchronous Product Example)**.** Consider the system of CFSMs $\mathcal{S} = (\mathcal{A}_A, \mathcal{A}_B)$ from the Example 10. Its synchronous product is $P = \text{prod}_s(\mathcal{S}) = (L, Arr, \delta, l_0, F)$, 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 $\text{prod}_s(\mathcal{S})$'s automaton.

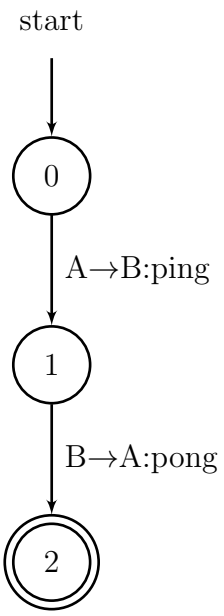

Figure 4.2: Synchronous Product of the CFSM system in Example 11.

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 properties added to the tool.

Let's consider Definition 2.2.12. I will copy the definition here, changing only the semantic of the system. This implies that the system uses the synchronous product (Definition 4.2.1).

Definition 4.2.2 (Deadlock-free)**.** 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 $e' \in \mathcal{L}^{\mathsf{synch}}_{\mathsf{exec}}(\mathcal{S})$.

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 rsc 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 $e \in \mathcal{L}_{\mathsf{exec}}(P)$, or
- there exists another execution $e' \in \mathcal{L}_{\mathsf{exec}}(\widehat{P})$ 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.

Additionally, 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 [14].

4.3 Examples

To illustrate these notions, I 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 12. 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 12

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.

