Implementability of Asynchronous Communication Protocols

The Power of Choice

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Natural Sciences (Dr. rer. nat.)

to

Felix Stutz

Date of Defence:	15 December 2023
Dean:	Prof. Dr. Christoph Garth
Reviewer:	Prof. Dr. Rupak Majumdar
Reviewer:	Prof. Dr. Annette Middelkoop-Bieniusa
Reviewer:	Prof. Dr. Thomas Wies
Reviewer:	Dr. Damien Zufferey

DE-386

Copyright © 2024 by Felix Stutz.

Typeset using LaTeX. The main text uses the Linux Libertine font.
Most of the illustrations are produced using the Ti*k*z package.

Summary

Distributed message-passing systems have become ubiquitous and essential for our daily lives. Hence, designing and implementing them correctly is of utmost importance. This is, however, very challenging at the same time. In fact, it is well-known that verifying such systems is algorithmically undecidable in general due to the interplay of asynchronous communication (messages are buffered) and concurrency. When designing communication in a system, it is natural to start with a global protocol specification of the desired communication behaviour. In such a top-down approach, the *implementability problem* asks, given such a global protocol, if the specified behaviour can be implemented in a distributed setting without additional synchronisation. This problem has been studied from two perspectives in the literature. On the one hand, there are Multiparty Session Types (MSTs) from process algebra, with global types to specify protocols. Key to the MST approach is a so-called projection operator, which takes a global type and tries to project it onto every participant: if successful, the local specifications are safe to use. This approach is efficient but brittle. On the other hand, High-level Message Sequence Charts (HMSCs) study the implementability problem from an automata-theoretic perspective. They employ very few restrictions on protocol specifications, making the implementability problem for HMSCs undecidable in general. The work in this thesis is the first to formally build a bridge between the world of MSTs and HMSCs. To start, we present a generalised projection operator for sender-driven choice. This allows a sender to send to different receivers when branching, which is crucial to handle common communication patterns from distributed computing. Despite this first step, we also show that the classical MST projection approach is inherently incomplete. We present the first formal encoding from global types to HMSCs. With this, we prove decidability of the implementability problem for global types with sender-driven choice. Furthermore, we develop the first direct and complete projection operator for global types with sender-driven choice, using automata-theoretic techniques, and show its effectiveness with a prototype implementation. We are the first to provide an upper bound for the implementability problem for global types with sender-driven (or directed) choice and show it to be in PSPACE. We also provide a session type system that uses the results from our projection operator. Last, we introduce protocol state machines (PSMs) – an automata-based protocol specification formalism – that subsume both global types from MSTs and HMSCs with regard to expressivity. We use transformations on PSMs to show that many of the syntactic restrictions of global types are not restrictive in terms of protocol expressivity. We prove that the implementability problem for PSMs with mixed choice, which requires no dedicated sender for a branch but solely all labels to be distinct, is undecidable in general. With our results on expressivity, this answers an open question: the implementability problem for mixed-choice global types is undecidable in general.

Acknowledgements

For me, this thesis marks a milestone in an incredible journey of scientific and personal development. Many people have accompanied me on this journey and I would like to take the opportunity to thank them.

I thank Damien Zufferey and Rupak Majumdar: for being a great team of advisors, for giving me the freedom to pursue what interested me most, for teaching me to search for the essence of research problems, and for giving me new insightful perspectives in virtually every discussion we had.

The work in this thesis would not have been possible without them – as well as an amazing group of collaborators: thanks to Madhavan Mukund for his patience with a 1st year doctoral student, thanks to Elaine Li for being a great fellow combatant on the proof front and all the fruitful low-level technical discussions, and thanks to Thomas Wies for his great eye to detail, for his joy in both exposition and technical discussions, and for being an examiner for my thesis.

A big thank you goes to Emanuele D'Osualdo who has become a friend and mentor to me. He has always been pushing for the best results possible whilst being the kindest and most understanding.

I thank Annette Bieniusa, who helped me find the path through a jungle of bureaucracy and agreed to be an examiner for my thesis, as well as Klaus Schneider, who chaired my thesis defence.

During my time at Max Planck Institute for Software Systems (MPI-SWS), I was fortunate to be a part of a big and supportive research group with various interests and expertise, including Rupak Majumdar, Damien Zufferey, Anne Kathrin Schmuck, Daniel Neider, Burcu Kulahcioglu Ozkan, Ivan Gavran, Kaushik Mallik, Mahmoud Salamati, Ivan Fedotov, Rajarshi Roy, Xuan Xie, Stanly Samuel, Aman Mathur, Marcus Pirron, Mehrdad Zareian, Ramanathan Thinniyam, Jie An, Jiarui Gan, Jaroslav Bendik, Julian Haas, Ashwani Anand, Satya Prakash Nayak, Ana Mainhardt, Germano Schafaschek, Munko Tsyrempilon, Sathiyanarayana Venkatesan Ramesh, and Alexandra Bugariu.

Numerous interactions with amazing researchers during conferences as well as summer and winter schools have influenced my scientific understanding and thinking. Contributing to this thesis, Anca Muscholl helped me understand the undecidability proof by Markus Lohrey, allowing me to adapt it to our setting for this thesis. Less related but still an invaluable experience was an internship at Massachusetts Institute of Technology, hosted by Martin Rinard, during which I worked with Nikos Vasilakis, Konstantinos Kallas, and Michael Greenberg.

One could argue that I had the steepest learning curve outside of academia, thanks to Manuel Gomez Rodriguez. During our first ski trip, we pushed beyond my perceived limits, sparking the joy of skiing again.

For many of us, MPI-SWS is more than a research institute. There is a big social group and I would not want to miss our dinners, playing pool and board games but also chats over coffee or tea. Besides many of the people above, there have been Stratis Tsirtsis, Marco Maida, Marco Perronet, Pascal Baumann, Arpan Gujarati, Manohar Vanga, Tobias Blass, James Robb, Kata Einarsdottir, Rosa Abbasi, Amir Mashaddi, Michalis Kokologiannakis, Azalea Raad, Clothilde Jeangodoux, Hasan Eniser, Numair Mansour, Nina Corvelo Benz, Nastaran Okati, Iason Marmanis, Eleni Straitouri, Kimaya Bedarkar, Lia Schütze, Eiren Vlassi Pandi, Irmak Saglam, Cédric Courtaud, Léo Stefanesco, Filip Markovic, Pavel Golovin, Aristotelis Koutsouridis, Suhas Muniyappa, and Ivy Chatzi.

I also want to thank the office, IT and HR staff at MPI-SWS for their help and support, in particular Vera Schreiber, Susanne Girard, Geraldine Anderson, Tobias Kaufmann, Andreas Ries, Claudia Hesse, Michael Bentz, and Sarah Naujoks.

It is fair to say that my fascination for computer science was sparked in high school. My first computer science teacher, Laura Schmidt (now Stilgenbauer), who had just graduated from University of Kaiserslautern, did an amazing job in facilitating both creative chaos and structured learning. In my last two years of high school, Michael Bergau was a great and enthusiastic facilitator for all our projects and prepared us incredibly well for university. At that time, I participated in a computer science competition. Its final round was co-sponsored by MPI-SWS so Mary-Lou Albrecht gave us a tour. Roughly two years later, she introduced me to Björn Brandenburg who hired me as undergraduate research assistant. This marked my first foray into the world of research, during which I learned a lot and I am thankful for the mentorship and guidance I received. Eventually, after some detours, I ended up at MPI-SWS again.

All along my journey, I had the unwavering support from my family and friends for which I am deeply grateful. I thank my sister Anja Stutz, my mother Sabine Stutz and her partner Ralf-Dieter Irsch, my grandparents Ulla and Willi Harz, my father Frank Stutz and his wife Birte Stutz, my late grandparents Trudel and Dieter Stutz, as well as my brother-in-law Philipp Neuheisel and my parents-in-law Marina and Hans Neuheisel. My biggest thank you, though, goes to my wife Laura Stutz: for your love, support and patience. Your gift to cheer me up in any situation is incredible and the certainty that you never loose faith in me has carried me throughout this journey. We both share a great dedication to what we do, and I dedicate this thesis to you.

To my wife Laura.

Contents

Contents ix

List of Figures xiii

List of Tables xv

List of Abbreviations xvii

1 Introduction 1
- 1.1 An Introduction to Communication Protocols . . . 2
- 1.2 The Implementability Problem . . . 5
- 1.3 Problem Setting . . . 7
- 1.4 State of the Art . . . 8
- 1.5 Research Questions and Contributions . . . 10
- 1.6 Publications . . . 12
- 1.7 Outline . . . 13

2 Protocol Specifications and the Implementability Problem 15
- 2.1 Preliminaries . . . 15
- 2.2 Alphabets for Protocols . . . 16
- 2.3 The Implementation Model: Communicating State Machines . . . 17
- 2.4 Indistinguishability Relation . . . 20
- 2.5 High-level Message Sequence Charts . . . 24
- 2.6 Global Types from Multiparty Session Types . . . 29
- 2.7 The Implementability Problem . . . 32

3 Generalising Projection for Multiparty Session Types 35
- 3.1 Classical Multiparty Session Type Projection . . . 35
  - 3.1.1 Introductory Example . . . 35
  - 3.1.2 Local Types . . . 36
  - 3.1.3 Classical Projection Operator with Parametric Merge . . . 38
  - 3.1.4 Visual Explanation of the Parametric Projection Operator . . . 39
  - 3.1.5 Visual Explanation of Merge Operators . . . 40
  - 3.1.6 Features of Different Merge Operators by Example . . . 41
- 3.2 Generalising Classical Projection for Sender-driven Choice . . . 45
  - 3.2.1 Motivating Example: Load Balancing . . . 45
  - 3.2.2 Available Messages . . . 47
  - 3.2.3 Availability Merge Operator . . . 49

3.2.4 Generalised Projection . . . 50
3.2.5 Revisiting the Load Balancing Protocols . . . 51
3.2.6 Evaluation . . . 52
3.3 Soundness of Generalised Projection: Projectability implies Implementability . . . 53
3.3.1 Implementability = Protocol Fidelity + Deadlock Freedom . . . 54
3.3.2 Generalised Projection Does Not Remove Behaviours . . . 54
3.3.3 Generalised Projection Does Not Introduce New Behaviours . . . 55
3.3.4 Family of Run Mappings Exists for Projectable Global Types . . . 56
3.3.5 From Run Mappings via Control Flow Agreement to Implementability . . . 70
3.3.6 Wrapping Up: Projectable Global Types are Implementable . . . 74
3.4 Incompleteness of Classical Projection Approaches . . . 75

4 Building a Bridge from MSTs to HMSCs **79**
4.1 Encoding Global Types from MSTs as HMSCs . . . 79
4.2 MST Implementability is Decidable . . . 86
4.3 MSC Techniques for MST Verification . . . 95
4.3.1 $\mathcal{I}$-closed Global Types . . . 95
4.3.2 Payload Implementability . . . 97
4.4 Implementability with Intra-participant Reordering . . . 99
4.4.1 A Case for More Reordering . . . 99
4.4.2 Undecidability . . . 101

5 Direct and Complete Projection for Multiparty Session Types **107**
5.1 Constructing Implementations . . . 107
5.2 Checking Implementability . . . 110
5.2.1 Send Validity . . . 110
5.2.2 Receive Validity . . . 111
5.2.3 Subset Projection . . . 114
5.3 Soundness . . . 114
5.4 Completeness . . . 117
5.5 CSMs vs. Local Types . . . 119
5.5.1 On Mixed-choice States . . . 120
5.5.2 Sink States and Deadlocks . . . 120
5.6 Complexity . . . 124
5.7 Evaluation . . . 125
5.8 Properties Entailed by Implementability . . . 127

6 A Type System Using Communicating State Machines **129**
6.1 Payload Types and Delegation . . . 129
6.2 Process Calculus . . . 132
6.3 Type System for Processes and Runtime Configurations . . . 136
6.4 Soundness of Type System . . . 143
6.5 On Subtyping . . . 164

7 Channel Restrictions of Protocols and CSMs **167**
- 7.1 Channel Restrictions . . . 167
  - 7.1.1 Half-duplex Communication . . . 168
  - 7.1.2 Existential B-boundedness . . . 169
  - 7.1.3 k-synchronisability . . . 170
  - 7.1.4 Channel Restrictions and Indistinguishability Relation $\sim$ . . . 172
- 7.2 Channel Restrictions of Protocols . . . 173
  - 7.2.1 Channel Restrictions of High-level Message Sequence Charts . . . 174
  - 7.2.2 Channel Restrictions of Global Types . . . 176
- 7.3 Channel Restrictions of CSMs . . . 177

8 A Unifying Protocol Specification Formalism **179**
- 8.1 Protocol State Machines . . . 179
- 8.2 Expressivity Results . . . 183
  - 8.2.1 Global Types as Special Class of PSMs . . . 183
  - 8.2.2 From $\Sigma 1$-PSM to Global Types . . . 184
  - 8.2.3 From HMSCs to PSMs . . . 195

9 Checking Implementability with Mixed Choice is Undecidable **197**

10 Related Work **209**
- 10.1 High-level Message Sequence Charts . . . 209
  - 10.1.1 Choice Restrictions . . . 209
  - 10.1.2 Structural Restrictions . . . 211
- 10.2 Session Types . . . 211
  - 10.2.1 Generalising Restrictions on Choice . . . 212
  - 10.2.2 MST-based Works . . . 213
  - 10.2.3 Subtyping . . . 214
  - 10.2.4 Extensions . . . 215
- 10.3 Communicating State Machines and Channels . . . 215
- 10.4 Choreographic Programming . . . 216

11 Conclusion **219**

Bibliography **221**

A Appendix for Chapter 5 **237**
- A.1 Additional Material for Section 5.1 . . . 237
- A.2 Additional Material for Section 5.3 . . . 238
- A.3 Additional Material for Section 5.4 . . . 249

Curriculum Vitae **257**

List of Figures

1.1 Configuring the washing machine as a flow chart. 1
1.2 Laundry protocol. 2
1.3 Local behaviour for tumble dryer. 5

2.1 A communicating state machine. 18
2.2 Highlighting the elements of an MSC. 25
2.3 Constructing a prefix MSC from a FIFO-compliant word. 26
2.4 Concatenating MSCs. 27
2.5 Sending a list, specified as HMSC. 27
2.6 List-sending Protocol: FSM for the semantics of G_{LiSe}. 31

3.1 Two Buyer Protocol. 36
3.2 A positive and a negative example for plain merge. 42
3.3 A positive example for semi-full merge. 43
3.4 An example: negative for semi-full merge and positive for full merge. . . 44
3.5 Negative example for full merge. 44
3.6 Load Balancing Protocol, as HMSC. 45
3.7 Load Balancing Protocol, as FSM for G_{LoBa}. 45
3.8 Variant of load balancing with confusion (G'_{LoBa}): as HMSC and an execution with confusion as MSC. 47
3.9 Variant of load balancing with confusion (G'_{LoBa}): as FSM and after collapsing erasure. 48
3.10 An HMSC and its unfolding where all participants are blocked at the end. 60
3.11 Odd-even Protocol: implementable but not (yet) projectable. 77
3.12 Local implementations for the odd-even protocol. 77

4.1 Two Buyer Protocol: as HMSC and FSM as well as its determinised projection by erasure onto s. 87
4.2 An implementable HMSC which is not globally-cooperative with its implementation. 93
4.3 HMSC encoding $H(G_{\mathrm{MPCP}})$ of the MPCP encoding. 102

5.1 Odd-even revisited: as HMSC with labels for r and the subset construction for r. 108
5.2 HMSCs for G_s and G'_s and subset constructions onto participant r. . . . 110
5.3 HMSC for G_r and its subset construction onto participant r. 112
5.4 HMSC for G'_r and its subset construction onto participant r. 112
5.5 Evolution of $\mathrm{R}^G_{-}(\text{-})$ sets when p sends a message m and q receives it. . . . 116
5.6 Subset projection of global type that is not sink-final. 120

5.7 Projecting G_k for $0 < n < 14$ with our prototype tool. 127

6.1 Projection of the one buyer protocol onto seller s. 131
6.2 Projections of two global types onto its participants. 138
6.3 Subset projection and a wrong subtype. 165

7.1 List-sending Protocol revisited: specifications and implementation. . . . 168
7.2 Comparing half-duplex, existential B-bounded, and k-synchronisable protocols and systems. 173
7.3 BMSCs that satisfy different channels restrictions. 175
7.4 CSM with FSMs for participants p and q. 178

8.1 Kindergarten Leader Election Protocol. 181
8.2 A PSM whose semantics cannot be represented as HMSC. 182

9.1 MSC representation $\mathrm{msc}(w(C_1, D_1, C_2, D_2, C_3, D_3))$. 206

10.1 Reconstructible HMSC that is not implementable. 210
10.2 Conditions for choreography automata are unsound in the asynchronous setting. 214

List of Tables

3.1 Projecting global types with generalised projection. 52

5.1 Projecting global types with subset and generalised projection. 126

List of Abbreviations

BMSC	**B**asic **M**essage **S**equence **C**hart
CSM	**C**ommunicating **S**tate **M**achine
EXPSPACE	the class of problems that can be computed by a Turing Machine with **exp**onential (tape) **space**
EXPTIME	the class of problems that can be computed by a Turing Machine in **exp**onential **time**
FIFO	**F**irst **I**n **F**irst **O**ut
FSM	**F**inite **S**tate **M**achine
HMSC	**H**igh-level **M**essage **S**equence **C**hart
MSC	**M**essage **S**equence **C**hart
MST	**M**ultiparty **S**ession **T**ype
NFA	**N**on-deterministic **F**inite **A**utomaton
PCP	**P**ost **C**orrespondence **P**roblem
PSM	**P**rotocol **S**tate **M**achine
PSPACE	the class of problems that can be computed by a Turing Machine with **p**olynomial (tape) **space**
RE	**R**egular **E**xpression

Chapter 1

Introduction

Do you remember the first time you used a washing machine? For my sister and me, it was during our teenage years when our parents had decided it was time for us to gain independence and take our share of chores. Long before, we had learned not to leave tissues or pennies in our pockets. But, of course, this is only one of the numerous mistakes when using a washing machine. Both our parents do laundry so I can only speculate, but it could be our mum's professional background as teacher that made her the better fit to teach us doing laundry. Our mum decided to take things step by step and let us start with three types of laundry: whites, colours, and towels. On the one hand, there was a more general part, which was easy to remember, like pressing the start button and taking the clothes out of the washing machine. On the other hand, there were the laundry-type-specific instructions: temperature, washing detergent (both type and quantity) and number of rotations per minute. Admittedly, it was not particularly hard to remember but misremembering came with some risk as everyone knows who happened to wash colours at a temperature as high as for towels. So our mum put a sticky note with these instructions on our washing machine. It was similar to the flow chart diagram in Figure 1.1. At the top, one would first decide which type of laundry to do and could then easily recall the right instructions.

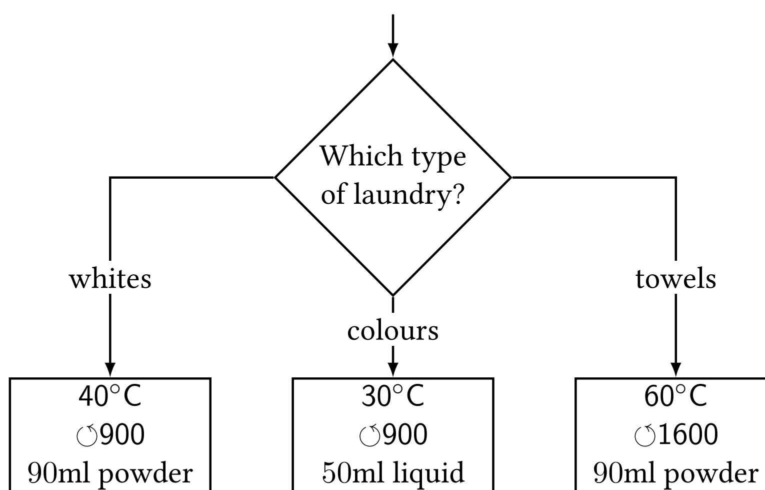


Figure 1.1: Configuring the washing machine as a flow chart.

1.1 An Introduction to Communication Protocols

One could say that this flow chart was part of our recipe to do laundry. As their more prominent counterparts for baking, recipes are basically *algorithms*, which specify a sequence of actions towards a goal – like a home-made currant-chocolate cake (my favourite one). Baking recipes, as our laundry recipe, describe all actions from one person's perspective. For instance, a user ought to press a button on the washing machine to start it but it would not specify the washing machine's behaviour once the button was pressed. So let us change perspective and consider the whole process of doing laundry as interaction between a (human) user, a washing machine and a tumble dryer. Such a hypothetical laundry protocol is visualised in Figure 1.2.

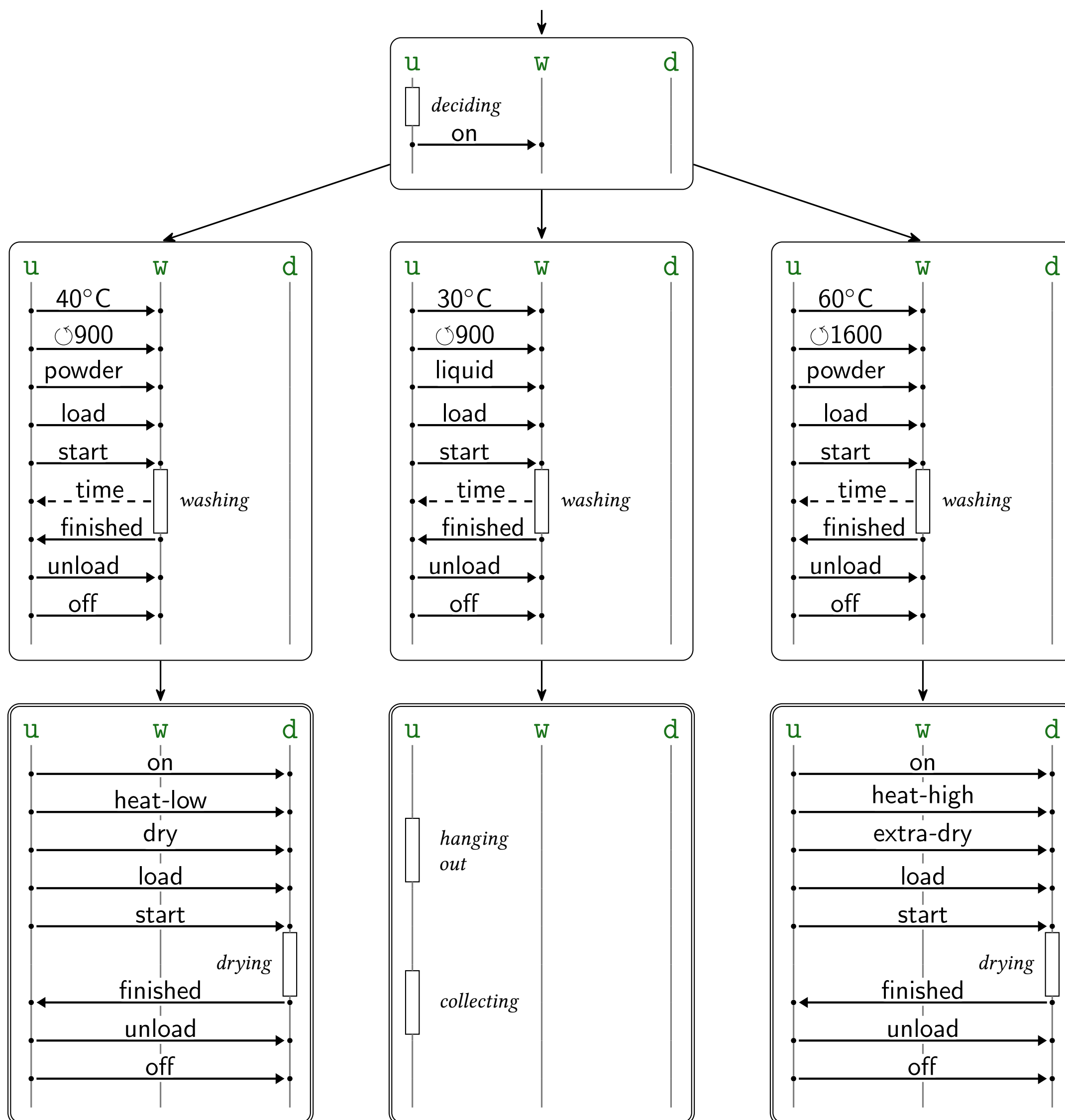


Figure 1.2: Laundry protocol
with a user `u`, a washing machine `w`, and a tumble dryer `d`.

Laundry Protocol. The *protocol* is structurally similar to the flow chart in Fig. 1.1: there is the initial block at the top, indicated by an incoming arrow, and there are three branches for the three types of laundry. In each block, every *participant* – the user u, the washing machine w, and the tumble dryer d – is represented with a vertical line. Time flows from top to bottom so in the initial block, the user first decides the type of laundry and then switches the washing machine on. The first action is purely *internal* to the user and their decision is not shared with any of the other participants (yet). The second action is, in fact, an *interaction* between the user and the washing machine. It is represented with an arrow from the user's vertical line to the one for the washing machine, labelled with on. Practically, one would press a button on the washing machine but our protocol is not as specific. There are no more actions in the initial block, leading us to the point where one of the three branches ought to be taken. Here, they can be distinguished by the first interaction where the user communicates the temperature to the washing machine, which are distinct for each type of laundry. If one thinks of an interaction as notifying another participant through a message, the user is the *sender* and the washing machine is the *receiver* of this message. In our protocol, the sender drives the decision which branch to choose, which is why we say the protocol has *sender-driven choice*. The washing machine learns about this choice by receiving a notification about the temperature. It knows which branch has been taken because the temperature in each branch is distinct. The structure of the first part of all three branches is similar. The user first notifies the washing machine about the number of rotations per minute, adds the right type (and quantity) of detergent, and loads the laundry into to washing machine. Afterwards, the user starts the washing machine. In turn, the washing machine actually washes the laundry, which is an internal action. While washing, it displays the remaining time to the user – indicated by the different style of message. Once completed, it notifies the user who unloads the laundry and switches the machine off. For colours, the user then hangs out the laundry, waits for some time and collects it once dried. For whites and towels, the user makes use of the tumble dryer. Their interaction resembles the one with the washing machine. The user switches the tumble dryer on, communicates the configuration details (heat and humidity level), loads the laundry into the tumble dryer, and starts it. The tumble dryer notifies the user once the laundry reached the configured dryness level who then unloads it and switches the tumble dryer off. The double lines around the last blocks indicate that we completed the protocol.

Abstraction. Our laundry protocol does not specify every single detail – for example how communication happens between the participants. This gives freedom when following the protocol and how the participants can communicate. For example, one could also use a smartphone app to configure settings of the washing machine instead of physical buttons. This follows the idea of *abstraction*, an important concept in computer science. In our example, it also allows to simply have blocks for internal actions which

do not specify how the washing machine exactly washes the laundry. With the buttons and means to configure, the washing machine exposes a user interface that allows to influence part of its internal actions but little do users know what happens inside. In fact, there are several internal components that themselves interact during the washing process. One could say that they follow their own subprotocol. In our setting, abstraction allows us to focus on the interactions between a particular group of participants and not to consider all internals of the washing machine. More broadly, abstraction also paved the way that we do not program processors, the very basic computation unit in a computer, directly but can use sophisticated high-level programming languages. To stay in our picture, these basically offer a user interface to the computers' internal components and have been proven essential for what can be accomplished with modern computer systems today.

Communication Protocols. Here, we take the idea of abstraction one step further and also abstract the internal actions for protocols. Intuitively, their results are not exposed to the outside world but, if desirable, they can be made explicit through interactions, e.g. our laundry protocol could also involve our mum and I could tell her which type of laundry I would do in the beginning. We call a protocol that only specifies interactions between participants a *communication protocol*. We only consider communication protocols in this work and may abbreviate them with the term *protocol* for readability. They are usually used to let a number of participants coordinate towards achieving a goal. Algorithms basically do the same but for a single participant and, hence, necessarily focus on the internal actions. With this idea in mind, communication protocols can be identified in virtually all scenarios where humans as well as machines interact. One can think of assembly lines as introduced by Henry Ford more than 100 years ago but also how hunters coordinated to hunt bigger animals long before. Today, every online purchase does not only involve the buyer and seller but requires communication with a payment service, a bank/credit card issuer, and a delivery company. For such scenarios, like a geo-distributed service, where multiple participants need to interact to achieve a goal, communication protocols are essential. Standardisation is inherent to communication protocols as a specification language for interactions and it is often useful to optimise processes. In our setting, it comes with another advantage. For different executions of the same communication protocol, different people, computer systems, or companies can play the role of the same participant: in our laundry protocol, it does not matter whether my sister or I were the user while, in an online purchase, different delivery companies can be used to ship a parcel. This also allows an online shopping platform to provide their services more easily at bigger scale. In our context, scalability translates to the ability to execute the same protocol more often and, thus, possibly in parallel.

1.2 The Implementability Problem

A communication protocol is a joint description of all participants' interactions from a global perspective – one could say a *global protocol*. When executing such a protocol, each participant can only observe the interactions it is involved in. For instance, the tumble dryer does not know the temperature the user communicates to the washing machine, which is key for the latter to know which branch was taken. In the presence of choices or branches in the protocol, this partial information may be insufficient for a participant to know what it can do to stay compliant with previous choices. In our laundry protocol, there is enough information. In particular, the user chooses the only branch and tells the tumble dryer later in each branch about the desired configuration. We can infer the local communication behaviour of the tumble dryer. Intuitively, we consider all interactions and ignore the ones the tumble dryer is not involved in. The result is illustrated in Figure 1.3.

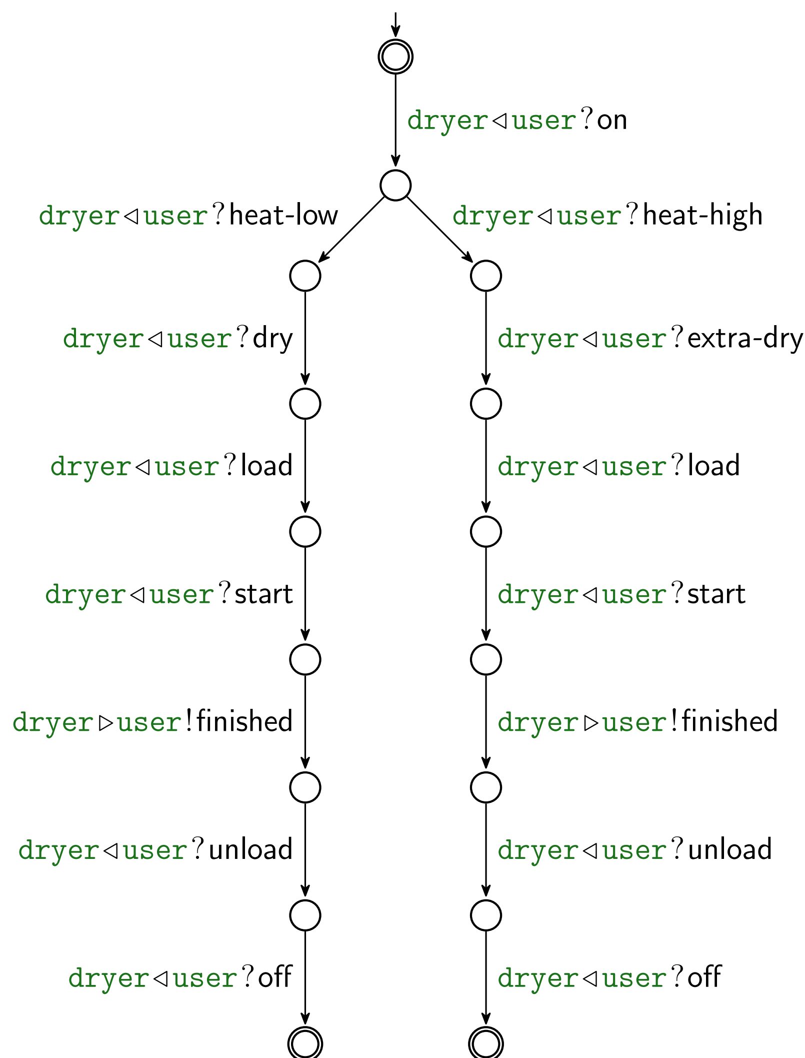


Figure 1.3: Local behaviour for tumble dryer.

It is represented as a *finite state machine* which consists of a finite set of states, here denoted with a circular shape, of which one is initial, some can be final, and they are connected with labelled transitions. The state at the top is the initial state and it is the one where the tumble dryer starts in the beginning. Transitions connect states in a directed way and can be taken to move from one state to another. They have labels, which describe part of an interaction in our setting: sending or receiving a message. For instance, there is only one transition from the initial state, which leads to second state and is labelled with $\texttt{dryer} \triangleleft \texttt{user} ? \mathsf{on}$, indicating that the tumble dryer (active participant and hence in front) receives (denoted by $?$ and the orientation of $\triangleleft$) message on from the user. Conversely, $\texttt{dryer} \triangleright \texttt{user} ! \mathsf{finished}$ denotes that the tumble dryer sends (denoted by $!$ and the orientation of $\triangleright$) message $\mathsf{finished}$ to the user. From the second state, there is a branch but the tumble dryer cannot decide which of both branches to take but the message to receive from the user determines the branch. The initial state and the two states at the bottom are final states, indicated by double lines as for the laundry protocol. This means it is fine for the tumble dryer to finish in one of these states. The initial state is also final because our laundry protocol does not involve the tumble dryer in one branch. We could simplify the local behaviour as presented: we observe that the last four actions are the same for both branches so we could merge both branches for the last four actions, not changing the possible local behaviour of the tumble dryer.

It is straightforward to construct the local behaviours of the user and washing machine for our laundry protocol in a similar way. Assume we let all three participants follow their local specification and we do not allow any additional interaction between them. Also, there is no external coordinator, e.g. our mum who might fix some configuration on the washing machine. In this scenario, when all three participants *locally* follow their local behaviour *in a distributed setting* (in contrast to a centralised setting with an external coordinator), they follow precisely the described global laundry protocol. This is only the case because all information about branches is propagated properly: the tumble dryer is informed about desired temperature level for drying, which is distinct for both branches. If not, it would need to guess and, overall, the global protocol, would be violated.

This is the first occurrence of the main research problem of this thesis.

Implementability Problem: *Given a global communication protocol, can it be implemented locally in a distributed setting?*

To be precise, *implementing* comprises two properties: *deadlock freedom* and *protocol fidelity*. Deadlock freedom is satisfied if every possible locally generated execution can always finish to completion: we would not want the tumble dryer to get stuck at some point and it will never be switched off. For protocol fidelity, two conditions ought to be met. First, every execution that is generated locally is also specified as global execution in the protocol. Second, all specified global executions can be generated locally in a distributed setting. Intuitively, we want to be able to wash and dry all described types of laundry with the prescribed configurations but also no more. Considering the

implementability problem as stated, an algorithm to solve it could simply return *yes* or *no* as result. We call this a *decision problem*. However, its solution does only tell us if there are local behaviours and will not provide them. But, of course, we would usually also like to obtain the local behaviours in order to use them subsequently. This would be the corresponding *synthesis problem*. It turns out, though, that generating the local behaviours is usually rather simple. For each global protocol and participant, one can apply the above idea of omitting irrelevant actions to obtain the local behaviour for every participant. If there is one that works, this will also work. Hence, it is not critical to distinguish between the decision and the synthesis problem. For most problem settings, an algorithm should be *sound*: if it outputs *yes* for a global protocol, then it is indeed implementable. The counterpart to soundness is completeness. An algorithm is *complete* if output *no* implies that the global protocol is not implementable. There are problems for which there are no terminating algorithms, i.e. that stop computing for every possible input, that are sound and complete.

1.3 Problem Setting

Let us elaborate on the setting for which we consider the implementability problem.

Regarding communication protocols, we mostly focus on protocols with sender-driven choice which means that, for every branch, there is a dedicated participant who sends the first message in all branches and the message-receiver-pairs are distinct. In contrast, if there is no such dedicated sender, but all labels are distinct, we say the protocol has *mixed choice*.

There are two main paradigms for communication: *synchronous* and *asynchronous* communication. For a synchronous interaction to happen, both participants need to be available at the same time to conduct their part of the interaction, as for a phone call. Asynchronous communication, in contrast, allows a participant to send its message without waiting for the receiver to receive it, as for a telephone answering machine or e-mails. We consider asynchronous communication in this thesis. This yields performance benefits when executing protocols but also makes the implementability problem more challenging.

Messages that were sent but have not been received yet are buffered in *channels* in our computational model. We consider a *point-to-point* setting with *FIFO-ordered*, *reliable* and *unbounded* channels. Point-to-point means that there are two channels between every two participants, one for each direction. For a single participant's perspective on its incoming channels, think of an e-mail inbox with a dedicated folder for every sender – a rather unlikely design choice for an e-mail inbox but still possible and good for illustrative purposes. FIFO-ordered means that, for the receiver, the messages in each directory occur in the order they were sent by the sender. Reliability amounts to not losing or duplicating e-mails. A channel is unbounded if it does not have a maximum capacity and, thus, never lets the sender of a message block, which would

be the case for a full channel. In terms of design choices, these assumptions are quite strong but they can be satisfied by implementing communication protocols on top of a fault-tolerant network layer that handles faulty communication like the Transmission Control Protocol (TCP). For many applications, this is a reasonable level of abstraction since handling communication faults comes with its own set of challenges.

In theoretical computer science, the described model is known as *communicating state machines* (CSMs) [23] and is commonly used to model computations in a distributed setting. In fact, it is a very powerful computational model as it was shown to be Turing-complete [23]. For instance, bounding the channels significantly restricts the computational power of the model. In fact, one can then simulate all behaviours in a finite amount of time. If channels can lose messages, one obtains lossy channel systems [3] while not imposing an order on messages yields Petri Nets [104]. Both these models of computation do not allow to explicitly simulate all behaviours in finite time but are not as powerful as communicating state machines.

1.4 State of the Art

For our setting, the implementability problem has been studied from two perspectives: with *High-level Message Sequence Charts* (HMSCs) [95] using automata-theoretic means, and *Multiparty Session Types* (MSTs) [67] which originate in process algebra.

High-level message sequence charts have been specified as industry standard in 1996 [127]. In the 2000s, HMSCs enjoyed popularity as part of the UML standard for the design of software. Research on implementability focused on the complexity of checking if a protocol can be implemented. The problem was shown to be undecidable in general [91] but several structural restrictions have been proposed that work around the undecidability [7, 100, 98, 91, 55], e.g. boundedness, which entails that channels never exceed a certain bound. In addition to the implementability problem, research on automated verification has been concerned with model-checking HMSC specifications, e.g. against temporal logic specifications. Again, this problem is undecidable in general but becomes decidable for bounded HMSCs [6].

Multiparty session types have been proposed as typing mechanism for communication channels. Their history starts back in 1993 when Honda [65] proposed binary session types for two party communication. They identified duality as a condition that ensures that communication channels are used in safe fashion. Roughly speaking, duality requires the following: if one participant sends a message of a specific type to a channel, the other endpoint should expect to receive a message of this type. This work was inspired by linear logic [58]. Intuitively, session type systems are often linear as they require that every channel type in the typing context needs to be taken care of and they cannot be dropped from the context. Moving from binary to the multiparty session types in 2008, Honda et al. [67] identified consistency as generalisation of duality. They provide a type system to type processes from a process calculus and well-typed

processes are guaranteed *not to go wrong* – a classical property of type systems. Also for multiparty communication, one concern is that there are no mismatches in the use of channels, e.g. messages of the given type are not expected. Different behavioural properties have also been considered, e.g. the absence of orphan messages [45]: once all participants terminated, there should be no messages left in the channel. In our work, we will focus on the implementability problem and show that many of the properties of interest follow from its guarantees. Session types have been implemented in various programming languages [8] and ideas from MSTs have been applied to various other domains, including web services [131], cyber-physical systems [94], and smart contracts [44].

The HMSC and MST approach differ both in terms of expressivity of their protocol specification mechanism and their approach to the implementability problem.

Let us first consider the protocols one can specify. If one omits the internal actions, the visual in Figure 1.2 depicts an HMSC. Each block is a basic message sequence chart (BMSC) in which communication is specified as explained before. It is also possible to have non-horizontal or crossing message arrows but only within a single BMSC. HMSCs do not impose any restrictions on branching like sender-driven choice. In MST frameworks, communication protocols are specified syntactically as global types. Here, we omit a formal description but explain their restrictions visually using HMSCs. Message arrows can only occur horizontally. Virtually every MST framework employs *directed choice*, which means that there is a dedicated sender-receiver-pair for each branching and only messages can be distinct. In contrast, sender-driven choice allows the receiver-message-pairs to be distinct, permitting patterns like load balancing. Structurally, all protocols have a tree-like structure. In addition, no protocol can finish somewhere in the middle but needs to explicitly specify termination. In other words, no completed execution can be part of another completed execution.

When it comes to the implementability problem, literature on HMSCs split the problem into two steps: (a) constructing a candidate implementation comprising local specifications for each participant and (b) checking implementability of the protocol. As hinted at before, (a) is rather straightforward so most efforts focus on (b). The implementability problem for HMSCs is undecidable in general [91] so there cannot be an algorithm that always finishes its computation and returns the correct result for all instances. Different classes of HMSCs were identified for which implementability is decidable, so sound and complete algorithms exist for these subclasses.

Classical MST approaches consider each participant in isolation and join both the construction of its local behaviour and checking whether it is safe to use. Central is the so-called *projection operator* which, given a global type and participant, traverses the global type in a linear fashion and projects each interaction onto the local behaviour for the participant. For some interactions, a participant might not be involved, leading to potential confusion about previous choices. An in-built *merge operator* checks whether such confusions can be resolved. This yields an efficient approach to the

implementability problem for global types. However, it is also rather brittle because these checks are overly restrictive and the linear traversal prohibits unfoldings, yielding an inherently incomplete approach. For example, sender-driven choice is virtually not supported by any classical MST framework. Interestingly, despite the rather heavy restrictions on global types, neither a lower nor upper complexity bound for the implementability problem (even with directed choice) is known. This means, regarding the lower bound, it is unclear how difficult the problem is so, roughly speaking, one does not know what run times can be deemed acceptable. For the upper bound, it is not even clear if there can be a sound and complete algorithm, which is known not to be the case for HMSCs.

1.5 Research Questions and Contributions

These observations give rise to a two-dimensional spectrum for interesting research questions. First, there is the spectrum of protocol expressivity, ranging from basically no restrictions for HMSCs to quite a number of restrictions for MSTs. Second, there is the difficulty of the implementability problem, ranging from undecidability for HMSCs in general to only incomplete linear approaches for MSTs.

We approached the expressivity spectrum rather from the MST perspective and considered ways to make the MST methodology applicable to more protocols, hoping to find ways not to sacrifice too much performance.

First, we observed that directed choice prevents the use of MST verification for patterns from distributed computing like load balancing, which comprise features that are supported by sender-driven choice.

How can one support sender-driven choice in the MST methodology?