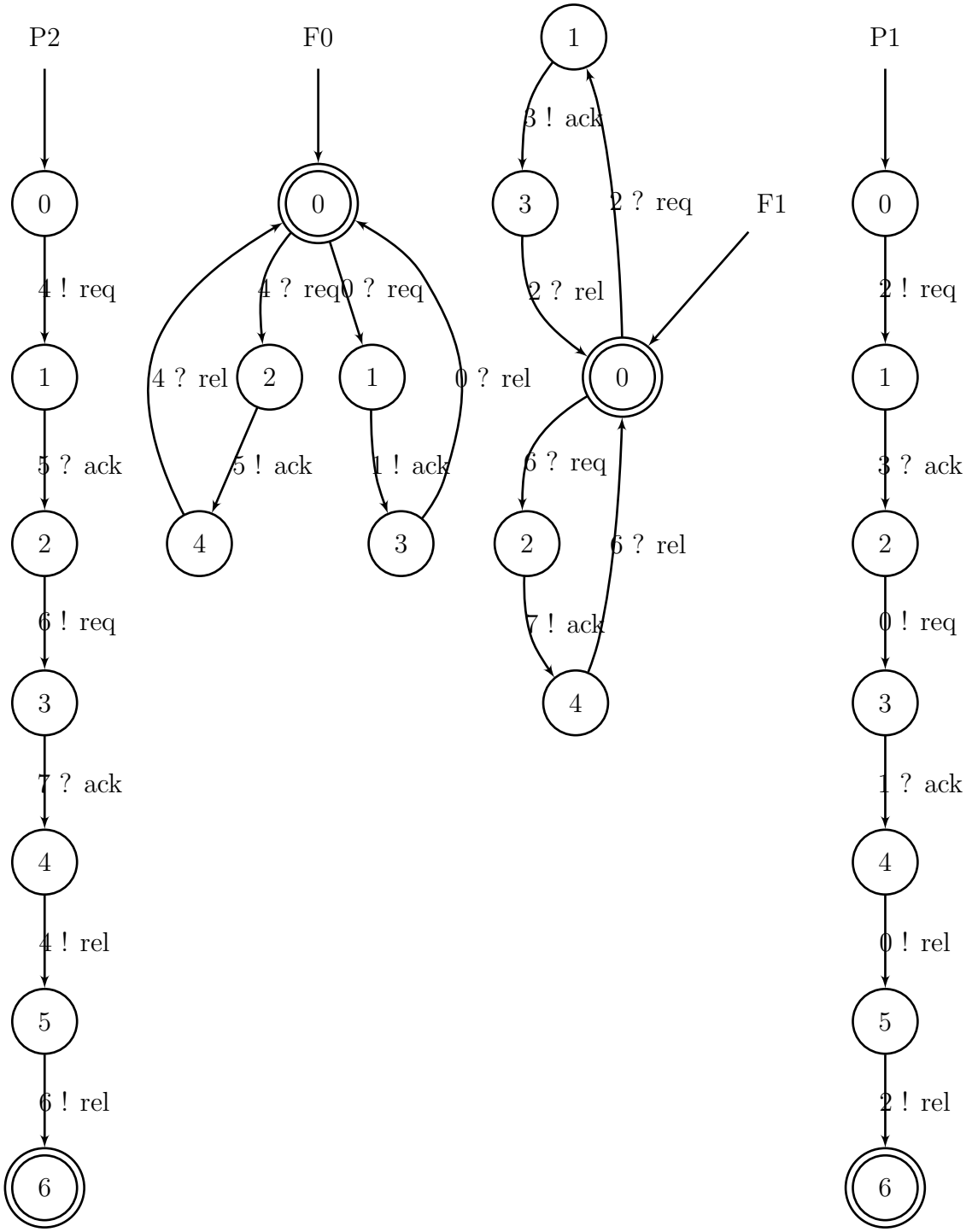

Figure 4.3: SCM automata representation of the Example 12.

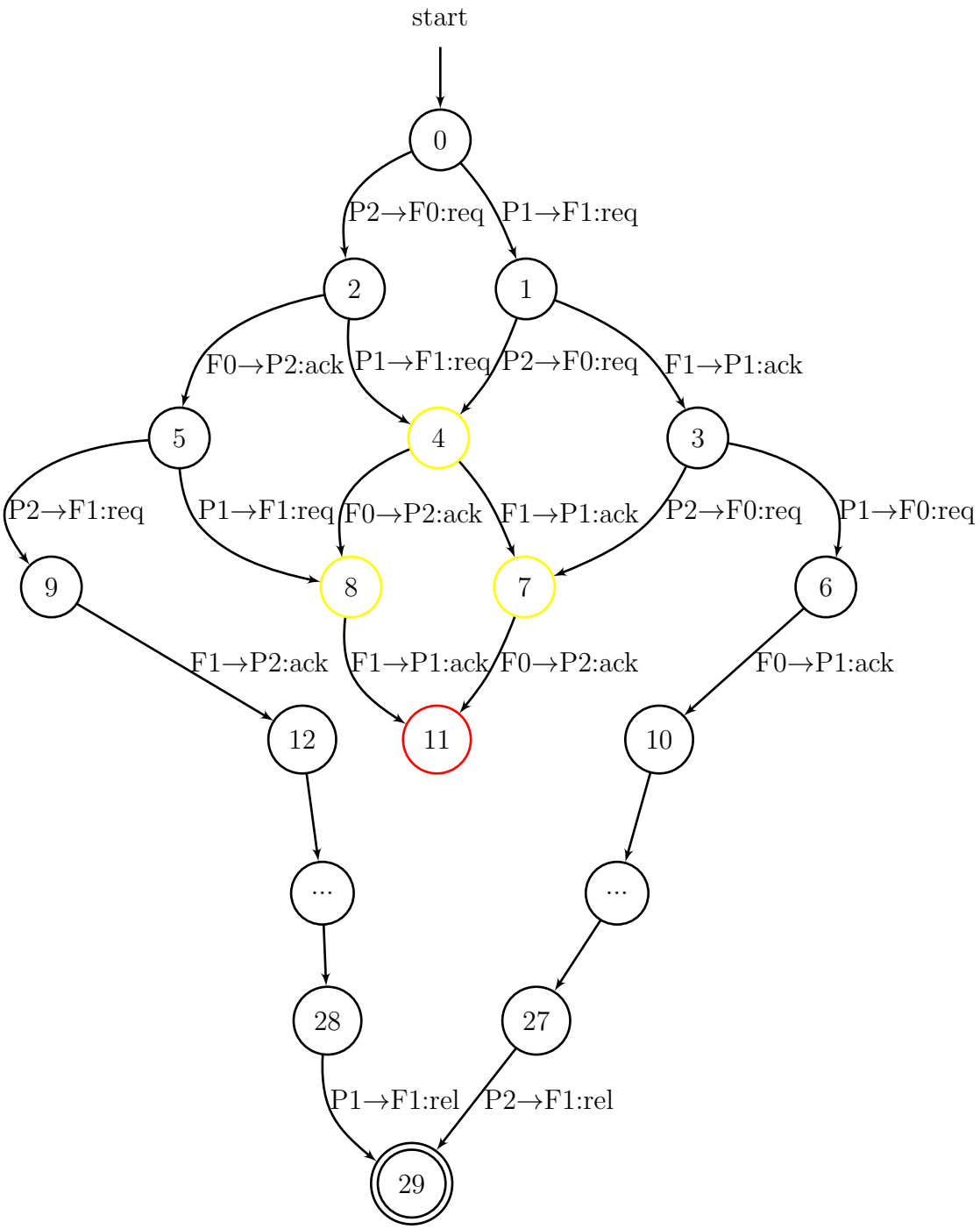

Figure 4.4: Synchronous Product of the Example 12.