We extended the classical projection approach to support sender-driven choice. It was interesting to realise that the main difficulty originates from the fact that a participant can learn about branches by receiving from different senders. Messages from different senders are not FIFO-ordered, however. This is why one needs to ensure that such important messages cannot overtake each other. We employed a novel message availability analysis, inspired by HMSCs, to achieve this and proved it to be sound.

This projection operator has been based on the classical projection approach which we showed to be inherently incomplete.

Is there (any hope for) a complete algorithm for the MST implementability problem with directed or even sender-driven choice?

We gave the first EXPSPACE decision procedure for a subclass of sender-driven global types – to be precise, the ones that always can but do not need to terminate. This result is obtained via a reduction to the HMSC implementability problem and proving that any implementable global type falls into a class of HMSCs for which implementability is decidable. Subsequently, we also provided the first direct and complete projection operator through automata-theoretic constructions, yielding a PSPACE upper bound and lifting the restriction on termination for protocols. We also present a family of examples that requires exponential time to generate an implementation. Despite, we demonstrated its effectiveness and efficiency with a prototype tool on examples from the literature.

Our projection operator uses CSMs as implementation model. Usually, MST frameworks use local types instead, which are subsequently used in a session type system to type-check processes from a process calculus. In our setting, it was reasonable to consider the use of CSMs in a type system.

Can CSMs be used in a session type system?

We answered this question positively and constructed a type system where CSMs take the place of local types. We proved that our type system ensures subject reduction, i.e. if a well-typed process can take a step, its typing context can also take a step and can be used to type-check the new process. Because errors can never be well-typed, our type system enjoys type safety, i.e. no well-typed process can ever reduce to an error. In a nutshell, our sound and complete projection operator generates CSMs and our type system uses CSMs. Together, it shows that the use of CSMs is beneficial and advantageous for both projection as well as type-checking, proving it to be reasonable intermediate interface. Overall, this improves generality without loosing efficiency.

While we were developing algorithms for the implementability problem, we wondered how implementations for protocols are different from arbitrary CSMs. Since channels are key to the computational power of CSMs, it was natural to consider this aspect.

How do protocols use channels in comparison to
arbitrary communicating state machines?

We considered three common channel restrictions and, interestingly, found that their relations differ in the context of protocols and arbitrary communicating state machines. We found that protocols, as specified with global types and HMSCs, provide a number of sanity guarantees by default. For instance, it should be possible to receive every sent message in a bounded (logical) period of time – a property called *existential boundedness*. We noticed, though, that HMSCs cannot express all such protocols.

Is there a formalism to specify (more) existentially bounded protocols
for which sanity guarantees are checkable?

We introduced *protocol state machines* (PSMs) as such formalism, which are basically finite state machines with transitions labelled by send and receive events. In contrast to global types and HMSCs, a condition in their definition ensures that channels are used in FIFO manner for every word. PSMs are more expressive than global types. First, they allow to specify arbitrary state machines while we show global types to always specify state machines with a tree-like structure in which recursion only happens at the leaves and to ancestors, and final states cannot have outgoing transitions. Second, we showed that global types only specify existentially $\Sigma 1$-bounded protocols, i.e. where every execution can be reordered in a way that there is at most one message in all channels. Similarly, we defined the subclass of $\Sigma 1$-PSMs. Our sound and complete projection algorithm actually applies to sender-driven $\Sigma 1$-PSMs where final states have no outgoing transitions – a property abbreviated as sink-final since states without outgoing transitions are called sink states. Still, we showed that every sink-final $\Sigma 1$-PSM can be transformed into a global type, specifying the same protocol and preserving sender-driven and mixed choice if given. This entails that most of the structural restrictions on global types are actually non-restrictive in terms of expressivity.

We proved sender-driven choice to be instrumental in the design of a complete projection operator. Naturally, the question arises if similar techniques can be applied for mixed-choice protocols.

Is the implementability problem decidable for protocols with mixed choice?

Lohrey [91] showed that the implementability problem for HMSCs is undecidable in general. Consequently, the same holds for PSMs. However, the notion of choice was not studied much for HMSCs and it was unclear how choice comes into play for this result. Hence, we decided to transcribe the original proof to understand the role of choice in detail. Further changes allowed us to show that checking implementability is undecidable for sink-final $\Sigma 1$-PSMs in general. With our earlier expressivity results, this settles an open question: the implementability problem for mixed-choice global types is undecidable in general. Both the implementability problem for sender-driven HMSCs and PSMs are open and we consider both to be interesting avenues for future work.

1.6 Publications

The material in this thesis is based on four publications in peer-reviewed conferences and one unpublished draft:

(1) Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. “Generalising Projection in Asynchronous Multiparty Session Types”. In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*. Ed. by Serge Haddad and Daniele Varacca. Vol. 203. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 35:1–35:24. DOI: 10.4230/LIPIcs.CONCUR.2021.35

(2) Felix Stutz and Damien Zufferey. "Comparing Channel Restrictions of Communicating State Machines, High-level Message Sequence Charts, and Multiparty Session Types". In: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022.* Ed. by Pierre Ganty and Dario Della Monica. Vol. 370. EPTCS. 2022, pp. 194–212. DOI: 10.4204/EPTCS.370.13

(3) Felix Stutz. "Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts". In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States.* Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 32:1–32:31. DOI: 10.4230/LIPIcs.ECOOP.2023.32

(4) Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. "Complete Multiparty Session Type Projection with Automata". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III.* ed. by Constantin Enea and Akash Lal. Vol. 13966. Lecture Notes in Computer Science. Springer, 2023, pp. 350–373. DOI: 10.1007/978-3-031-37709-9_17

(5) Felix Stutz and Emanuele D'Osualdo. "An Automata-theoretic Basis for Specification and Type Checking of Multiparty Protocols"

Many of these publications are the result of fruitful collaborations with great researchers, including my advisors, external collaborators and people who became mentors to me over time. I will use the first-person plural *we* throughout the thesis, also to indicate that most of these results were not obtained solely by myself.

I can be considered the primary contributor to all these publications – except for Publication (4) for which Elaine Li and I share equal contribution. It uses the novel idea of available message analysis from Publication (1) but improves on it in many aspects. Elaine and I both independently observed that this idea could be applicable more broadly when applying automata-theoretic techniques. We then closely co-developed the ideas and definitions for this publication. We also discussed proof ideas as well as low-level proofs in detail but she was the one to put them down on paper (and also the one to find unexpected subtleties whilst doing this). I can take credit for the implementation of the prototype tool. Consequently, I decided to provide definitions, give proof sketches and intuition in the main text of this thesis and keep the detailed proofs in an appendix.

1.7 Outline

Chapter 2 formally introduces high-level message sequence charts (HMSCs) and global types from multiparty session types (MSTs) as protocol specification mechanisms, communicating state machines as implementation model, and the implementability problem. In Chapter 3, we generalise classical projection to support sender-driven

choice. We first explain the classical projection approach visually using finite state machines. Then, we elaborate on our generalisation in the main part of the chapter. Last, we exemplify the sources of incompleteness of the classical MST projection approach. Chapter 4 first establishes a formal connection between global types and high-level message sequence charts. We then thoroughly explain how this connection can help to solve the MST implementability problem, including our decision procedure for MST implementability. This is restricted to sender-driven choice and protocols such that every partial execution can be completed to a finite completed execution. In Chapter 5, we present a direct and complete MST projection operator, which uses automata-theoretic constructions and applies to the full class of sender-driven global types. We make use of these results in our session type system in Chapter 6, proving subject reduction and type safety. Chapter 7 investigates the relationship of common channel restrictions for protocols as well as general communicating state machines. In Chapter 8, we introduce protocol state machines and elaborate on their relation to global types and HMSCs, while we prove undecidability of the implementability problem for mixed-choice protocol state machines as well as mixed-choice global types in Chapter 9. We discuss related work in Chapter 10 and give a conclusion in Chapter 11.

Chapter 2

Protocol Specifications and the Implementability Problem

In this chapter, we lay the formal foundations for the developments in this thesis, by providing definitions for protocols and their implementations. We define communicating state machines as computational model for implementations of protocols, which can be specified as high-level message sequence charts and global types from multiparty session types. Last, we define the implementability problem for protocols.

2.1 Preliminaries

Let us give a brief exposition of notation and standard definitions.

Finite and Infinite Words. Let Δ be an alphabet. We denote the set of finite words over Δ by Δ^* and the set of infinite words by Δ^ω. Their union is denoted by Δ^∞. For two words $u \in \Delta^*$ and $v \in \Delta^\infty$, we say that u is a *prefix* of v, denoted with $u \leq v$, if there is some $w \in \Delta^\infty$ such that $u \cdot w = v$, where $\cdot$ is the concatenation operator and might be omitted for conciseness. All prefixes of v are denoted by $\mathrm{pref}(v)$, which is lifted to languages as expected. For a language $L \subseteq \Delta^\infty$, we distinguish between the language of finite words $L_{\text{fin}} := L \cap \Delta^*$ and the language of infinite words $L_{\text{inf}} := L \cap \Delta^\omega$.

Partial Orders and Linearisations. Let S be a set and $\leq$ be a partial order over S, i.e. $\leq$ is reflexive, antisymmetric, and transitive. We say that $w_1 \ldots w_{|S|} \in S^*$ is a *linearisation* of S with regard to $\leq$ if for every $i \leq j$ it holds that $w_i \leq w_j$ and for every $x \in S$, there is i with $w_i = x$. We omit the set S when it is clear from context.

State Machines. A *state machine* $A = (Q, \Delta, \delta, q_0, F)$ is a 5-tuple with a finite set of states Q, an alphabet Δ, a transition relation $\delta \subseteq Q \times (\Delta \cup \{\varepsilon\}) \times Q$, an initial state $q_0 \in Q$ from the set of states, and a set of final states F with $F \subseteq Q$. We say $q \in Q$ is a *sink* if it has no outgoing transitions, i.e. $\forall a \in \Delta \cup \{\varepsilon\}, q' \in Q. (q, a, q') \notin \delta$. The state machine A is called *sink-final* if every state is final if and only if it is a sink. A is deterministic if for every transition $(q, a, q') \in \delta$, it holds that $a \neq \varepsilon$ and for every transition $(q, a, q'') \in \delta$, we have $q' = q''$. For $(q, a, q') \in \delta$, we also write $q \xrightarrow{a} q'$. For the transitive and reflexive closure, we write δ^* or $\rightarrow^*$. If the letter does not matter, $q \rightarrow q'$ denotes that there is some a with $q \xrightarrow{a} q'$. A sequence $\rho = q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \ldots$, with $q_i \in Q$ and $w_i \in \Delta \cup \{\varepsilon\}$ for $i \geq 0$, such that q_0 is the initial state, and for each $i \geq 0$, it holds that $(q_i, w_i, q_{i+1}) \in \delta$, is called a *run* in A with its *trace* $w_0 w_1 \ldots \in \Delta^\infty$, denoted by $\text{trace}(\rho)$. A run is *maximal* if it ends in a final state or is infinite. The *language* $\mathcal{L}(A)$ of A is the set of traces of all maximal runs. If Q is finite, we say A is a *finite state machine* (FSM).

State machines can generate languages of finite and infinite words and will serve as basic component for communicating state machines. Figure 1.3 depicts an FSM. Its transition labels are from a special alphabet that we use to formalise the execution of protocols in a distributed setting and define next.

2.2 Alphabets for Protocols

We formalise the execution of protocols as formal languages over an alphabet of (asynchronous) events. To start with, let $\mathcal{P}$ be a finite set of (protocol) *participants* and $\mathcal{V}$ be a finite set of messages. Let us first consider the events for a single participant. For $\mathtt{p} \in \mathcal{P}$, we define the alphabet of *send* events $\Gamma_{\mathtt{p},!} = \{\mathtt{p}\triangleright\mathtt{q}!\mathtt{m} \mid \mathtt{q} \in \mathcal{P} \setminus \{\mathtt{p}\},\ \mathtt{m} \in \mathcal{V}\}$ and the alphabet of *receive* events $\Gamma_{\mathtt{p},?} = \{\mathtt{p}\triangleleft\mathtt{q}?\mathtt{m} \mid \mathtt{q} \in \mathcal{P} \setminus \{\mathtt{p}\},\ \mathtt{m} \in \mathcal{V}\}$. The event $\mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$ denotes that participant $\mathtt{p}$ sends a message $\mathtt{m}$ to $\mathtt{q}$, and $\mathtt{p}\triangleleft\mathtt{q}?\mathtt{m}$ denotes that participant $\mathtt{p}$ receives a message $\mathtt{m}$ from $\mathtt{q}$. Note that a participant cannot send to or receive from itself. Thus, we assume $\mathtt{p} \neq \mathtt{q}$ when dealing with such events. The union of send and receive events yields all events for $\mathtt{p}$: $\Gamma_{\mathtt{p}} = \Gamma_{\mathtt{p},!} \cup \Gamma_{\mathtt{p},?}$. We say that participant $\mathtt{p}$ is *active* in the event x if $x \in \Gamma_{\mathtt{p}}$. We also define the alphabets of send and receive events by all participants: $\Gamma_! = \bigcup_{\mathtt{p} \in \mathcal{P}} \Gamma_{\mathtt{p},!}$, and $\Gamma_? = \bigcup_{\mathtt{p} \in \mathcal{P}} \Gamma_{\mathtt{p},?}$. Together, they consist of all asynchronous events $\Gamma_{\mathcal{P}} = \Gamma_! \cup \Gamma_?$. When dealing with words or languages over $\Gamma_{\mathcal{P}}$, we might want to refer to all events of a subalphabet Δ of $\Gamma_{\mathcal{P}}$. For an alphabet $\Delta \subseteq \Gamma$, we define a homomorphism $\Downarrow_\Delta : \Gamma \rightarrow \Delta \uplus \{\varepsilon\}$, where $x\Downarrow_\Delta$ is x if $x \in \Delta$ and ε otherwise. We lift $\Downarrow__$ to words and languages as expected. We may use _ to abbreviate notation for alphabet descriptions, e.g. $\mathtt{p}\triangleright\mathtt{q}!_$ for $\{\mathtt{p}\triangleright\mathtt{q}!\mathtt{m} \mid \mathtt{m} \in \mathcal{V}\}$, and write $w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}$ to select the subsequence of all send events in w where $\mathtt{p}$ sends a message to $\mathtt{q}$. We write $\mathcal{V}(w)$ to project the send and receive events in w onto their messages. A *paired event* is the sequence of a send event and its corresponding receive event: $\mathtt{p}\rightarrow\mathtt{q}:\mathtt{m} := \mathtt{p}\triangleright\mathtt{q}!\mathtt{m} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathtt{m}$. Consequently, we define the set of *paired events*

$\Sigma_{\mathcal{P}} := \{\mathtt{p} \to \mathtt{q} : \mathtt{m} \mid \mathtt{p}, \mathtt{q} \in \mathcal{P} \text{ and } \mathtt{m} \in \mathcal{V}\}$. For readability, we might exploit notation and treat $\mathtt{p} \to \mathtt{q} : \mathtt{m}$ as one instead of two letters, e.g. we might define finite state machines with transition labels from $\Sigma_{\mathcal{P}}$.

In a word $w = w_1 \ldots \in \Gamma_{\mathcal{P}}^{\infty}$, a send event $w_i = \mathtt{p} \triangleright \mathtt{q} ! \mathtt{m}$ is *matched* by a receive event $w_j = \mathtt{q} \triangleleft \mathtt{p} ? \mathtt{m}$ if $i < j$ and $\mathcal{V}((w_1 \ldots w_i) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} ! _}) = \mathcal{V}((w_1 \ldots w_j) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ? _})$, which is denoted by $w_i \vdash\!\dashv w_j$. A send event w_i is *unmatched* if there is no such receive event w_j.

A language $L \subseteq \Gamma^{\infty}$ satisfies *feasible eventual reception* if for every finite word $w := w_1 \ldots w_n \in L$ such that w_i is an unmatched send event, there is an extension $w \leq w' \in L$ of w such that w_i is matched in w'.

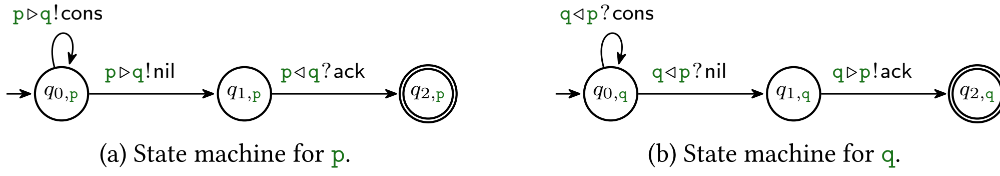
We fix $\mathcal{P}$ and $\mathcal{V}$ for the remainder of this thesis and may write Γ for $\Gamma_{\mathcal{P}}$ and Σ for $\Sigma_{\mathcal{P}}$.

2.3 The Implementation Model: Communicating State Machines

We model the behaviour of multiple participants that communicate through *asynchronous* messages over FIFO channels in a distributed setting. *Asynchronous* communication entails two properties. First, sending is non-blocking: a sender does not need to wait for the respective receiver to be available. Second, there can be an arbitrary delay between sending a message and its reception. We assume messages are not lost and, for each pair of participants, messages are delivered in FIFO order.

Definition 2.1 (Communicating state machines [23]). We say that $\mathcal{A} = \{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ is a *communicating state machine* (CSM) over $\mathcal{P}$ and $\mathcal{V}$ if $A_{\mathtt{p}}$ is a finite state machine over $\Gamma_{\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{P}$, denoted by $(Q_{\mathtt{p}}, \Gamma_{\mathtt{p}}, \delta_{\mathtt{p}}, q_{0,\mathtt{p}}, F_{\mathtt{p}})$. Let $\prod_{\mathtt{p} \in \mathcal{P}} Q_{\mathtt{p}}$ denote the set of global states and $\mathsf{Chan} = \{(\mathtt{p}, \mathtt{q}) \mid \mathtt{p}, \mathtt{q} \in \mathcal{P}, \mathtt{p} \neq \mathtt{q}\}$ denote the set of channels. A *configuration* of $\mathcal{A}$ is a pair $(\vec{q}, \xi)$, where $\vec{q}$ is a vector of states, one for each participant, and $\xi \colon \mathsf{Chan} \to \mathcal{V}^*$ is a mapping from each channel to a sequence of messages. We may write $\xi(\mathtt{p}, \mathtt{q})$ for $\xi((\mathtt{p}, \mathtt{q}))$. We use $\vec{q}_{\mathtt{p}}$ to denote the state of $\mathtt{p}$ in $\vec{q}$. The CSM transition relation, denoted $\to$, is defined as follows.

- $(\vec{q}, \xi) \xrightarrow{\mathtt{p} \triangleright \mathtt{q} ! \mathtt{m}} (\vec{q}\,', \xi')$ if $(\vec{q}_{\mathtt{p}}, \mathtt{p} \triangleright \mathtt{q} ! \mathtt{m}, \vec{q}\,'_{\mathtt{p}}) \in \delta_{\mathtt{p}}$, $\vec{q}_{\mathtt{r}} = \vec{q}\,'_{\mathtt{r}}$ for every participant $\mathtt{r} \neq \mathtt{p}$, $\xi'((\mathtt{p}, \mathtt{q})) = \xi((\mathtt{p}, \mathtt{q})) \cdot \mathtt{m}$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(\vec{q}, \xi) \xrightarrow{\mathtt{q} \triangleleft \mathtt{p} ? \mathtt{m}} (\vec{q}\,', \xi')$ if $(\vec{q}_{\mathtt{q}}, \mathtt{q} \triangleleft \mathtt{p} ? \mathtt{m}, \vec{q}\,'_{\mathtt{q}}) \in \delta_{\mathtt{q}}$, $\vec{q}_{\mathtt{r}} = \vec{q}\,'_{\mathtt{r}}$ for every participant $\mathtt{r} \neq \mathtt{q}$, $\xi((\mathtt{p}, \mathtt{q})) = \mathtt{m} \cdot \xi'((\mathtt{p}, \mathtt{q}))$ and $\xi'(c) = \xi(c)$ for every other channel $c \in \mathsf{Chan}$.
- $(\vec{q}, \xi) \xrightarrow{\varepsilon} (\vec{q}\,', \xi)$ if $(\vec{q}_{\mathtt{p}}, \varepsilon, \vec{q}\,'_{\mathtt{p}}) \in \delta_{\mathtt{p}}$ for some participant $\mathtt{p}$, and $\vec{q}_{\mathtt{q}} = \vec{q}\,'_{\mathtt{q}}$ for every participant $\mathtt{q} \neq \mathtt{p}$.



(a) State machine for p. (b) State machine for q.

Figure 2.1: A communicating state machine.

In the initial configuration $(\vec{q}_0, \xi_0)$, each participant's state in $\vec{q}_0$ is the initial state $q_{0,\mathrm{p}}$ of A_p, and ξ_0 maps each channel to ε. Let $(\vec{q}, \xi)$ be a configuration. It is said to be *final* iff $\vec{q}_\mathrm{p}$ is final for every p and ξ maps each channel to ε. We call $(\vec{q}, \xi)$ a *sink-state* configuration iff $\vec{q}_\mathrm{p}$ is a sink state for every p. A *run* of $\mathcal{A}$ always starts with its initial configuration $(\vec{q}_0, \xi_0)$, and is a finite or infinite sequence $(\vec{q}_0, \xi_0) \xrightarrow{w_0} (\vec{q}_1, \xi_1) \xrightarrow{w_1} \dots$ for which $(\vec{q}_i, \xi_i) \xrightarrow{w_i} (\vec{q}_{i+1}, \xi_{i+1})$. The word $w_0 \cdot w_1 \cdot \ldots \in \Gamma^\infty$ is said to be the *trace* of the run. A run is *maximal* if either it is finite and ends in a final configuration, or it is infinite. The language $\mathcal{L}(\mathcal{A})$ of the CSM $\mathcal{A}$ is defined as the set of maximal traces. A configuration $(\vec{q}, \xi)$ is a *deadlock* if it is not final and has no outgoing transitions while it is *reachable* if it occurs on some run of $\mathcal{A}$. 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. A CSM satisfies *feasible eventual reception* if its language does.