4.3.2 Example with a loop

Example 13. 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 13.

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.

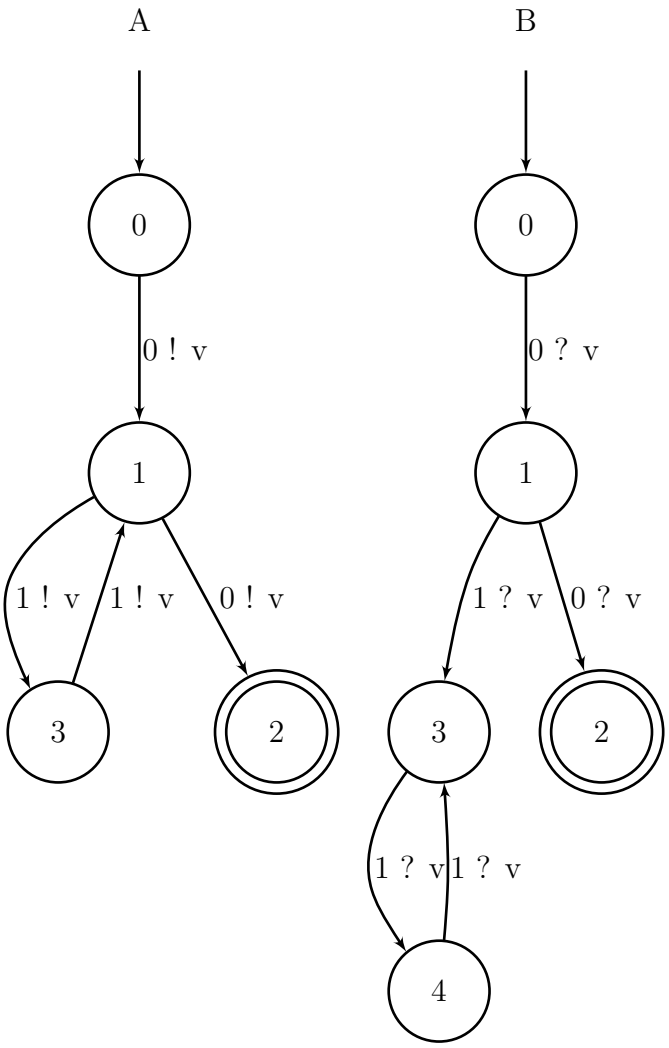

Figure 4.5: SCM automata representation of the Example 13.

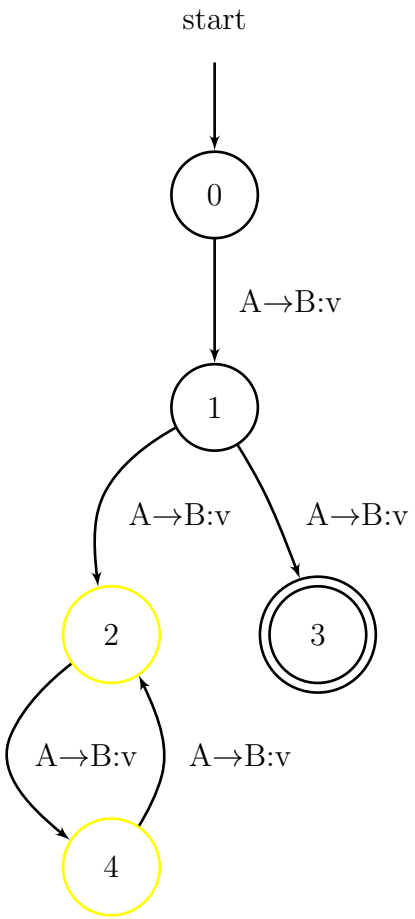

Figure 4.6: Synchronous Product of the Loop Example 13

Chapter 5

Related work

5.1 Hierarchy of communication model's semantics

We defined early some communication semantics of out interest. Furthermore, [10] 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 implementability problem.

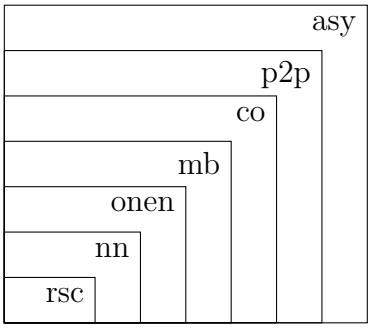

Figure 5.1: Hierarchy of communication model semantics.

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 [20].

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.