We use the prefix *feasible* for feasible eventual reception to indicate that the reception does not need to happen but is feasible. Thus, this does not impose a fairness condition on the runs of the CSM.

Example 2.2. Figure 2.1 depicts a CSM where participant p sends a list of elements to q. Let us exemplify the execution of a CSM. At the start, each is in its initial state, $q_{0,\mathrm{p}}$ and $q_{0,\mathrm{q}}$. Participant p is the only one that can take a step initially since q attempts to receive a message but all channels are initially empty. Thus, we let p send a sequence of elements using the self-loop to $q_{0,\mathrm{p}}$, e.g. two elements, yielding the trace p▷q!cons· p▷q!cons. We then let q receive the first message and p send another element of the list, appending q◁p?cons· p▷q!cons to our trace. All these actions lead them always back to their initial state. Once p sends nil, it proceeds to state $q_{1,\mathrm{p}}$, appending p▷q!nil. There, p waits to receive a message that has not been sent yet, yielding the first time where p cannot take any action. We let q receive all the messages in its channel and send ack to acknowledge the reception of the list. Together this yields the following trace:

p▷q!cons · p▷q!cons · q◁p?cons · p▷q!cons · p▷q!nil
· q◁p?cons · q◁p?cons · q◁p?nil · q▷p!ack · p◁q?ack

where q receives all cons-messages before the nil-message as we consider FIFO-ordered communication. After this trace, both p and q are in their final states and all channels are empty so the trace is in the language of the CSM. ◀

Given a word of send and receive events, we can tell if it is compliant with FIFO order and if all channels were empty if it was a trace of a CSM.

Definition 2.3 (FIFO-compliant and complete)**.** A word $w \in \Gamma^\infty$ is FIFO-compliant if messages are received after they are sent and, between two participants, the reception order is the same as the send order. Formally, for each prefix w' of w, we require $\mathcal{V}(w'\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_})$ to be a prefix of $\mathcal{V}(w'\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_})$, for every $\mathtt{p}, \mathtt{q} \in \mathcal{P}$. A FIFO-compliant word $w \in \Gamma^\infty$ is complete if it is infinite or the send and receive events match: if $w \in \Gamma^*$, then $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_})$ for every $\mathtt{p}, \mathtt{q} \in \mathcal{P}$.

Example 2.4. The trace in Example 2.2 is FIFO-compliant and complete. The following trace is not FIFO-compliant:

$$\mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{nil} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{nil} \ .$$

Here, $\mathtt{q}$ does not receive the second cons-message. Let us consider another trace:

$$\mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{nil} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{cons} \ .$$

It is FIFO-compliant but not (yet) complete. Appending $\mathtt{q}\triangleleft\mathtt{p}?\mathsf{nil}$ yields a complete trace that is also in the language of the CSM in Fig. 2.1. ◀

The following lemma summarises that traces of CSMs satisfy these conditions. While all traces are FIFO-compliant, only maximal traces are complete. Note that an infinite FIFO-compliant trace is always complete. This makes sense because we cannot check if the channels are empty at the end of an infinite trace.

Lemma 2.5. Let $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. For any run $(\vec{q}_0, \xi_0) \xrightarrow{x_0} \cdots \xrightarrow{x_n} (\vec{q}, \xi)$ with trace $w = x_0 \ldots x_n$, it holds that

(1) $\xi((\mathtt{p}, \mathtt{q})) = u$ where $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u$ for every pair of participants $\mathtt{p}, \mathtt{q} \in \mathcal{P}$ with $\mathtt{p} \neq \mathtt{q}$,

(2) w is FIFO-compliant, and

(3) maximal traces of $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are complete.

Proof. We prove (1) by induction on the trace w of a run.

The base case where $w = \varepsilon$ is trivial. For the induction step, we consider wx with the following run in $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$: $(\vec{q}_0, \xi_0) \xrightarrow{w}{}^{*} (\vec{q}, \xi) \xrightarrow{x} (\vec{q}\,', \xi')$. The induction hypothesis holds for w and $(\vec{q}, \xi)$ and we prove the claims for $(\vec{q}\,', \xi')$ and wx. We do a case analysis on x. If $x = \varepsilon$, the claim trivially follows.

Let $x = \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m}$. From the induction hypothesis, we know that $\xi((\mathtt{p}, \mathtt{q})) = u$ where $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u$. Since $x = \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m}$ is a possible transition, we know that $u = \mathsf{m} \cdot u'$ for some u' and $\xi'((\mathtt{p}, \mathtt{q})) = u'$. Thus, we have

$$\begin{aligned}
\mathcal{V}((wx)\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) &= \mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) \\
&= \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u \\
&= \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot \mathsf{m} \cdot u' \\
&= \mathcal{V}((wx)\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u' \ .
\end{aligned}$$

For all other pairs of participants, the induction hypothesis applies since the above projections do not change:

$$\mathcal{V}(w\Downarrow_{\mathtt{r}\triangleright\mathtt{s}!_}) = \mathcal{V}((wx)\Downarrow_{\mathtt{r}\triangleright\mathtt{s}!_}) \quad \text{and} \quad \mathcal{V}(w\Downarrow_{\mathtt{s}\triangleleft\mathtt{r}?_}) = \mathcal{V}((wx)\Downarrow_{\mathtt{s}\triangleleft\mathtt{r}?_})$$

for $\mathtt{r}, \mathtt{s} \in \mathcal{P}$.

Let $x = \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$. From the induction hypothesis, we know that $\xi((\mathtt{p}, \mathtt{q})) = u$ where $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u$. Since x is a send event, we know that $\xi'((\mathtt{p}, \mathtt{q})) = u \cdot \mathsf{m}$. By definition and induction hypothesis, we have: $\mathcal{V}((wx)\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) \cdot \mathsf{m} = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \cdot u \cdot \mathsf{m}$. As for the previous case, for all other combinations of participants, the induction hypothesis applies since the above projections do not change. Together, this proves (1).

For (2), we observe from the reasoning for (1) immediately that, for every prefix w' of w, it holds that

$$\mathcal{V}(w'\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) \leq \mathcal{V}(w'\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_}) \ .$$

For (3), it is straightforward that any maximal trace is FIFO-compliant from (2). For a finite maximal word w, we know that all channels are empty and, thus, with (1), it holds that $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleright\mathtt{q}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleleft\mathtt{p}?_})$ for every pair of participants $\mathtt{p}$ and $\mathtt{q}$, making w complete. An infinite maximal word w is trivially complete. □

2.4 Indistinguishability Relation

Example 2.6 (Motivation for indistinguishability relation $\sim$)**.** In Example 2.2, we considered a trace for the CSM in Figure 2.1. Let us consider the trace where $\mathtt{p}$ only sends a single element and $\mathtt{q}$ receives the message as soon as it is available:

$$\mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{nil} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{nil} \cdot \mathtt{q}\triangleright\mathtt{p}!\mathsf{ack} \cdot \mathtt{p}\triangleleft\mathtt{q}?\mathsf{ack} \ .$$

Every CSM that admits the previous trace also admits the following one:

$$\mathtt{p}\triangleright\mathtt{q}!\mathsf{cons} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{nil} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{cons} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{nil} \cdot \mathtt{q}\triangleright\mathtt{p}!\mathsf{ack} \cdot \mathtt{p}\triangleleft\mathtt{q}?\mathsf{ack}$$

where the 2nd and 3rd event were swapped. Both belong to different participants and they can act independently, allowing $\mathtt{p}$ to send its second message before $\mathtt{q}$ receives the first one. However, one could not swap the 5th and 6th event: they are corresponding send and receive events. ◀

Intuitively, one can swap the events of different participants as long as they are not corresponding send and receive events and the trace will still be admitted by the same CSM. Similarly, two events can be swapped if they are not related by the *happened-before* relation [82]. Two traces that are obtained by swapping such independent events can be considered *indistinguishable* in terms of language membership for a CSM: either both are in the language or both are not.

Let us formalise this phenomenon.

Definition 2.7. The *indistinguishability relation* $\sim \subseteq \Gamma^* \times \Gamma^*$ is the smallest equivalence relation such that

(1) If $p \neq r$, then $w \cdot p \triangleright q!m \cdot r \triangleright s!m' \cdot u \;\sim\; w \cdot r \triangleright s!m' \cdot p \triangleright q!m \cdot u$.
(2) If $q \neq s$, then $w \cdot q \triangleleft p?m \cdot s \triangleleft r?m' \cdot u \;\sim\; w \cdot s \triangleleft r?m' \cdot q \triangleleft p?m \cdot u$.
(3) If $p \neq s \wedge (p \neq r \vee q \neq s)$, then $w \cdot p \triangleright q!m \cdot s \triangleleft r?m' \cdot u \;\sim\; w \cdot s \triangleleft r?m' \cdot p \triangleright q!m \cdot u$.
(4) If $|w \Downarrow_{p \triangleright q!_}| > |w \Downarrow_{q \triangleleft p?_}|$, then $w \cdot p \triangleright q!m \cdot q \triangleleft p?m' \cdot u \;\sim\; w \cdot q \triangleleft p?m' \cdot p \triangleright q!m \cdot u$.

We define $u \preceq_\sim v$ if there is $w \in \Gamma^*$ such that $u \cdot w \sim v$. Observe that $u \sim v$ iff $u \preceq_\sim v$ and $v \preceq_\sim u$. To extend $\sim$ to infinite words, we follow the approach of Gastin [50]. For infinite words $u, v \in \Gamma^\omega$, we define $u \preceq^\omega_\sim v$ if for each finite prefix u' of u, there is a finite prefix v' of v such that $u' \preceq_\sim v'$. Define $u \sim v$ iff $u \preceq^\omega_\sim v$ and $v \preceq^\omega_\sim u$.
We lift the equivalence relation $\sim$ on words to languages. For a language L, we define

$$\mathcal{C}^\sim(L) = \left\{ w' \;\middle|\; \bigvee \begin{array}{l} w' \in \Gamma^* \wedge \exists w \in \Gamma^*.\, w \in L \text{ and } w' \sim w \\ w' \in \Gamma^\omega \wedge \exists w \in \Gamma^\omega.\, w \in L \text{ and } w' \preceq^\omega_\sim w \end{array} \right\}.$$

The definition is split into two cases. For a finite word w', we require that there is a word w in L such that they are indistinguishable. For the infinite case, we use $\preceq^\omega_\sim$. Unlike the closure operator by Gastin [50, Def. 2.1], $\preceq^\omega_\sim$ is asymmetric. Let us briefly explain why. Consider the protocol $(p \triangleright q!m\,.\,q \triangleleft p?m)^\omega$. Since we do not make any fairness assumption on scheduling, we need to include in the closure the execution where only the sender is scheduled, i.e. $(p \triangleright q!m)^\omega \preceq^\omega_\sim (p \triangleright q!m.\; q \triangleleft p?m)^\omega$.

Example 2.8 (Indistinguishability relation $\sim$ by example)**.** The four rules of $\sim$ present conditions under which two adjacent events in a trace can be reordered. These conditions are designed in a way that they characterise possible changes in a trace that cannot be distinguished by any CSM. To be precise, if w is recognised by some CSM $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ and $w' \sim w$ holds, then w' is also recognised by $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$. The order of local state changes of participants differs though. In this example, we illustrate the intuition behind these rules and assume that variables do not alias, i.e. two participants or messages with different names are different.

Two send events (or two receive events) can be swapped if the active participants are distinct because there cannot be any dependency between two such events which do occur next to each other in an execution. For send events, the 1st rule, thus, admits $p \triangleright r!m_1 \cdot q \triangleright r!m_2 \sim q \triangleright r!m_2 \cdot p \triangleright r!m_1$ even though the receiver is the same. In contrast, the corresponding receive events cannot be swapped: $r \triangleleft p?m_1 \cdot r \triangleleft q?m_2 \not\sim r \triangleleft q?m_2 \cdot r \triangleleft p?m_1$. Note that the 1st rule is the only one with which two send events can be swapped while the 2nd rule is the only one for receive events so indeed no rule applies for the two receive events from before.

The 3rd rule allows one send and one receive event to be swapped if either both senders or both receivers are different – in addition to the requirement that both active participants are different. For instance, it admits $\mathtt{p}\triangleright\mathtt{r}!\mathsf{m} \cdot \mathtt{q}\triangleleft\mathtt{r}?\mathsf{m} \sim \mathtt{q}\triangleleft\mathtt{r}?\mathsf{m} \cdot \mathtt{p}\triangleright\mathtt{r}!\mathsf{m}$. However, it does not admit to swap: $\mathtt{p}\triangleright\mathtt{q}!\mathsf{m} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m} \not\sim \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$. This is reasonable since the send event could be the one which emits m in the corresponding channel. In this trace, this is, in fact, the case because there are no earlier events, but in general one needs to incorporate the context to understand whether this is the case. The 4th rule does this and, hence, admits swapping the same events when appended to some prefix: $\mathtt{p}\triangleright\mathtt{q}!\mathsf{m} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{m} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m} \not\sim \mathtt{p}\triangleright\mathtt{q}!\mathsf{m} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m} \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$. With the prefix, the FIFO order of channels ensures that the first message will be received first and the second send event can happen after the reception of the first message. ◀

If two words of events are both FIFO-compliant, they are related by $\sim$ if and only if the projected word for each participant is the same.

Lemma 2.9. Let $w \in \Gamma^\infty$ be FIFO-compliant. Then, $w \sim w'$ iff w' is FIFO-compliant and $w{\Downarrow}_{\Gamma_\mathtt{p}} = w'{\Downarrow}_{\Gamma_\mathtt{p}}$ for every participant $\mathtt{p} \in \mathcal{P}$.

Proof. We use the characterisation of $\sim$ using dependence graphs [50]. For a word w and a letter $a \in \Gamma$ that appears in w, let (a, i) denote the ith occurrence of a in w. Define the dependence graph (V, E, λ), where $V = \{(a, i) \mid a \in \Gamma, i \geq 1\}$, $E = \{((a, i), (b, j)) \mid a$ and b cannot be swapped and (a, i) occurs before (b, j) in $w\}$, and $\lambda(a, i) = a$ for all $a \in \Gamma$, $i \geq 1$. A fundamental result of trace theory states that $w \sim w'$ iff they have isomorphic dependence graphs. We observe that for two FIFO-compliant words, the ordering of the letters on each $\Gamma_\mathtt{p}$ for $\mathtt{p} \in \mathcal{P}$ ensures isomorphic dependence graphs, since the ordering of receives is thus fixed. ☐

We motivated the term *indistinguishability* with the fact that CSMs cannot distinguish two words that are related by $\sim$ in terms of language membership. Now, we prove this claim and show their languages are closed under $\sim$. In fact, we strengthen this point and show that a CSM will reach the same configuration after processing two $\sim$-related traces.

Lemma 2.10. Let $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. Then, for every finite w with a run in $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ and every $w' \sim w$, w' has a run that ends with the same configuration. The language $\mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}})$ is closed under $\sim$, i.e. $\mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{C}^\sim(\mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$.

Proof. Let w be a finite word with a run in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ and $w' \sim w$. For simplicity, we index $\sim$ to indicate the number of times events were swapped. Then, $\sim$ can be applied n times to w to obtain w' for some n. We prove that w' has a run that ends in the same configuration by induction on n. The base case for $n = 0$ is trivial. For the induction step, we assume that the claim holds for n and prove it for $n+1$. Suppose that $w \sim_{n+1} w'$. Then, there is w'' such that $w' \sim_1 w''$ and $w'' \sim_n w$. By assumption, we know that $w' = u'xyu''$ and $w'' = u'yxu''$ for some $u', u'' \in \Gamma^*$, $x, y \in \Gamma$. By induction hypothesis, we know that $w'' \in \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}})$ so there is run for w'' in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$. Let us investigate the run at x and y: $\cdots(\vec{q}_1, \xi_1) \xrightarrow{x} (\vec{q}_2, \xi_2) \xrightarrow{y} (\vec{q}_3, \xi_3)\cdots$. It suffices to show that $\cdots(\vec{q}_1, \xi_1) \xrightarrow{y} (\vec{q}\,'_2, \xi'_2) \xrightarrow{x} (\vec{q}_3, \xi_3)\cdots$ is possible in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ for some configuration $(\vec{q}\,'_2, \xi'_2)$. We do a case analysis on the rule that was applied for $w' \sim_1 w''$:

(1) $x = p\triangleright q!m$, $y = r\triangleright s!m'$, and $p \neq r$:
We define $\vec{q}\,'_2$ such that $\vec{q}\,'_{2,p} = \vec{q}_{1,p}$, $\vec{q}\,'_{2,r} = \vec{q}_{3,r}$, and $\vec{q}\,'_{2,t} = \vec{q}_{3,t}$ for all $t \in \mathcal{P}$ with $t \neq p$ and $t \neq r$. Both transitions are feasible in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ because both p and r are different and send a message to different channels. They can do this independently from each other.

(2) $x = q\triangleleft p?m$, $y = s\triangleleft r?m'$, and $q \neq s$:
We define $\vec{q}\,'_2$ such that $\vec{q}\,'_{2,q} = \vec{q}_{1,q}$, $\vec{q}\,'_{2,s} = \vec{q}_{3,s}$, and $\vec{q}\,'_{2,t} = \vec{q}_{3,t}$ for all $t \in \mathcal{P}$ with $t \neq p$ and $t \neq r$. Both transitions are feasible in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ because both q and s are different and receive a message from a different channel. They can do this independently from each other.

(3) $x = p\triangleright q!m$, $y = s\triangleleft r?m'$, and and $p \neq s \wedge (p \neq r \vee q \neq s)$.
We define $\vec{q}\,'_2$ such that $\vec{q}\,'_{2,p} = \vec{q}_{1,p}$, $\vec{q}\,'_{2,s} = \vec{q}_{3,s}$, and $\vec{q}\,'_{2,t} = \vec{q}_{3,t}$ for all $t \in \mathcal{P}$ with $t \neq p$ and $t \neq r$. Let us do a case split according to the side conditions.
First, let $p \neq s$ and $p \neq r$. The channels of x and y are different and p and s are different, so both transitions are feasible in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$.
Second, let $p \neq s$ and $q \neq s$. The channels of x and y are different and q and s are different, so both transitions are feasible in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$.

(4) $x = p\triangleright q!m$, $y = q\triangleleft p?m'$, and $|u'\Downarrow_{p\triangleright q!_}| > |u'\Downarrow_{q\triangleleft p?_}|$:
We define $\vec{q}\,'_2$ such that $\vec{q}\,'_{2,p} = \vec{q}_{1,p}$, $\vec{q}\,'_{2,q} = \vec{q}_{3,q}$, and $\vec{q}\,'_{2,t} = \vec{q}_{3,t}$ for all $t \neq p$ and $t \neq q$. In this case, the channel of x and y is the same but the side condition ensures that y actually has a different message to receive since the channel $\xi_1((p,q))$ is not empty by Lemma 2.5 and, hence, both transitions can act independently and lead to the same configuration.

This proves that w' has a run in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ that ends in the same configuration which concludes the proof of the first claim.

For the second claim, we know that $\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}))$ by definition. Hence, it suffices to show that $\mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}})) \subseteq \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}})$.

We show the claim for finite traces first:

$$\mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}})) \cap \Gamma^* \subseteq \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^*.$$

Let $w' \in \mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}})) \cap \Gamma^*$. There is $w \in \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^*$ such that $w \sim w'$. By definition, w has a run in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ which ends in a final configuration. From the first claim, we know that w' also has a run that ends in the same configuration, which is final. Therefore, $w \in \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^*$. Hence, the claim holds for finite traces.

It remains to show the claim for infinite traces. To this end, we first establish the following fact $(*)$: for every trace w of $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ such that $w \sim u$ for $u \in \text{pref}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}))$ and any continuation x of w, i.e. wx is a trace of $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$, it holds that $wx \sim ux$ and $ux \in \text{pref}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}))$ $(*)$. We know $w \sim u$, so $wx \sim ux$ because the same swaps can be mimicked when extending both w and u by x. From the first claim, we know that $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ is in the same configuration (q, ξ) after processing w and u. Therefore, ux is a trace of $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ because wx is, proving $(*)$.

We now show that

$$\mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^\omega \subseteq \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^\omega.$$

Let $w \in \mathcal{C}^{\sim}(\mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^\omega$. We show that w has an infinite run in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$.

Consider a tree $\mathcal{T}$ where each node corresponds to a run ρ on some finite prefix $w' \leq w$ in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$. The root is labelled by the empty run. The children of a node ρ are runs that extend ρ by a single transition – these exist by $(*)$. Since our CSM is built from a finite number of finite state machines, $\mathcal{T}$ is finitely branching. By König's Lemma [80, 79], there is an infinite path in $\mathcal{T}$ that corresponds to an infinite run for w in $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$, so $w \in \mathcal{L}(\{\!\{A_p\}\!\}_{p\in\mathcal{P}}) \cap \Gamma^\omega$. □

2.5 High-level Message Sequence Charts

In this section, we define the syntax and semantics of HMSCs, following work by Genest et al. [52]. We also show that HMSCs are closed under $\sim$.

To start with, let us define message sequence charts as building blocks for HMSCs.

Definition 2.11 ((Prefix) Message Sequence Charts)**.** A *prefix message sequence chart* is a 5-tuple $M = (N, p, f, l, (\leq_p)_{p\in\mathcal{P}})$ where

- N is a set of send (S) and receive (R) event nodes such that $N = S \uplus R$ (where $\uplus$ denotes disjoint union),
- $p\colon N \to \mathcal{P}$ maps each event node to the participant acting on it,
- $f\colon S \rightharpoonup R$ is a partial function that is injective and surjective, linking send and receive event nodes,
- $l\colon N \to \Gamma$ labels every event node with an event, and

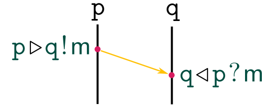


Figure 2.2: Highlighting the elements of an MSC $(N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}})$.

- $(\leq_{\mathtt{p}})_{\mathtt{p}\in\mathcal{P}}$ is a family of total orders for the event nodes of each participant: $\leq_{\mathtt{p}} \subseteq p^{-1}(\mathtt{p}) \times p^{-1}(\mathtt{p})$.

We visually highlight the elements of an MSC in Figure 2.2. For readability, we will omit the event labels but label the arrow with the respective message instead.
A prefix MSC M induces a partial order $\leq_M$ on N that is defined co-inductively:

$$\frac{e \leq_{\mathtt{p}} e'}{e \leq_M e'}\ \text{PART} \qquad \frac{s \in S}{s \leq_M f(s)}\ \text{SND-RCV} \qquad \frac{}{e \leq_M e}\ \text{REFL} \qquad \frac{e \leq_M e' \quad e' \leq_M e''}{e \leq_M e''}\ \text{TRANS}$$

The labelling function l respects the functions p and f: for every send event node e, we have that $l(e) = p(e) \triangleright p(f(e))!\mathsf{m}$ and $l(f(e)) = p(f(e)) \triangleleft p(e)?\mathsf{m}$ for some $\mathsf{m} \in \mathcal{V}$ if $f(e)$ is defined and $l(e) = p(e) \triangleright \mathtt{q}!\mathsf{m}$ for some $\mathsf{m} \in \mathcal{V}$ and $\mathtt{q} \in \mathcal{P} \setminus \{p(e)\}$ if not.

We say an MSC respects FIFO order if, for every two participants $\mathtt{p}$ and $\mathtt{q}$, for all $e_1, e_2 \in p(\mathtt{p})$ with $f(e_1) \in p(\mathtt{q})$, $f(e_2) \in p(\mathtt{q})$ and $e_1 \leq_{\mathtt{p}} e_2$, one of the following holds: $f(e_1) \leq_{\mathtt{q}} f(e_2)$ or if $f(e_1)$ is undefined then $f(e_2)$ is undefined. In this thesis, we do only consider MSCs that respect FIFO order. If f is total, we omit the term *prefix* and call M a message sequence chart (MSC). A *basic MSC* (BMSC) has a finite number of nodes N and $\mathcal{M}$ denotes the set of all BMSCs. When unambiguous, we omit the index M for $\leq_M$ and write $\leq$. We define $\lneq$ as expected. The language $\mathcal{L}(M)$ of an MSC M collects all words $l(w)$ for which w is a linearisation of N with regard to $\leq_M$.

Remark 2.12. Let us explain how our conditions for MSCs ensure FIFO order. Since f is a surjective function, there is no receive event node for which f^{-1} is undefined. This ensures that every receive event node is matched. This is important, because, in a prefix MSC, there can be send event nodes for which no corresponding receive event node exists but not vice versa. The condition for each channel has two parts. The first part ensures that messages do not cross while the second one ensures that the receive event nodes are matched against the earliest send event nodes.

In the definition of MSCs, event nodes are labelled. For conciseness, we will label the arrow representing f in visualisations with the message because sender and receiver will always be clear from context.

For every FIFO-compliant word w, one can construct a unique prefix MSC M such that w is a linearisation of M.

Lemma 2.13 (msc(-) ([52], Sec. 3.1))**.** Let $w \in \Gamma^\infty$ be a FIFO-compliant word. Then, there is unique prefix MSC, denoted by $\mathrm{msc}(w)$, such that w is a linearisation of $\mathrm{msc}(w)$. In case the above conditions are not satisfied, $\mathrm{msc}(w)$ is undefined.

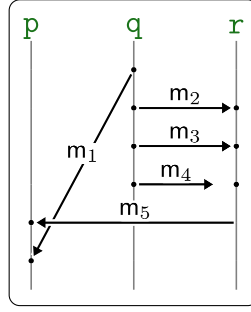


Figure 2.3: Constructing a prefix MSC from a FIFO-compliant word.

The operator $\mathrm{msc}(\text{-})$ allows us to easily represent a trace of a run as MSC.

Example 2.14. Let us consider the FIFO-compliant word

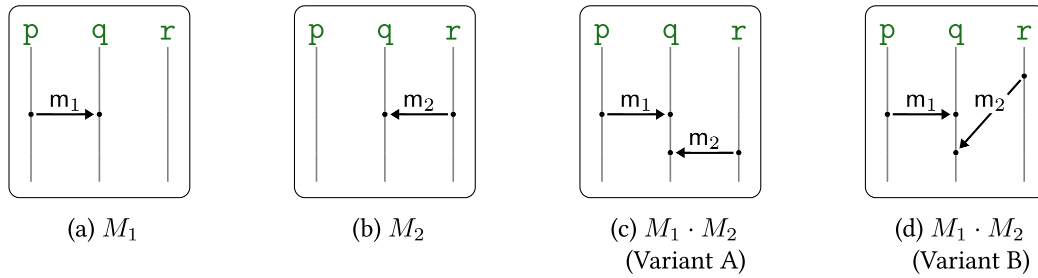
$$w := \mathtt{q}\triangleright\mathtt{p}!\mathsf{m}_1 \cdot \mathtt{q}\triangleright\mathtt{r}!\mathsf{m}_2 \cdot \mathtt{q}\triangleright\mathtt{r}!\mathsf{m}_3 \cdot \mathtt{q}\triangleright\mathtt{r}!\mathsf{m}_4 \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathsf{m}_2 \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathsf{m}_3 \cdot \mathtt{r}\triangleright\mathtt{p}!\mathsf{m}_5 \cdot \mathtt{p}\triangleleft\mathtt{r}?\mathsf{m}_5 \cdot \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m}_1 \ .$$

In Fig. 2.3, we illustrate $\mathrm{msc}(w)$. We use horizontal arrows for messages if possible, in contrast to Fig. 2.2. This is standard and beneficial for readability and conciseness. Still, we consider asynchronous communication: In w, message m_3 is sent before m_2 in the same channel has been received. In the MSC, this dependency is not present so it admits a linearisation where m_2 is received first. One can see that all receive event nodes have a corresponding send event node. However, the send event node for m_4 has no receive event node. Still, our conditions ensure that the messages m_2 and m_3 are matched first, leaving the last message m_4 unmatched. In w, the message m_1 is sent first and received last. This shows that we consider a point-to-point setting where FIFO order is only enforced between pairs of participants. Note, though, that there is no dependency between sending m_4 and receiving m_1 so these events could be reordered in w. We will show that these missing dependencies precisely characterise the reorderings that are possible with $\sim$. ◀

One can also concatenate prefix MSCs M_1 and M_2. We restrict to the case where M_1 is a basic MSC and M_2 is an MSC so we do not consider unmatched send events and the number of event nodes is finite for M_1. One can define more general concatenation operators, with additional conditions but this suffices for our purposes. Let us define concatenation formally prior to giving an example.

Definition 2.15 (MSC Concatenation). Let $M_i = (N_i, p_i, f_i, l_i, (\leq^i_p)_{\mathtt{p}\in\mathcal{P}})$ for $i \in \{1,2\}$ such that M_1 is a BMSC and M_2 is an MSC with disjoint sets of event nodes: $N_1 \cap N_2 = \emptyset$. We define their *concatenation* $M_1 \cdot M_2$ as the MSC $M = (N, p, f, l, (\leq_p)_{\mathtt{p}\in\mathcal{P}})$ where

- $N \ := \ N_1 \cup N_2$,
- for $\zeta \in \{p, f, l\}$: $\quad \zeta(e) := \begin{cases} \zeta(e) & \text{if } e \in N_1 \\ \zeta(e) & \text{if } e \in N_2 \end{cases}$, and
- for each $\mathtt{p} \in \mathcal{P}$:
  $\leq_{\mathtt{p}} := \leq^1_{\mathtt{p}} \cup \leq^2_{\mathtt{p}} \cup \{(e_1, e_2) \mid e_1 \in N_1 \wedge e_2 \in N_2 \wedge p(e_1) = p(e_2) = \mathtt{p}\}$.



(a) M_1 (b) M_2 (c) $M_1 \cdot M_2$ (Variant A) (d) $M_1 \cdot M_2$ (Variant B)

Figure 2.4: Concatenating MSCs.

Example 2.16. Let us consider the BMSCs M_1 and M_2 in Fig. 2.4a and Fig. 2.4b. We give two visual representations of their concatenation: Variant A in Fig. 2.4c and Variant B in Fig. 2.4d. Both represent isomorphic MSCs. Still, Variant A is closer to what one expects. In both, the sequence of events for q exemplifies that events for each participant are ordered sequentially. However, this applies to individual participants only. The send event of r from M_2 does not depend on any other event in $M_1 \cdot M_2$. This is why it is possible to move it to the top, as depicted in Variant B. ◀

If one thinks of a BMSC as straight-line code, a high-level message sequence chart adds control flow. It embeds BMSCs into a graph structure, which permits to specify branching and recursion.

Definition 2.17 (High-level Message Sequence Charts)**.** A *high-level message sequence chart* (HMSC) is a 5-tuple (V, E, v^I, V^T, μ) where V is a finite set of vertices, $E \subseteq V \times V$ is a set of directed edges, $v^I \in V$ is an initial vertex, $V^T \subseteq V$ is a set of final vertices, and $\mu : V \to \mathcal{M}$ is a function mapping every vertex to a BMSC. A path in an HMSC is a sequence of vertices $v_1, \ldots$ from V that is connected by edges, i.e. $(v_i, v_{i+1}) \in E$ for every i. A path is *maximal* if it is infinite or ends in a vertex from V^T.

Example 2.18. In Example 2.2, we considered a CSM where p sends a list of elements to q. The same protocol can be specified as HMSC and is illustrated in Figure 2.5. The protocol starts at the initial vertex at the top. There, p decides the branch by sending cons or nil to q. In the former case, the protocol recurses. In the latter case, q replies with ack and the protocol terminates. ◀

Intuitively, the language of an HMSC is the union of all languages of the finite and infinite MSCs generated from maximal paths in the HMSC.

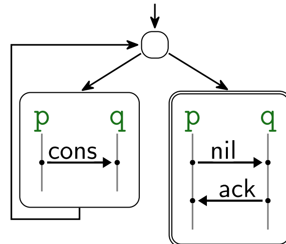


Figure 2.5: Sending a list, specified as HMSC.

Definition 2.19 (Language of an HMSC)**.** Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. The language of H is defined as

$$\begin{aligned}\mathcal{L}(H) := \{w \mid w \in \mathcal{L}(\mu(v_1) \cdot \mu(v_2) \cdot \ldots \cdot \mu(v_n)) \text{ with} \\ v_1 = v^I \wedge \forall\, 0 < i < n :\, (v_i, v_{i+1}) \in E \wedge v_n \in V^T\} \\ \cup\ \{w \mid w \in \mathcal{L}(\mu(v_1)\mu(v_2)\ldots) \text{ with } v_1 = v^I \wedge \forall\, i > 0 :\, (v_i, v_{i+1}) \in E\}\ .\end{aligned}$$

We say that an MSC from a maximal path in H is an MSC of H.

It is easy to check that HMSCs specify FIFO-compliant complete words.

Proposition 2.20. The language $\mathcal{L}(H)$, respectively $\mathcal{L}(M)$, for any HMSC H, respectively MSC M, is a set of FIFO-compliant complete words.

Furthermore, $\sim$ captures exactly the events that are independent in any HMSC. Phrased differently, HMSC include these reorderings by design.

Lemma 2.21. Let H be any HMSC. Then, $\mathcal{L}(H) = \mathcal{C}^{\sim}(\mathcal{L}(H))$.

Proof. We prove the claim by two inclusions. The first inclusion $\mathcal{L}(H) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(H))$ trivially holds. We show that $\mathcal{C}^{\sim}(\mathcal{L}(H)) \subseteq \mathcal{L}(H)$.

As for the proof of Lemma 2.10, we index the indistinguishability relation $\sim$ to indicate the number of reorderings. We show that word membership is preserved for one reordering.

Claim I: Let M be some MSC of H. Let w be a sequence of events such that there is $w' \in \mathcal{L}(M)$ with $w \sim_1 w'$. Then, $w \in \mathcal{L}(M)$.

Proof of Claim I. We do a case analysis on the rule of $\sim$ which has been applied to obtain $w \sim_1 w'$. In all cases, it is crucial to observe that two consecutive events are swapped and hence transitive dependencies in $\leq_M$ cannot kick in and the events either have to be ordered by the total participant orders or constitute send-reception pairs.

(1) Here, two send events for different participants are swapped. These two events cannot be ordered by $\leq_M$ (without some intermediate receive event which is not present) and therefore $w \in \mathcal{L}(M)$.

(2) In this case, two receive events for different participants are swapped. Again, they cannot be ordered by $\leq_M$ without some intermediate send event and the claim follows.

(3) This case deals with swapping a send event $\mathtt{p}{\triangleright}\mathtt{q}!\mathsf{m}$ and a receive event $\mathtt{s}{\triangleleft}\mathtt{r}?\mathsf{m}'$. The first condition $\mathtt{p} \neq \mathtt{s}$ ensures that both events do not belong to the same participant. However, this does not suffice. We do a case analysis according to the disjunction of the conditions.
First, $\mathtt{p} \neq \mathtt{r}$ entails that the sender of the send event is neither the sender nor the receiver of the receive event. Then, both events cannot be related by the participant order but they do also not constitute a send-reception pair (ordered through the function f that matches send with receive events) and hence the claim follows.

Second, $\mathtt{q} \neq \mathtt{s}$ entails that the receiver of the send event and the receiver of the receive event are not the same. This ensures that we do not swap a receive event in front of its corresponding send event. Again, they cannot be ordered by $\leq_M$.

(4) This case also deals with swapping a send event $\mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$ and a receive event $\mathtt{q}\triangleleft\mathtt{p}?\mathsf{m}'$. Note that m might be the same as m' here and the side condition is needed to ensure that both do not constitute a pair of send and receive event so that they were ordered through the function f that matches send with receive events.

End Proof of Claim I.

We first consider the case of finite words. Let $u \in \mathcal{C}^{\sim}(\mathcal{L}(H)) \cap \Gamma^*$. We show that $u \in \mathcal{L}(H)$. By definition, there is an MSC M of H such that $u \in \mathcal{C}^{\sim}(\mathcal{L}(M)) \cap \Gamma^*$. Therefore, there is $u' \in \mathcal{L}(M)$ such that $u' \sim_n u$ for some n. By applying *Claim I* n-times, the claim follows.

Second, let $u \in \mathcal{C}^{\sim}(\mathcal{L}(H)) \cap \Gamma^\omega$. By definition, there is an MSC M of H such that $u \in \mathcal{C}^{\sim}(\mathcal{L}(M)) \cap \Gamma^\omega$. Therefore, there is $v \in \mathcal{L}(M)$ such that $u \preceq^\omega_\sim v$ which means that for every prefix u' of u, there is some prefix v' of v such that $u' \preceq v'$, i.e. there is some w such that $u'w \sim v'$. From the case for finite sequences, it follows that $u' \in \operatorname{pref}(\mathcal{L}(M))$ for every finite prefix u' of u. Note that there is a single (infinite) MSC M for which this applies. Therefore, the path of every prefix u' in H is the same and can be extended for longer prefixes. Since every prefix u' of u is in $\operatorname{pref}(\mathcal{L}(M))$, the infinite sequence u is in $\mathcal{L}(M)$ and hence in $\mathcal{L}(H)$. □

2.6 Global Types from Multiparty Session Types

In this section, we define the syntax and semantics of global types for our Multiparty Session Type (MST) framework.

We give the syntax of global types following work by Honda et al. [67], Hu and Yoshida [70], as well as Scalas and Yoshida [109]. We focus on the core message-passing aspects of asynchronous MSTs and do not include features such as delegation or subsessions.

Definition 2.22 (Syntax of global types)**.** *Global types* for MSTs are defined by the grammar:

$$G ::= 0 \;\mid\; \sum_{i\in I} \mathtt{p}_i \to \mathtt{q}_i : \mathsf{m}_i \,.\, G_i \;\mid\; \mu t \,.\, G \;\mid\; t$$

The term 0 explicitly represents termination. A term $\mathtt{p}_i \to \mathtt{q}_i : \mathsf{m}_i$ indicates an interaction where the sender $\mathtt{p}_i$ sends message m_i to the receiver $\mathtt{q}_i$. We assume both the sender and receiver of an interaction are different. The sum operator denotes choice between a set of branches, given by a finite index set I. We assume $|I| > 0$ and, if $|I| = 1$, we omit the sum operator. The operators μt and t allow to encode loops. We require them to be guarded, i.e. there must be at least one interaction between the binding μt and the

use of the recursion variable t. Without loss of generality, all occurrences of recursion variables t are bound and distinct. For the size of a global type, we disregard multiple occurrences of the same subterm. We introduce restrictions on choice, i.e. who decides which branch to take. It is common to require a dedicated sender to choose a branch: $\forall i, j \in I \,.\, \mathsf{p}_i = \mathsf{p}_j$ for every syntactic subterm $\sum_{i \in I} \mathsf{p}_i \rightarrow \mathsf{q}_i : \mathsf{m}_i \,.\, G_i$. Then, *sender-driven choice* allows a sender to send to different receivers upon branching[1] while *directed choice* [68] also requires the receiver to be the same: $\forall i, j \in I \,.\, \mathsf{q}_i = \mathsf{q}_j$ (and $\forall i, j \in I \,.\, \mathsf{p}_i = \mathsf{p}_j$). If there is no dedicated sender but all transitions are still distinct, we say the global type satisfies *mixed-choice*: for every syntactic subterm $\sum_{i \in I} \mathsf{p}_i \rightarrow \mathsf{q}_i : \mathsf{m}_i \,.\, G_i$, it holds that $\forall i, j \in I \,.\, i \neq j \implies \mathsf{p}_i \neq \mathsf{p}_j \vee \mathsf{q}_i \neq \mathsf{q}_j \vee \mathsf{m}_i \neq \mathsf{m}_j$. Without restrictions, we call them *non-deterministic* global types. Subsequently, *global types* always satisfy sender-driven choice, unless explicitly mentioned.

Example 2.23 (Global type)**.** Let us specify our list-sending example as global type:

$$G_{\text{LiSe}} \;:=\; \mu t \,.\, + \begin{cases} \mathsf{p} \rightarrow \mathsf{q} : \mathsf{cons} \,.\, t \\ \mathsf{p} \rightarrow \mathsf{q} : \mathsf{nil} \,.\, \mathsf{q} \rightarrow \mathsf{p} : \mathsf{ack} \,.\, 0 \end{cases} \quad .$$

It first binds a recursion operator t, which is used in the top branch. The bottom branch does not use t but allows to terminate the protocol. ◀

A global type always jointly specifies send and receive events. In a CSM execution, there may be independent events that can occur between a send and its respective receive event. Hence, we define the semantics of global types using the indistinguishability relation $\sim$.

Definition 2.24 (Semantics of global types)**.** We construct a state machine $\mathsf{GAut}(G)$ to obtain the semantics of a global type G. We index every syntactic subterm of G with a unique index to distinguish common syntactic subterms, denoted with $[G', k]$ for syntactic subterm G' and index k. Without loss of generality, the index for G is 1: $[G, 1]$. For readability, we do not quantify indices. We define

$$\mathsf{GAut}(G) = (Q_{\mathsf{GAut}(G)}, \Sigma, \delta_{\mathsf{GAut}(G)}, q_{0,\mathsf{GAut}(G)}, F_{\mathsf{GAut}(G)}) \text{ where}$$

- $Q_{\mathsf{GAut}(G)}$ is the set of all indexed syntactic subterms $[G', k]$ of G
- $\delta_{\mathsf{GAut}(G)}$ is the smallest set containing $([\sum_{i \in I} \mathsf{p}_i \rightarrow \mathsf{q}_i : \mathsf{m}_i \,.\, [G_i, k_i], k], \mathsf{p}_i \rightarrow \mathsf{q}_i : \mathsf{m}_i, [G_i, k_i])$ for each $i \in I$, and $([\mu t \,.\, [G', k_2'], k_1'], \varepsilon, [G', k_2'])$ and $([t, k_3'], \varepsilon, [\mu t \,.\, [G', k_2'], k_1'])$,
- $q_{0,\mathsf{GAut}(G)} = [G, 1]$, and $F_{\mathsf{GAut}(G)} = \{[0, k] \mid k \text{ is an index for subterm } 0\}$.

[1] Hu and Yoshida introduce a similar choice restriction, which they call located choice. However, they require that "[t]his decision [...] must be appropriately coordinated by explicit messages" [71, p. 4], which we do not require in our developments.

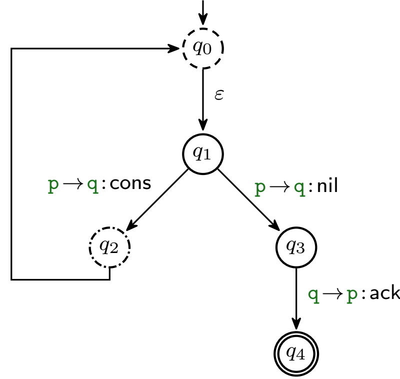


Figure 2.6: List-sending Protocol: FSM for the semantics of G_{LiSe}; styles of states indicate their kind, e.g. recursion states (dashed lines) while final states have double lines.