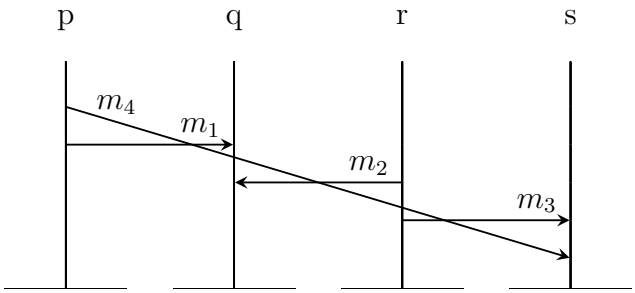

Figure 5.2: An example of mailbox semantic.

FIFO 1-n This model (`onen`) is the dual of `mb`, it coordinates a sender with all the receivers. Any two messages sent by a process must be received in the same order as they are sent. These two messages might be received by different processes and the two receive events might be concurrent.

FIFO n-n In this model (`nn`), messages are globally ordered and delivered according to their emission order. Any two messages must be received in the same order as they are sent. These two messages might be sent or receives by any process and the two send or receive events might be concurrent. The FIFO `n-n` coordinates all the senders with all the receivers.

5.2 Realisability for MSCs

For finite sets of MSCs, weakly realisability as defined in [3] is `coNP`-complete and safe realisability is shown to be decidable in `P`-time [3]. The problem was subse-

quently studied for infinite MSC languages, defined as MSC-Graphs (MSGs). For bounded MSGs, safe realisability remains decidable, but weak realisability is undecidable [3]. Extensions of these results to non-FIFO semantics were investigated in [25], corresponding to bag semantics under peer-to-peer communication. Later, Lohrey proved that in the general case, safe realisability is undecidable [23], though it is decidable (and EXPSPACE-complete) for MSGs. Most positive results assume bounded channels, but [4] introduces a new class of HMSCs that allows unbounded channels while maintaining implementability. 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.

5.3 Realisability for MPST

Recent work has focused on strengthening the connection between MPST and automata-theoretic formalisms. Stutz and Zufferey showed that implementability is decidable by encoding global types into HMSCs that are globally cooperative [29, 26]. Building on this, Li et al. [22] proposed a complete projection function for MPST, guaranteeing that every implementable global type admits a correct distributed implementation.

Stutz's thesis [27] 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 implementability shown to be in PSPACE-the first precise complexity bound for this fragment.

5.4 Choreographies

Choreographies [24] are another formalism to describe distributed communication protocols. Unlike MSCs or MPST, which focus either on trace-based semantics or type systems, 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 throughout a projection operation from the global-view. The *knowledge of choice* problem is similar to the implementability one, and explored also in choreographies, but it was first introduced by Castagna et al. [7].

5.5 Other works

Stutz et al. [28] proposed *Protocol State Machines* (PSMs), an automata-based formalism subsuming both MPST and HMSCs. PSMs show that many syntactic restrictions of global types are not true expressivity limits. Yet, the implementability problem for PSMs with unrestricted mixed choice remains undecidable, resolving the open question that mixed-choice global types are undecidable in general.

In summary, projectability is well understood for sender-driven choice, where decidability and complexity bounds are established, but moving towards mixed choice inevitably leads to undecidability. Automata-based techniques such as HMSCs and PSMs provide the most powerful tools for extending the theory while preserving decidability in restricted cases.

Chapter 6

Conclusion

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

On the theoretical side, I introduced the necessary background notions (i.e. CFSMs, Global Types, MSCs, and communication models) and formalized weak realisability. The main contribution was to connect the implementability problem to classical undecidability results, in particular through a reduction to the Relaxed Post Correspondence Problem (RPCP).

On the practical side, I 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.

Overall, the contributions span two complementary directions: a refined theoretical understanding of implementability, and concrete advances in tool support for experimenting with increasingly expressive models.

6.1 Future Work

Future directions include extending the theoretical results beyond weak realisability toward a decidability result of *safe realisability* (therefore, including deadlock-

freedom) for Global Types, building on the techniques developed here and extending an existing proof made by Lohrey, et al. [23]. On the practical side, a natural goal is to further enhance ReSCu to support these results, ultimately aiming for a complete algorithm to decide implementability for restricted classes of Global Types. This would enable systematic benchmarking against existing methods and real-world protocols.

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] 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.

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

[6] 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.

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

[8] 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.

[9] 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].

[10] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

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

[25] Rémi Morin. Recognizable sets of message sequence charts. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 523–534. Springer, 2002.

[26] Felix Stutz. Asynchronous multiparty session type implementability is decidable–lessons learned from message sequence charts. *arXiv preprint arXiv:2302.11272*, 2023.

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

[28] 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.

[29] Felix Stutz and Damien Zufferey. Comparing channel restrictions of communicating state machines, high-level message sequence charts, and multiparty session types. *arXiv preprint arXiv:2208.05559*, 2022.

Acknowledgments

Thanks for...