Each interaction is implicitly split into its send and receive event from Γ. In addition, we consider CSMs as implementation model for global types and, from Lemma 2.10, we know that CSM languages are always closed under the indistinguishability relation $\sim$. Thus, we also apply its closure to obtain the semantics of G:

$$\mathcal{L}(G) := \mathcal{C}^{\sim}(\mathcal{L}(\mathsf{GAut}(G))) \ .$$

The closure $\mathcal{C}^{\sim}(\text{-})$ corresponds to similar reordering rules in standard MST developments, e.g. [68, Def. 3.2 and 5.3]. It is straightforward that any word in $\mathcal{L}(\mathsf{GAut}(G))$ for some G is FIFO-compliant and complete. The application of $\mathcal{C}^{\sim}(\text{-})$ does not change this.

Proposition 2.25. For any global type G, every word in $\mathcal{L}(G)$ is FIFO-compliant and complete.

No global type different from 0 can specify ε as part of its semantics. By assumption on guarded recursion, any other global type contains a topmost subterm of shape $\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \, . \, G_i$, which adds two letters to any word in its semantics.

Proposition 2.26. For any non-deterministic global type G, if $\varepsilon \in \mathcal{L}(G)$ then $G = 0$.

Example 2.27. Recall the global type G_{LiSe} (Example 2.23) where p sends a list to q. The FSM for its semantics is visualised in Fig. 2.6. The different styles of its states represent the different kind of subterms they correspond to: *binder states* with their dashed line correspond to recursion variable binders, while *recursion states* with their dash-dotted lines indicate the use of a recursion variable. We omit ε for transitions from recursion to binder states. ◀

The syntax of global types yield interesting structural properties for the FSMs of their semantics. Intuitively, every such FSM has a tree-like structure where backward transitions only happen at leaves of the tree, are always labelled with ε, and only lead to ancestors.

Definition 2.28 (Ancestor-recursive, intermediate recursion, non-merging, dense, and cone)**.** Let $A = (Q, \Delta, \delta, q_0, F)$ be a finite state machine. We say that A is *ancestor-recursive* if there is a function $\mathrm{lvl}\colon Q \to \mathbb{N}$ such that, for every transition $q \xrightarrow{x} q' \in \delta$, one of the following two properties holds:

(a) $\mathrm{lvl}(q) > \mathrm{lvl}(q')$, or

(b) $x = \varepsilon$ and there is a run from the initial state q_0 (without going through q) to q' which can be completed to reach q: $q_0 \to \ldots \to q_n$ is a run with $q_n = q'$ and $q \neq q_i$ for every $0 \leq i \leq n$, and the run can be extended to $q_0 \to \ldots \to q_n \to \ldots \to q_{n+m}$ with $q_{n+m} = q$. Then, the state q' is called *ancestor* of q.

We call the first kind (a) of transition *forward transition* while the second kind (b) is a *backward transition*. We say A is free from *intermediate recursion* if every state q with more than one outgoing transition, i.e. $|\{q' \mid q \to q' \in \delta\}| > 1$, has only forward transitions. We say that A is *non-merging* if every state only has one incoming edge with greater level, i.e. $\forall q'. \{q \mid q \to q' \in \delta \wedge \mathrm{lvl}(q) > \mathrm{lvl}(q')\} \leq 1$. The state machine A is *dense* if, for every $q \xrightarrow{x} q' \in \delta$, the transition label x is ε implies that q has only one outgoing transition. Last, the *cone* of q are all states q' which are reachable from q and have a smaller level than q, i.e. $\mathrm{lvl}(q) > \mathrm{lvl}(q')$.

The FSM in Fig. 2.6 illustrates this shape where the root of the tree is at the top.

Proposition 2.29 (Shape of $\mathsf{GAut}(G)$)**.** Let G be some global type. Then, $\mathsf{GAut}(G)$ is ancestor-recursive, free from intermediate recursion, non-merging, dense, and sink-final. For each word w, there are at most three runs ρ, ρ_1, and ρ_2 in $\mathsf{GAut}(G)$; moreover, when they exist, the runs can be chosen such that $\rho_1 := \rho \xrightarrow{\varepsilon} q$ and $\rho_2 := \rho_1 \xrightarrow{\varepsilon} q'$ for some states q and q'.

2.7 The Implementability Problem

Equipped with formal definitions of our implementation model and protocol specification mechanisms, we define the implementability problem.

Definition 2.30 (Implementability Problem)**.** A language $L \subseteq \Gamma^\omega$ is said to be *implementable* if there exists a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ such that

- *deadlock freedom*: $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ is deadlock-free, and
- *protocol fidelity*: L is the language of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$.

We say that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ implements L. If L is given as global type G (or HMSC H), we also say that G (or H) is implemented.

Example 2.31. The CSM in Figure 2.1 implements the HMSC in Figure 2.5 and the global type from Example 2.23. ◀

For HMSCs, the implementability question was studied as *safe realisability* [6]. If the CSM is not required to be deadlock-free, it is called weak realisability. Here, we focus on deadlock-free implementations. Our notion of deadlock is the natural one for CSMs: non-final configurations for which there is no further transition. In session types, the notion of deadlock often slightly differs.

Definition 2.32 (Soft deadlocks)**.** Let $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. A configuration $(\vec{q}, \xi)$ is a *soft deadlock* if there is no $(\vec{q}\,', \xi')$ with $(\vec{q}, \xi) \rightarrow (\vec{q}\,', \xi')$ and $(\vec{q}, \xi)$ is no final sink-state configuration. We say $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is free from soft deadlocks if every reachable configuration is no soft deadlock.

Intuitively, a configuration can only be considered actually final if no state machine has an outgoing transition. It is straightforward that every deadlock is a soft deadlock. Let us define the *soft implementability problem.*

Definition 2.33 (Soft Implementability Problem)**.** A language $L \subseteq \Gamma^\omega$ is said to be *softly implementable* if there exists a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that

- *soft deadlock freedom*: $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is free from soft deadlocks, and
- *protocol fidelity*: L is the language of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

We say that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ softly implements L.

Different applications call for different notions of deadlock freedom. In distributed computing, it is fine if a server keeps listening to incoming requests while, in embedded computing, however, it can be essential that all participants eventually stop. We believe this is a design choice and provide solutions to both problems. In Chapter 5, we will develop a sound and complete algorithm to solve the implementability problem for global types. We will explain that a simple postprocessing step makes the presented approach also sound and complete for the soft implementability problem (cf. Section 5.5.2). Further properties of protocol implementations have been considered in the literature. We refer to Section 5.8 for details on which of these are guaranteed by our complete MST projection operator.

Chapter 3

Generalising Projection for Multiparty Session Types

Given a global type, a partial projection operator attempts to generate local specifications for participants, called local types. If they are defined, they are safe to use and provide a solution for the implementability problem of the given global type. In this chapter, we first define and visually explain the classical projection approach for MSTs. We exemplify the features different merge operator, which is used during projection, support. In particular, we showcase the lack of support for sender-driven choice. We present a generalised (classical) projection operator that supports sender-driven choice thanks to a novel message availability analysis. Notably, proving correctness requires new proof ideas. Lastly, we explain how classical projection approaches are brittle and easily reject implementable protocols, guiding us to the next chapter where we show decidability of the MST implementability problem using MSC techniques.

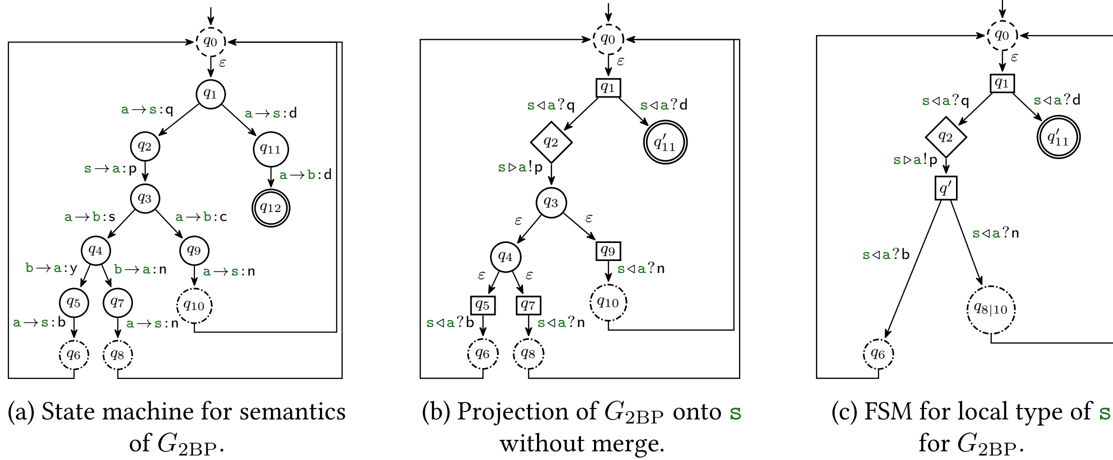
3.1 Classical Multiparty Session Type Projection

In this section, we present the first automata-based explanation of the classical projection approach for MSTs, considering different merge operators.

3.1.1 Introductory Example

Let us explain the idea behind the classical projection with an example first. Consider a variant of the well-known two buyer protocol from the MST literature, e.g. [109, Fig. 4(2)], which can be specified as global type:

$$G_{2\mathrm{BP}} \;:=\; \mu t \,.\, + \begin{cases} \mathsf{a}\to\mathsf{s}:\mathsf{query}\,.\,\mathsf{s}\to\mathsf{a}:\mathsf{price}\,.\, + \begin{cases} \mathsf{a}\to\mathsf{b}:\mathsf{split}\,.\,(\mathsf{b}\to\mathsf{a}:\mathsf{yes}\,.\,\mathsf{a}\to\mathsf{s}:\mathsf{buy}\,.\,t + \mathsf{b}\to\mathsf{a}:\mathsf{no}\,.\,\mathsf{a}\to\mathsf{s}:\mathsf{no}\,.\,t) \\ \mathsf{a}\to\mathsf{b}:\mathsf{cancel}\,.\,\mathsf{a}\to\mathsf{s}:\mathsf{no}\,.\,t \end{cases} \\ \mathsf{a}\to\mathsf{s}:\mathsf{done}\,.\,\mathsf{a}\to\mathsf{b}:\mathsf{done}\,.\,0 \end{cases} .$$



(a) State machine for semantics of $G_{2\mathrm{BP}}$.

(b) Projection of $G_{2\mathrm{BP}}$ onto s without merge.

(c) FSM for local type of s for $G_{2\mathrm{BP}}$.

Figure 3.1: Two Buyer Protocol.

Two buyers a and b purchase a sequence of items from seller s. We informally describe the protocol and *emphasise* the interactions. At the start and after every purchase (attempt), buyer a can decide whether to buy the next item or whether they are *done*. For each item, buyer a *queries* its price and the seller s replies with the *price*. Subsequently, buyer a decides whether to *cancel* the purchase process for the current item or proposes to *split* to buyer b that can *accept* or *reject*. In both cases, buyer a notifies the seller s if they want to *buy* the item or *not*.

Classical MST frameworks employ a partial *projection operator* with an in-built *merge operator* to solve the implementability problem. For each participant, the projection operator takes the global type and removes all interactions the participant is not involved in. Fig. 3.1a illustrates the semantics of $G_{2\mathrm{BP}}$ while Fig. 3.1b gives the projection onto seller s before the merge operator is applied – in both, messages are abbreviated with their first letter. It is easy to see that this procedure introduces non-determinism, e.g. in q_3 and q_4, which shall be resolved by the merge operator. Most merge operators can resolve the non-determinism in Fig. 3.1b. A merge operator checks whether it is safe to merge the states and it might fail so it is a partial operation. For instance, every kind of state, indicated by a state's style in Fig. 3.1b, can only be merged with states of the same kind or states of circular shape. For a participant, the result of the projection, if defined, is a local type. They act as local specifications and their syntax is similar to the one of global types.

3.1.2 Local Types

Here, local types can be considered as syntactic intermediate representation for local specifications. Intuitively, every collection of local types constitutes a CSM through their semantics, justifying our treatment.

Definition 3.1 (Syntax of local types)**.** For a participant p, the *local types* are defined as follows:

$$L ::= 0 \mid \bigoplus_{i\in I} \mathsf{q}_i!\mathsf{m}_i \,.\, L_i \mid \&_{i\in I} \mathsf{q}_i?\mathsf{m}_i \,.\, L_i \mid \mu t \,.\, L \mid t$$

We call $\oplus_{i\in I} \mathsf{q}_i!\mathsf{m}_i$ an internal choice and $\&_{i\in I} \mathsf{q}_i?\mathsf{m}_i$ an external choice. For both, we require the choice to be unique, i.e. $\forall i, j \in I . i \neq j \Rightarrow \mathsf{q}_i \neq \mathsf{q}_j \vee \mathsf{m}_i \neq \mathsf{m}_j$. Similarly to global types, we assume $|I| > 0$, omit $\oplus$ or $\&$ if there is no actual choice, and require recursion to be guarded as well as recursion variables to be bound and distinct.

For the semantics of local types, we analogously construct a state machine $\mathsf{LAut}(\text{-})$. In contrast to global types, we omit the closure $\mathcal{C}^{\sim}(\text{-})$ because events of participants are not reordered by $\sim$.

Example 3.2 (Local type)**.** Classical projection yields the following local type when $G_{2\text{BP}}$ is projected onto s:

$$G_{2\text{BP}} \;:=\; \mu t \,.\, \& \begin{cases} \mathsf{s}\triangleleft\mathsf{a}?\mathsf{query} \,.\, \mathsf{s}\triangleright\mathsf{a}!\mathsf{price} \,.\, \& \begin{cases} \mathsf{s}\triangleleft\mathsf{a}?\mathsf{buy} \,.\, t \\ \mathsf{s}\triangleleft\mathsf{a}?\mathsf{no} \,.\, t \end{cases} \\ \mathsf{s}\triangleleft\mathsf{a}?\mathsf{done} \,. \end{cases} .$$

◀

Definition 3.3 (Semantics for local types)**.** Given a local type L for participant p, we index syntactic subterms as for the semantics of global types. We construct a state machine $\mathsf{LAut}(L) = (Q, \Gamma_{\mathsf{p}}, \delta, q_0, F)$ where

- Q is the set of all indexed syntactic subterms in L,
- δ is the smallest set containing $([\oplus_{i\in I} \mathsf{q}_i!\mathsf{m}_i \,.\, [L_i, k_i], k], \mathsf{p}\triangleright\mathsf{q}_i!\mathsf{m}_i, [L_i, k_i])$ and $([\&_{i\in I} \mathsf{q}_i?\mathsf{m}_i \,.\, [L_i, k_i], k], \mathsf{p}\triangleleft\mathsf{q}_i?\mathsf{m}_i, [L_i, k_i])$ for each $i \in I$, as well as $([\mu t \,.\, [L', k'_2], k'_1], \varepsilon, [L', k'_2])$ and $([t, k'_3], \varepsilon, [\mu t \,.\, [L', k'_2], k'_1])$,
- $q_0 = [L, 1]$ and $F = \{[0, k] \mid k \text{ is an index for subterm } 0\}$.

We define the semantics of L as language of this automaton: $\mathcal{L}(L) = \mathcal{L}(\mathsf{LAut}(L))$.

Example 3.4. The FSM for the semantics of the local type in Example 3.2 is visualised in Fig. 3.1c. Compared to global types, we distinguish two more kinds of states for local types: a *send state* (internal choice) has a diamond shape while a *receive state* (external choice) has a rectangular shape. For states with ε as next action, we keep the circular shape and call them *neutral states*. ◀

It is easy to see that FSMs for local types have similar structural properties like the ones for global types.

Proposition 3.5 (Shape of $\mathsf{LAut}(L)$)**.** Let L be some local type. Then, $\mathsf{LAut}(L)$ is ancestor-recursive, free from intermediate recursion, non-merging, dense, and sink-final. In addition, there are no mixed-choice states, i.e. there is no state with both outgoing send and receive transitions.

For both global and local types, the only forward ε-transitions occur precisely from binder states, associated with μt, while backward transitions happen from recursion states, associated with t, to binder states. The illustrations for our examples always have the initial state, which is the state with the greatest level, at the top. This is why we use greater and higher as well as smaller and lower interchangeably for levels.

In Section 8.2.2, we present a workflow to transform FSMs in a way that they satisfy the restrictions above. This shows that local types and FSMs over $\Gamma_{\mathtt{p}}$ are equi-expressive if every final state has no outgoing transitions and they do not have mixed-choice states, i.e. there is no state with both send and receive transition.

3.1.3 Classical Projection Operator with Parametric Merge

The classical projection operator checks on-the-fly if the resulting local types would be safe to use. Behind the scenes, these checks are conducted by a partial merge operator. We consider different variants of merge operators from the literature and exemplify the features they support. We provide visual explanations of the classical projection operator with these merge operators on the state machines of global types.

Definition 3.6 (Projection operator)**.** For a merge operator $\sqcap$, the *projection* of a global type G onto a participant $\mathtt{r} \in \mathcal{P}$ is a local type that is defined as follows:[1]

- $0 {\restriction}^{\sqcap}_{\mathtt{r}} := 0$
- $t {\restriction}^{\sqcap}_{\mathtt{r}} := t$
- $(\mu t \,.\, G) {\restriction}^{\sqcap}_{\mathtt{r}} := \begin{cases} \mu t \,.\, (G {\restriction}^{\sqcap}_{\mathtt{r}}) & \text{if } G {\restriction}^{\sqcap}_{\mathtt{r}} \neq t \\ 0 & \text{otherwise} \end{cases}$
- $\left(\sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : \mathtt{m}_i \,.\, G_i\right) {\restriction}^{\sqcap}_{\mathtt{r}} := \begin{cases} \oplus_{i \in I}\ \mathtt{q}_i ! \mathtt{m}_i \,.\, (G_i {\restriction}^{\sqcap}_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{p} \\ \&_{i \in I}\ \mathtt{p} ? \mathtt{m}_i \,.\, (G_i {\restriction}^{\sqcap}_{\mathtt{r}}) & \text{if } \mathtt{r} = \mathtt{q} \\ \sqcap_{i \in I}\ G_i {\restriction}^{\sqcap}_{\mathtt{r}} & \text{otherwise} \end{cases}$

Both rules for 0 and t act as base case and simply yield their input. The rule for recursion variable binders simply keeps the binder and recurses. For choice, the situation is slightly more involved. One checks if the participant to project onto is either the sender or receiver and keeps its part in this case. If not, the participant is not involved in the choice and the subsequent branches ought to be *merged*. Merging ensures that a participant either does not have different for these different branches or learns about the branch prior to committing to one of the branches by a distinct action. Several *merge operators* have been proposed in the literature.

Definition 3.7 (Merge operators)**.** Let L_1 and L_2 be local types for a participant $\mathtt{r}$, and $\sqcap$ be a merge operator. We define different cases for the result of $L_1 \sqcap L_2$:

[1] The case split for the recursion binder changes slightly across different definitions. We choose a simple but also the least restrictive condition. We simply check whether the recursion is vacuous, i.e. $\mu t \,.\, t$, and omit it in this case. We also require to omit μt if t is never used in the result.

(1) L_1 if $L_1 = L_2$

(2) $\begin{pmatrix} \&_{i \in I \setminus J}\ \mathtt{q}?m_i \,.\, L'_{1,i} & \& \\ \&_{i \in I \cap J}\ \mathtt{q}?m_i \,.\, (L'_{1,i} \sqcap L'_{2,i}) & \& \\ \&_{i \in J \setminus I}\ \mathtt{q}?m_i \,.\, L'_{2,i} & \end{pmatrix}$ if $\begin{cases} L_1 = \&_{i \in I}\ \mathtt{q}?\mathsf{m}_i \,.\, L'_{1,i}, \\ L_2 = \&_{i \in J}\ \mathtt{q}?\mathsf{m}_i \,.\, L'_{2,i} \end{cases}$

(3) $\mu t_1 \,.\, (L'_1 \sqcap L'_2[t_2/t_1])$ if $L_1 = \mu t_1 \,.\, L'_1$ and $L_2 = \mu t_2 \,.\, L'_2$

Each merge operator is defined by a collection of cases it can apply. If none of the respective cases applies, the result of the merge is undefined. The plain merge $\sqcap_p$ [67] can only apply Case (1). The semi-full merge $\sqcap_s$ [130] can apply Cases (1) and (2). The full merge $\sqcap_f$ [109] can apply all Cases (1), (2), and (3).

Note that Case (2) does not allow to receive from different senders – a shortcoming we address with our generalised projection operator later.

Remark 3.8 (Correctness of projection)**.** This would be the correctness criterion for projection: let G be some global type and let plain merge $\sqcap_p$, semi full merge $\sqcap_s$, full merge $\sqcap_f$, or availability merge $\sqcap_a$ be the merge operator $\sqcap$. If $G{\upharpoonright}^{\sqcap}_{\mathtt{p}}$ is defined for each participant $\mathtt{p}$, then the CSM $\{\!\{\mathsf{LAut}(G{\upharpoonright}^{\sqcap}_{\mathtt{p}})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ implements G.
We do not actually prove this so we do not state it as lemma. *But why does this hold?* The implementability condition is the combination of deadlock freedom and protocol fidelity. Honda et al. [67] show *progress*, which entails deadlock freedom and *session fidelity*, proving that the global type is followed, which amounts to protocol fidelity. *Subject reduction* intuitively requires that the considered types can take a step when the system takes a step. With [130, Thm. 1], protocol fidelity holds for the semi-full merge. Scalas and Yoshida pointed out that several versions of classical projection with the full merge are flawed [109, Sec. 8.1]. They list two MST frameworks with full merge operator that were not shown to be flawed. However, the merge operator by Toninho and Yoshida [122] does not allow to apply Case (3). Scalas et al. [108] actually use the projection operator by Deniélou et al. [47] for global types. Still, we have chosen a full merge operator whose correctness follows from the correctness of our more general availability merge operator that we present later in this chapter.

3.1.4 Visual Explanation of the Parametric Projection Operator

Example 3.9 (Projection without merge / Collapsing erasure)**.** In Section 3.1.1, we considered the two buyer protocol $G_{2\mathrm{BP}}$ and the FSM for its semantics in Fig. 3.1a. We projected (without merge) onto seller $\mathtt{s}$ to obtain the FSM in Fig. 3.1b. In general, we also collapse neutral states with a single ε-transition and their only successor. We call this *collapsing erasure*. We only need to actually collapse states for the protocol in Fig. 3.4a. In all other illustrations, we indicate the interactions the participant is not involved with the following notation: $[\mathtt{p} \to \mathtt{q} : \mathsf{m}] \rightsquigarrow \varepsilon$. ◀

Let us describe *collapsing erasure* more formally. Let G be some global type and $\mathtt{r}$ be the participant onto which we project. We apply the parametric projection operator to the state machine $\mathsf{GAut}(G)$. It projects each transition label onto the respective event for participant $\mathtt{r}$: every forward transition label $\mathtt{p} \to \mathtt{q} : \mathsf{m}$ turns to $\mathtt{p} \triangleright \mathtt{q} ! \mathsf{m}$ if $\mathtt{r} = \mathtt{p}$, the event $\mathtt{q} \triangleleft \mathtt{p} ? \mathsf{m}$ if $\mathtt{r} = \mathtt{q}$, and ε otherwise. Then, it collapses neutral states with a single successor: $q_{1|2}$ replaces two states q_1 and q_2 if $q_1 \xrightarrow{\varepsilon} q_2$ is the only forward transition for q_1. In case there is only a backward transition from q_1 to q_2, the state $q_{1|2}$ is also final. This accounts for loops a participant is not part of. We call this procedure collapsing erasure as it erases interactions that do not belong to a participant and collapses some states. It is common to all the presented merge operators. This procedure yields a state machine over $\Gamma_{\mathtt{r}}$. It is straightforward that it is still ancestor-recursive, intermediate-recursion-free and non-merging. However, it might not be dense. In fact, it is not dense if $\mathtt{r}$ is not involved in some choice with more than one branch. This is precisely where the merge operator is needed.

3.1.5 Visual Explanation of Merge Operators

Here, we first give general explanations how the merge operators are applied after collapsing erasure. Subsequently, we walk through a sequence of examples showcasing the different features of merge operators.

Parametric Merge in the Visual Explanation. The parametric projection operator applies the merge operator on states with non-determinism. Visually, these correspond precisely to the remaining neutral states (since all neutral states with a single successor have been collapsed) – for instance, a neutral state q_1 with $q_1 \xrightarrow{\varepsilon} q_2$ and $q_1 \xrightarrow{\varepsilon} q_3$ for $q_2 \neq q_3$. As for the syntactic version, we do only explain the 2-ary case but the reasoning easily lifts to the n-ary case. No information is propagated when the merge operator recurses and recursion variables are never unfolded. Thus, we can ignore backward transitions and consider the cones of q_2 and q_3, which are defined as all states with lower levels (and transitions) that are reachable from q_2 respectively q_3. Intuitively, we iteratively apply the merge operator from lower to higher levels. However, we might need to descend again when the merge operator is applied recursively.

Plain Merge. The plain merge is not applied recursively. Thus, we consider q_1 with $q_1 \xrightarrow{\varepsilon} q_2$ and $q_1 \xrightarrow{\varepsilon} q_3$ for $q_2 \neq q_3$ such that q_1 has the lowest level for which this holds. Hence, we can assume that each cone of q_2 and q_3 does not contain neutral states. Then, the plain merge is only defined if there is an isomorphism between the states of both cones that satisfy the following conditions:

- it preserves the transition labels and hence the kind of states, and
- if a state has a backward transition to a state outside of the cone, its isomorphic state has a transition to the same state.

If defined, the result is q_2 with its cone (and q_3 with its cone is removed).

Semi-full Merge. The semi-full merge applies itself recursively. Thus, we consider two states $q_2 \neq q_3$ that shall be merged. As before, we can assume that each cone of q_2 and q_3 does not contain neutral states (because they would have been taken care of prior in the recursion). In addition to plain merge, the semi-full merge allows to merge receive states. For these, we introduce a new receive state $q_{2|3}$ from which all new transitions start. For all possible transitions from q_2 and q_3, we check if there is a transition with the same label from the other state. For the ones not in common, we simply add the respective transition with the state it leads to and its respective cone. For the ones in common, we recursively check if the two states, which both transitions lead to, can be merged. If not, the semi-full merge is undefined. If so, we add the original transition to the state of the respective merge and keep its cone.

Full Merge. The full merge basically applies the idea of recursively applying the merge operator to another case: it allows to descend for recursion variable binders.

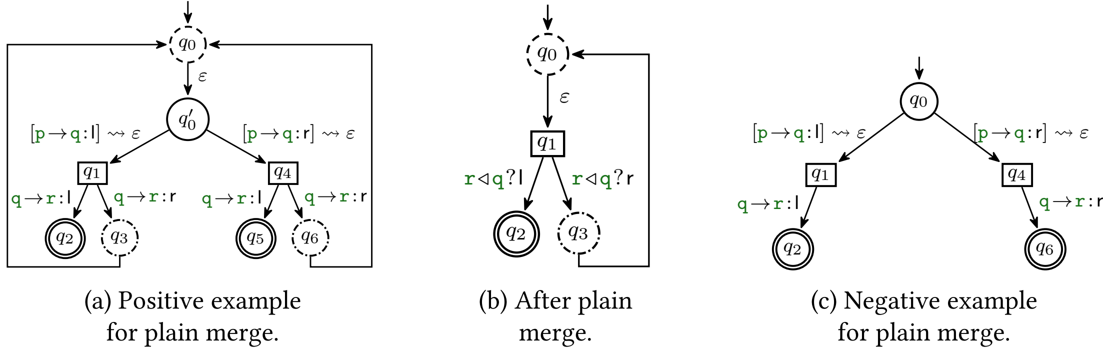
3.1.6 Features of Different Merge Operators by Example

In this section, we exemplify which features each of the merge operators supports. We present a sequence of implementable global types. Nevertheless, some cannot be handled by some merge operators. If a global type is not projectable using some merge operator, we say it is *rejected* and it constitutes a *negative* example for this merge operator. We focus on participant $\mathtt{r}$ when projecting. Thus, rejected mostly means that there is (at least) no projection onto $\mathtt{r}$. If a global type is projectable by some merge operator, we call it a *positive* example. All examples strive for minimality and follow the idea that participants decide whether to take a left (l) or right (r) branch of a choice.

Example 3.10 (Positive example for plain merge)**.** Consider this global type:

$$\mu t\,.\, + \begin{cases} \mathtt{p}\to\mathtt{q}:\mathsf{l}\,.\,(\mathtt{q}\to\mathtt{r}:\mathsf{l}\,.\,0 + \mathtt{q}\to\mathtt{r}:\mathsf{r}\,.\,t) \\ \mathtt{p}\to\mathtt{q}:\mathsf{r}\,.\,(\mathtt{q}\to\mathtt{r}:\mathsf{l}\,.\,0 + \mathtt{q}\to\mathtt{r}:\mathsf{r}\,.\,t) \end{cases} .$$

It is implementable and the state machine for its semantics is given in Fig. 3.2a. After collapsing erasure, there is a non-deterministic choice from q'_0 leading to q_1 and q_4 since $\mathtt{r}$ is not involved in the initial choice. The plain merge operator can resolve this non-determinism since both cones of q_1 and q_4 represent the same subterm, yielding the existence of an isomorphism satisfying the above conditions. The result is illustrated in Fig. 3.2b. It is also the FSM of a local type for $\mathtt{r}$ which is the result of the (syntactic) plain merge: $\mu t\,.\,(\mathtt{q}?\mathsf{l}\,.\,0 \,\&\, \mathtt{q}?\mathsf{r}\,.\,t)$. ◀



(a) Positive example for plain merge.

(b) After plain merge.

(c) Negative example for plain merge.

Figure 3.2: The FSM on the left represents an implementable global type that is accepted by plain merge. It implicitly shows the FSM after collapsing erasure: every interaction $\mathtt{r}$ is not involved in is given as $[\mathtt{p}\!\to\!\mathtt{q}\!:\!\mathsf{l}] \rightsquigarrow \varepsilon$. The FSM in the middle is the result of the plain merge. The FSM on the right represents an implementable global type that is rejected by plain merge. It is obtained from the left one by removing one choice option in each branch of the initial choice.

Our explanation on FSMs allows to check congruence of cones to merge while the definition requires syntactic equality. If we swap the order of branches $\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{l}$ and $\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{r}$ in Fig. 3.2a on the right, the syntactic merge rejects. Both are semantically the same protocol specification so we expect tools to check for such easy fixes.

Example 3.11 (Negative example for plain merge)**.** We consider the following simple implementable global type where the choice by $\mathtt{p}$ is propagated by $\mathtt{q}$ to $\mathtt{r}$:

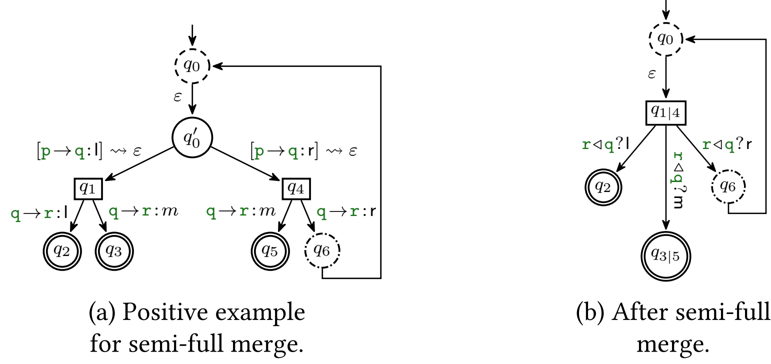
$$+\begin{cases} \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathsf{l}\,.\,\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{l}\,.\,0 \\ \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathsf{r}\,.\,\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{r}\,.\,0 \end{cases} .$$

The corresponding state machine is illustrated in Fig. 3.2c. Here, q_0 exhibits non-determinism but the plain merge fails because q_1 and q_4 have different outgoing transition labels. ◀

Intuitively, the plain merge operator forbids that any, but the two participants involved in a choice, can have different behaviour after the choice. It basically forbids propagating a choice. The semi-full merge overcomes this shortcoming and can handle the previous example. We present a slightly more complex one to showcase its features.

Example 3.12 (Positive example for semi-full merge)**.** Let us consider this implementable global type:

$$\mu t\,.\,+\begin{cases} \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathsf{l}\,.\,(\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{l}\,.\,0 + \mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{m}\,.\,0) \\ \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathsf{r}\,.\,(\mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{m}\,.\,0 + \mathtt{q}\!\to\!\mathtt{r}\!:\!\mathsf{r}\,.\,t) \end{cases} .$$



(a) Positive example for semi-full merge.

(b) After semi-full merge.

Figure 3.3: The FSM on the left represents an implementable global type that is accepted by semi-full merge. The FSM on the right is the result of the semi-full merge.

The state machine for its semantics is illustrated in Fig. 3.3a. After applying collapsing erasure, there is a non-deterministic choice from q_0 leading to q_1 and q_4 since $\mathtt{r}$ is not involved in the initial choice, We apply the semi-full merge for both states. Both are receive states so Case (2) applies. First, we observe that $\mathtt{r}\triangleleft\mathtt{q}?\mathsf{l}$ and $\mathtt{r}\triangleleft\mathtt{q}?\mathsf{r}$ are unique to one of the two states so both transitions, with the cones of the states they lead to, can be kept. Second, there is $\mathtt{r}\triangleleft\mathtt{q}?\mathsf{m}$ which is common to both states. We recursively apply the semi-full merge and, with Case (1), observe that the result $q_{3|5}$ is simply a final state. Overall, we obtain the state machine in Fig. 3.3b, which is equivalent to the result of the syntactic projection with semi-full merge: $\mu t\,.\,(\mathtt{q}?\mathsf{l}\,.\,0 \;\&\; \mathtt{q}?\mathsf{m}\,.\,0 \;\&\; \mathtt{q}?\mathsf{r}\,.\,t)$. ◀

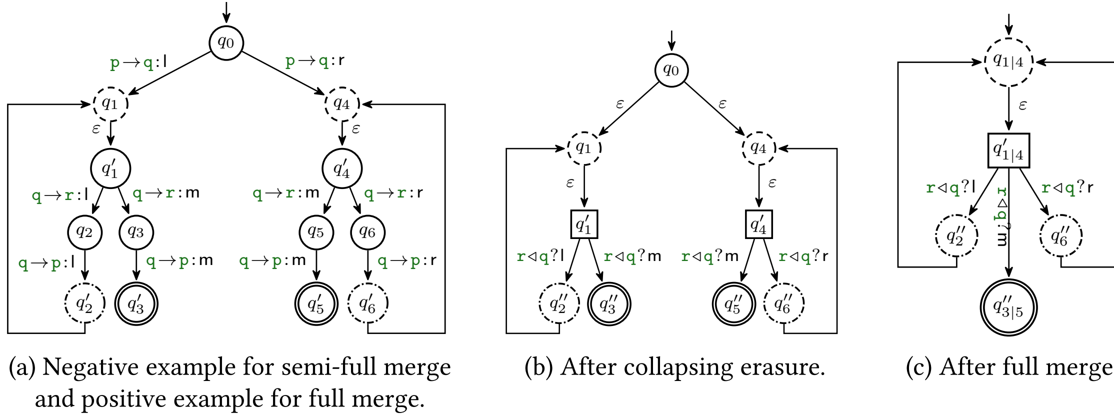
Example 3.13 (Negative example for semi-full merge and positive example for full merge)**.** The semi-full merge operator rejects the following implementable global type:

$$+\begin{cases}\mathtt{p}\to\mathtt{q}:\mathsf{l}\,.\,\mu t_1\,.\,(\mathtt{q}\to\mathtt{r}:\mathsf{l}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{l}\,.\,t_1 + \mathtt{q}\to\mathtt{r}:\mathsf{m}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{m}\,.\,0)\\ \mathtt{p}\to\mathtt{q}:\mathsf{r}\,.\,\mu t_2\,.\,(\mathtt{q}\to\mathtt{r}:\mathsf{m}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{m}\,.\,0 + \mathtt{q}\to\mathtt{r}:\mathsf{r}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{r}\,.\,t_2)\end{cases}.$$

Its FSM and the FSM after collapsing erasure is given in Figs. 3.4a and 3.4b. Intuitively, it would need to recursively merge the parts after both recursion binders in order to merge the branches with receive event $\mathtt{r}\triangleleft\mathtt{q}?\mathsf{m}$ but it cannot do so. The full merge can handle this global type. It can descend beyond q_1 and q_4 and is able to merge q_1' and q_4'. To obtain $q''_{3|5}$, it applies Case (1) while $q'_{1|4}$ is only feasible with Case (2). The result is embedded into the recursive structure to obtain the FSM in Fig. 3.4c. It is equivalent to the (syntactic) result, which renames the recursion variable for one branch: $\mu t_1\,.\,(\mathtt{q}?\mathsf{l}\,.\,t_1 \;\&\; \mathtt{q}?\mathsf{m}\,.\,0 \;\&\; \mathtt{q}?\mathsf{r}\,.\,t_1)$. ◀

Example 3.14 (Negative example for full merge)**.** We consider a simple implementable global type where $\mathtt{p}$ propagates its decision to $\mathtt{r}$ in the top branch while $\mathtt{q}$ propagates it in the bottom branch:

$$+\begin{cases}\mathtt{p}\to\mathtt{q}:\mathsf{l}\,.\,\mathtt{p}\to\mathtt{r}:\mathsf{l}\,.\,0\\ \mathtt{p}\to\mathtt{q}:\mathsf{r}\,.\,\mathtt{q}\to\mathtt{r}:\mathsf{r}\,.\,0\end{cases}.$$



(a) Negative example for semi-full merge and positive example for full merge.
(b) After collapsing erasure.
(c) After full merge.

Figure 3.4: The FSM on the left represents an implementable global type that is rejected by the semi-full merge and accepted by the full merge.

It is illustrated in Fig. 3.5. This cannot be projected onto r by the full merge operator for which all receive events need to have the same sender. This global type does not require sender-driven choice but, in fact, sender-driven choice is closely related to this phenomenon: if a sender can send to different receivers upon branching, we assume they can also receive from different senders. ◀

In this section, we presented the first automata-based explanation of the classical projection and merge operators for MSTs. The presented sequence of merge operators was intentionally incremental – with regard to the subset of cases from Definition 3.7 they can apply. Of course, one could consider the different cases in all combinations. However, it does not give any additional insights regarding the concept of the classical projection operator and its possible merge operators. After we show how to generalise the classical projection approach to sender-driven choice, we showcase why it does not lend itself to a complete approach.

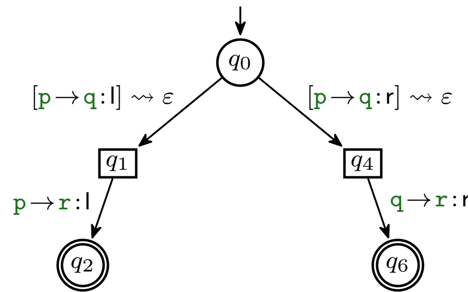


Figure 3.5: Negative example for full merge.

3.2 Generalising Classical Projection for Sender-driven Choice

In the previous section, we showed that the full merge operator cannot handle protocols where participants receive from different senders. In this section, we show how to extend the classical projection approach for global types with sender-driven choice. This allows senders to send to different receivers but, analogously, it also allows receivers to receive from different senders. In the next section, we prove our generalised projection correct.

3.2.1 Motivating Example: Load Balancing

To motivate our generalised projection operator, let us consider a load balancing protocol, a common pattern in distributed computing.

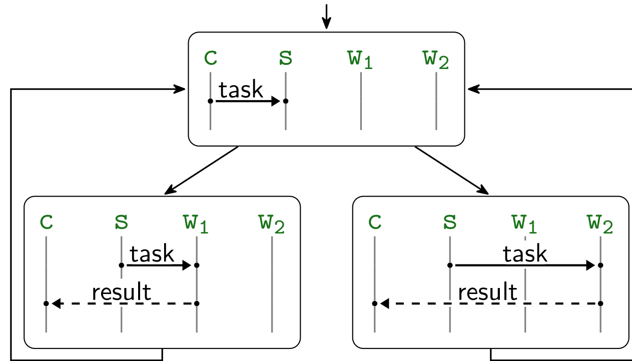
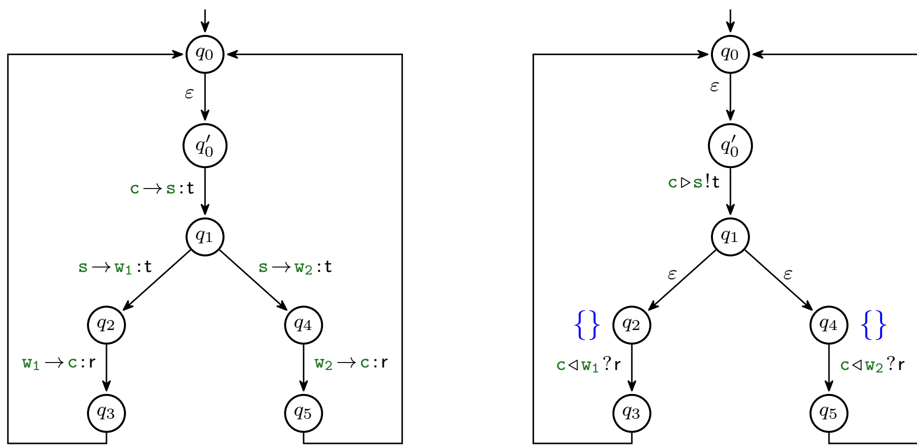


Figure 3.6: Load Balancing Protocol, as HMSC.



(a) As FSM.

(b) After collapsing erasure onto c with available messages.

Figure 3.7: Load Balancing Protocol, as FSM for G_{LoBa}.

A simple load balancing scenario can be modelled with the global type

$$G_{\text{LoBa}} := \mu t\,.\,\mathtt{c} \to \mathtt{s} : \mathsf{task}\,.\; + \begin{cases} \mathtt{s} \to \mathtt{w}_1 : \mathsf{task}\,.\,\mathtt{w}_1 \to \mathtt{c} : \mathsf{result}\,.\,t \\ \mathtt{s} \to \mathtt{w}_2 : \mathsf{task}\,.\,\mathtt{w}_2 \to \mathtt{c} : \mathsf{result}\,.\,t \end{cases} .$$

Figures 3.6 and 3.7a show the corresponding HMSC and the state machine for the global type. The client sends a request to a server in a loop. The server chooses one of two workers to forward the request to. The chosen worker handles the request and replies to the client. In this protocol, the server communicates with a different worker in each branch.

We can give the following local types for G_{LoBa}:

$$\begin{aligned} \mathtt{s} :\ & \mu t\,.\,\mathtt{c}?\mathsf{task}\,.\,(\mathtt{w}_1!\mathsf{task}\,.\,t \;\oplus\; \mathtt{w}_2!\mathsf{task}\,.\,t) \\ \mathtt{c} :\ & \mu t\,.\,\mathtt{s}!\mathsf{task}\,.\,(\mathtt{w}_1?\mathsf{result}\,.\,t \;\&\; \mathtt{w}_2?\mathsf{result}\,.\,t) \\ \mathtt{w}_\mathtt{i} :\ & \mu t.\,\mathtt{s}?\mathsf{task}\,.\,\mathtt{c}!\mathsf{result}\,.\,t \text{ for } i \in \{1,2\} \end{aligned}$$

However, none of the previous projection operators can yield these local types. They all cannot handle global types that allow to send to different receivers or to receive from different senders upon branching. Our MST framework overcomes this shortcoming. This generalisation is non-trivial. When limiting the reception to messages from a single participant, one can rely on the FIFO order provided by the corresponding channel. However, messages coming from different sources are only partially ordered. Thus, unlike previous approaches, our merge operator looks at the result of a *causality analysis* on the global type to make sure that this partial ordering cannot introduce any confusion.

We demonstrate that a straightforward generalisation of existing projection operators can lead to unsoundness. Consider a *naive* merge operator that merges branches with internal choice if they are equal, and for receives, simply always merges external choices – also from different senders. In addition, it removes empty loops. For G_{LoBa}, this naive projection yields the expected local types presented before.

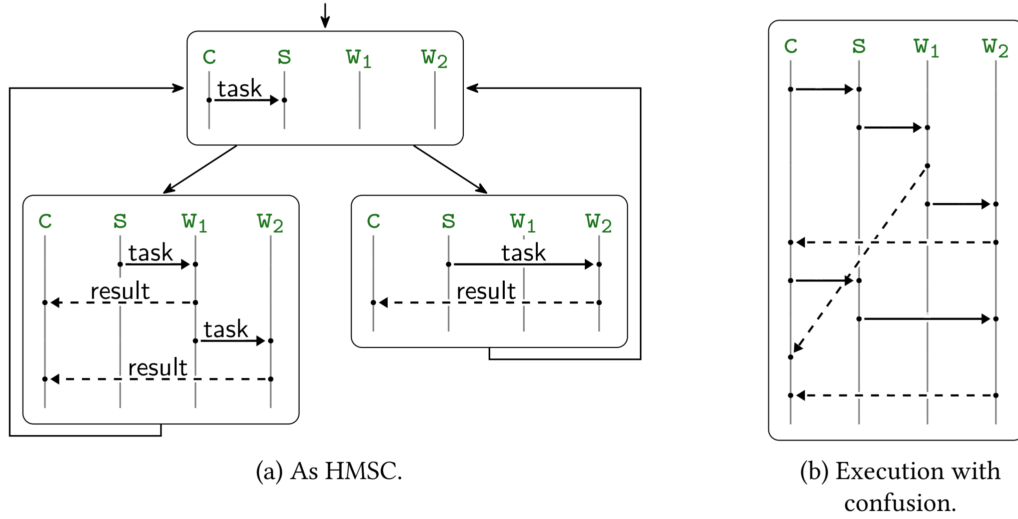
We show that naive projection can be unsound. Figures 3.8a and 3.9a illustrate a variant of load balancing, which can also be specified as global type:

$$G'_{\text{LoBa}} := \mu t\,.\,\mathtt{c} \to \mathtt{s} : \mathsf{task}\,.\; + \begin{cases} \mathtt{s} \to \mathtt{w}_1 : \mathsf{task}\,.\,\mathtt{w}_1 \to \mathtt{c} : \mathsf{result}\,.\,\mathtt{w}_1 \to \mathtt{w}_2 : \mathsf{task}\,.\,\mathtt{w}_2 \to \mathtt{c} : \mathsf{result}\,.\,t \\ \mathtt{s} \to \mathtt{w}_2 : \mathsf{task}\,.\,\mathtt{w}_2 \to \mathtt{c} : \mathsf{result}\,.\,t \end{cases} .$$

Naive projection yields the following local types:

$$\begin{aligned} \mathtt{s} :\ & \mu t\,.\,\mathtt{c}?\mathsf{task}\,.\,(\mathtt{w}_1!\mathsf{task}\,.\,t \;\oplus\; \mathtt{w}_2!\mathsf{task}\,.\,t) \\ \mathtt{c} :\ & \mu t\,.\,\mathtt{s}!\mathsf{task}\,.\,(\mathtt{w}_1?\mathsf{result}\,.\,\mathtt{w}_2?\mathsf{result}\,.\,t \;\&\; \mathtt{w}_2?\mathsf{result}\,.\,t) \\ \mathtt{w}_1 :\ & \mu t\,.\,\mathtt{s}?\mathsf{task}\,.\,\mathtt{c}!\mathsf{result}\,.\,\mathtt{w}_2!\mathsf{task}\,.\,t \\ \mathtt{w}_2 :\ & \mu t.\,(\mathtt{w}_1?\mathsf{task}\,.\,\mathtt{c}!\mathsf{result}\,.\,t \;\&\; \mathtt{s}?\mathsf{task}\,.\,\mathtt{c}!\mathsf{result}\,.\,t) \end{aligned}$$

Unfortunately, the global type is not implementable. The problem is that, for the client $\mathtt{c}$, the two messages in its left branch are not causally related.



(a) As HMSC.

(b) Execution with confusion.

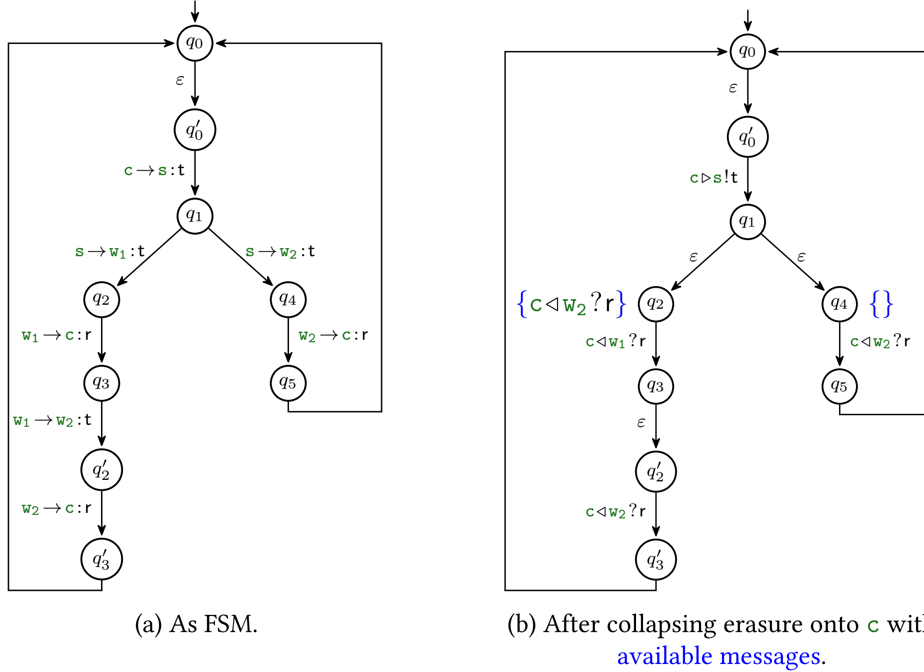
Figure 3.8: Variant of load balancing with confusion (G'_{LoBa}): as HMSC and an execution with confusion as MSC.

Consider the run depicted as MSC in Fig. 3.8b, which is not specified in the global type. The server s decided to first take the left (L) and then the right (R) branch. For s, the order LR is obvious from its events and the same applies for $\mathtt{w}_2$. For worker $\mathtt{w}_1$, every possible order R^*LR^* is plausible as it does not have events in the right branch. Since LR belongs to the set of plausible orders, there is no confusion. The messages from the two workers $\mathtt{w}_1$ and $\mathtt{w}_2$ to the client c are independent and, therefore, can be received in any order. If the client c receives $\mathtt{w}_2$?result first, then its local view is consistent with the choice RL as the order of branches. This can lead to confusion and, thus, runs that are not specified in the global type.

3.2.2 Available Messages

To identify such potential confusion when projecting, we keep track of causality between messages. We determine what messages a participant *could* receive at a given point in the global type (other than the one intended for this branch), which we call *available messages*. In Figures 3.7b and 3.9b, we depict the set of available messages where states ought to be merged. Tracking causality needs to be done at the level of the global type: we look for chains of dependent messages and also need to unfold loops. Fortunately, since we only check for the presence or absence of some messages at the head of each channel, it is sufficient to unfold each recursion at most once.

Technically, we annotate local types with the messages that could be received at that point in the protocol. We call these *availability annotated local types* and write them as $AL = \langle L, Msg \rangle$ where L is a local type and Msg is a set of messages – technically receive events, giving information about the sender, receiver and message. This signifies that when a participant has reached AL, the messages in Msg can be



(a) As FSM.

(b) After collapsing erasure onto c with available messages.

Figure 3.9: Variant of load balancing with confusion (G'_{LoBa}): as FSM and after collapsing erasure.

present in the communication channels. We annotate types using the grammar for local types (Definition 3.1), where each subterm is annotated. To recover a local type, we erase the annotation, i.e. recursively replace each $AL = \langle L, Msg \rangle$ by L. The projection internally uses annotated types.

To compute the available messages, we unfold the recursion in global types. We could get the unfolding through a congruence relation. However, this requires dealing with infinite structures, making some definitions not effective. Instead, we precompute the map from each recursion variable t to its unfolding. For a given global type G, let $get\mu$ be a function that returns a map from t to G' for each subterm $\mu t \, . \, G'$. Recall, each t in a type is different. $get\mu$ is defined as follows:

$$get\mu(0) := [\,] \qquad get\mu(t) := [\,] \qquad get\mu(\mu t \, . \, G') := [t \mapsto G'] \cup get\mu(G')$$
$$get\mu(\textstyle\sum_{i \in I} \mathtt{p} \to \mathtt{q}_i : \mathsf{m}_i \, . \, G_i) := \bigcup_{i \in I} get\mu(G_i)$$

We write $get\mu_G$ as shorthand for the map returned by $get\mu(G)$.

The projection of G onto $\mathtt{r}$, written $G{\upharpoonright}_{\mathtt{r}}$, traverses G to erase the operations that do not involve $\mathtt{r}$. During this phase, we also compute the messages that $\mathtt{r}$ may receive. The function $\mathrm{avail}(\mathcal{B}, T, G)$ computes the set of messages that other participants can send while $\mathtt{r}$ has not yet learned the outcome of the choice. This set depends on $\mathcal{B}$, the set of *blocked* participants, i.e. the participants which are waiting to receive a message and hence cannot move; T, the set of recursion variables we have already visited; and G, the subterm in G at which we compute the available messages.

Definition 3.15. Let G be the global type under consideration. We define the function avail(-, -, -) recursively:

$$\mathrm{avail}(\mathcal{B}, T, 0) := \emptyset$$
$$\mathrm{avail}(\mathcal{B}, T, \mu t\,.\,G') := \mathrm{avail}(\mathcal{B}, T \cup \{t\}, G')$$
$$\mathrm{avail}(\mathcal{B}, T, t) := \begin{cases} \emptyset & \text{if } t \in T \\ \mathrm{avail}(\mathcal{B}, T \cup \{t\}, get\mu_G(t)) & \text{if } t \notin T \end{cases}$$
$$\mathrm{avail}(\mathcal{B}, T, \textstyle\sum_{i\in I} \mathtt{p} \to \mathtt{q}_i : \mathsf{m}_i\,.\,G_i) := \begin{cases} \bigcup_{i\in I, m \in \mathcal{V}} (\mathrm{avail}(\mathcal{B}, T, G_i) \setminus \{\mathtt{q}_i \triangleleft \mathtt{p}?\mathsf{m}\}) \cup \{\mathtt{q}_i \triangleleft \mathtt{p}?\mathsf{m}_i\} & \text{if } \mathtt{p} \notin \mathcal{B} \\ \bigcup_{i\in I} \mathrm{avail}(\mathcal{B} \cup \{\mathtt{q}_i\}, T, G_i) & \text{if } \mathtt{p} \in \mathcal{B} \end{cases}$$

All channels are FIFO-ordered so we only keep the first possible message in each channel. The fourth case discards messages not at the head of the channel.

3.2.3 Availability Merge Operator

Notations and Assumptions. To simplify the notation, we assume that the index i of a choice uniquely determines the sender, receiver and message. Using this notation, we write $I \cap J$ to select the set of choices with identical sender and message value and $I \setminus J$ to select the alternatives present in I but not in J.

Definition 3.16 (Availability merge operator $\sqcap$). Let $\langle L_1, Msg_1\rangle$ and $\langle L_2, Msg_2\rangle$ be availability annotated local types for a participant $\mathtt{r}$. $\langle L_1, Msg_1\rangle \sqcap \langle L_2, Msg_2\rangle$ is defined by cases, as follows:

- $\langle L_1, Msg_1 \cup Msg_2\rangle$ if $L_1 = L_2$
- $\langle \mu t_1\,.\,(AL_1 \sqcap AL_2[t_2/t_1]), Msg_1 \cup Msg_2\rangle$ if $L_1 = \mu t_1\,.\,AL_1$, $L_2 = \mu t_2\,.\,AL_2$
- $\langle \oplus_{i\in I}\, \mathtt{q}_i!m_i\,.\,(AL_{1,i} \sqcap AL_{2,i}), Msg_1 \cup Msg_2\rangle$ if $\begin{cases} L_1 = \oplus_{i\in I}\, \mathtt{q}_i!\mathsf{m}_i\,.\,AL_{1,i}, \\ L_2 = \oplus_{i\in I}\, \mathtt{q}_i!\mathsf{m}_i\,.\,AL_{2,i} \end{cases}$
- $\left\langle \begin{array}{ll} \&_{i\in I\setminus J}\, \mathtt{q}_i?m_i\,.\,AL_{1,i} & \& \\ \&_{i\in I\cap J}\, \mathtt{q}_i?m_i\,.\,(AL_{1,i} \sqcap AL_{2,i}) & \& \\ \&_{i\in J\setminus I}\, \mathtt{q}_i?m_i\,.\,AL_{2,i} & , \\ Msg_1 \cup Msg_2 & \end{array} \right\rangle$ if $\begin{cases} L_1 = \&_{i\in I}\, \mathtt{q}_i?\mathsf{m}_i\,.\,AL_{1,i}, \\ L_2 = \&_{i\in J}\, \mathtt{q}_i?\mathsf{m}_i\,.\,AL_{2,i}, \\ \forall i \in I \setminus J.\, \mathtt{r} \triangleleft \mathtt{q}_i?\mathsf{m}_i \notin Msg_2, \\ \forall i \in J \setminus I.\, \mathtt{r} \triangleleft \mathtt{q}_i?\mathsf{m}_i \notin Msg_1 \end{cases}$

When no condition applies, the result of the merge is undefined.[2] Note that we use $\sqcap$ for our availability merge operator, which we used as parametric merge operator before. For the sake of clarity, we define only the binary merge. As the operator is commutative and associative, it generalises to a set of branches I. When I is a singleton, the merge just returns that one branch.

[2]When we use the n-ary notation $\sqcap_{i\in I}$ and $|I| = 0$, we implicitly omit this part. Note that this can only happen if $\mathtt{r}$ is the receiver among all branches for some choice so there is either another local type to merge with, or the projection is undefined anyway.

Our availability merge operator builds on the full merge operator presented before. In contrast, it also recurses for sends and, more importantly, it carries the information on available messages along, used when merging receives. When faced with choice, it only merges receptions that cannot interfere with each other. For a participant that ought to receive a message determining a previous branch choice, it checks that the message cannot be available in another branch so actually being able to receive this message uniquely determines which branch was taken by the participant to choose earlier. For the other cases, a participant can postpone learning the branch as long as the events on both branches are the same.

3.2.4 Generalised Projection

We also generalise the projection operator so our availability merge operator can take full effect. First, we account for the annotated local types. This is rather straightforward. One just needs to be a bit careful which participants are initially blocked and which recursion variables have been visited when computing the available messages. Second, we allow to eliminate empty loops, i.e. where a participant does not have any action. This cannot be done on the level of the merge operator but only when projecting. We introduce an additional parameter E for the generalised projection: E contains those variables t for which $\mathtt{r}$ has not observed any message send or receive event since μt.

Definition 3.17 (Generalised projection)**.** The *projection* $G{\upharpoonright}^{E}_{\mathtt{r}}$ of a global type G onto a participant $\mathtt{r} \in \mathcal{P}$ is an availability annotated local type, which is inductively defined:

$$t{\upharpoonright}^{E}_{\mathtt{r}} := \langle t, \text{avail}(\{\mathtt{r}\}, \{t\}, get\mu_G(t))\rangle \qquad 0{\upharpoonright}^{E}_{\mathtt{r}} := \langle 0, \emptyset\rangle$$

$$(\mu t\,.\,G){\upharpoonright}^{E}_{\mathtt{r}} := \begin{cases} \langle \mu t\,.\,(G{\upharpoonright}^{E\cup\{t\}}_{\mathtt{r}}), \text{avail}(\{\mathtt{r}\}, \{t\}, G)\rangle & \text{if } G{\upharpoonright}^{E\cup\{t\}}_{\mathtt{r}} \neq \langle t, _\rangle \\ \langle 0, \emptyset\rangle & \text{otherwise} \end{cases}$$

$$\left(\textstyle\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : \mathsf{m}_i\,.\,G_i\right){\upharpoonright}^{E}_{\mathtt{r}} := \begin{cases} \langle \oplus_{i\in I}\, \mathtt{q}_i!\mathsf{m}_i\,.\,(G_i{\upharpoonright}^{\emptyset}_{\mathtt{r}}), \bigcup_{i\in I} \text{avail}(\{\mathtt{q}_i, \mathtt{r}\}, \emptyset, G_i)\rangle & \text{if } \mathtt{r} = \mathtt{p} \\ \sqcap \begin{pmatrix} \langle \&_{i\in I_{[=\mathtt{r}]}}\, \mathtt{p}?\mathsf{m}_i\,.\,(G_i{\upharpoonright}^{\emptyset}_{\mathtt{r}}), \bigcup_{i\in I_{[=\mathtt{r}]}} \text{avail}(\{\mathtt{r}\}, \emptyset, G_i)\rangle \\ \sqcap_{i\in I_{[\neq\mathtt{r}]} \wedge \forall t\in E.\, G_i{\upharpoonright}^{E}_{\mathtt{r}} \neq \langle t, _\rangle}\, G_i{\upharpoonright}^{E}_{\mathtt{r}} \end{pmatrix} & \text{otherwise} \end{cases}$$
$$\text{where } I_{[=\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i = \mathtt{r}\} \text{ and } I_{[\neq\mathtt{r}]} := \{i \in I \mid \mathtt{q}_i \neq \mathtt{r}\}$$

Since the merge operator $\sqcap$ is partial, the projection may be undefined. We use $G{\upharpoonright}_{\mathtt{r}}$ as shorthand for $G{\upharpoonright}^{\emptyset}_{\mathtt{r}}$ and only consider the generalised projection with empty paths elimination from now on. A global type G is called *projectable* if $G{\upharpoonright}_{\mathtt{r}}$ is defined for every participant $\mathtt{r} \in \mathcal{P}$.

We highlight the differences for the empty paths elimination. Recall that E contains all recursion variables from which the participant $\mathtt{r}$ has not encountered any events. To guarantee this, for the case of recursion $\mu t\,.\,G$, the (unique) variable t is added to the current set E, while the parameter turns to the empty set $\emptyset$ as soon as participant $\mathtt{r}$ encounters an event. The previous steps basically constitute the necessary bookkeeping. The actual elimination is achieved with the condition $\forall t \in E.\, G_i{\upharpoonright}^{E}_{\mathtt{r}} \neq \langle t, _\rangle$ which filters all branches without events of participant $\mathtt{r}$.

Most classical projection operators require all branches of a loop to contain the same set of active participants. Other works [70, 31] achieve this with *connecting actions*, marking non-empty paths. Like classical MSTs, we do not include such explicit actions. Still, we can automatically eliminate such paths in contrast to previous work.

Scalas and Yoshida [109] give an example of local types that are not projectable. We inferred the global type and our the results of our projection operator would be equivalent to the local types given in their example [109, Fig. 4(2)].

Example 3.18 (Two Buyer Protocol with inner recursion)**.** This variant allows to recursively negotiate how to split the price (and omits the outer recursion):

$$G_{\text{2BPIR}} \;:=\; \mathtt{a}\to\mathtt{s}:\mathsf{query}\,.\,\mathtt{s}\to\mathtt{a}:\mathsf{price}\,.\,\mu t\,.\,+\begin{cases}\mathtt{a}\to\mathtt{b}:\mathsf{split}\,.\,(\mathtt{b}\to\mathtt{a}:\mathsf{yes}\,.\,\mathtt{a}\to\mathtt{s}:\mathsf{buy}\,.\,0+\mathtt{b}\to\mathtt{a}:\mathsf{no}\,.\,t)\\ \mathtt{a}\to\mathtt{b}:\mathsf{cancel}\,.\,\mathtt{a}\to\mathtt{s}:\mathsf{no}\,.\,0\end{cases}.$$

Here, seller $\mathtt{s}$ is not involved in the recursion. ◀

Our projection represents a shift in paradigm. In all previous MST frameworks, the merge operator works only on local types. No additional knowledge is required. This is possible because their type system limits the flexibility of communication. Since we allow more flexible communication, we need to keep some information, in form of available messages, about the possible global executions for the merge operator.

3.2.5 Revisiting the Load Balancing Protocols

For the standard load balancing protocol, we need to merge q_2 and q_4 in Fig. 3.7b. From there, the messages form chains, i.e. except for the participant making the choice, a participant only sends in reaction to another message. Thus, the sets of available messages, i.e. the ones that are not intended to be received in each branch, are both empty. The syntactic computations with $\text{avail}(\text{-},\text{-},\text{-})$ yield the same results and allow to merge using $\sqcap$. Hence, the projection of G_{LoBa} onto $\mathtt{c}$ is defined.

Let us consider the variant of the load balancing protocol $G'_{\text{LoBa}}{\upharpoonright}_{\mathtt{c}}$. Here, both replies are available in the left branch, as depicted in Fig. 3.9b. Therefore, our merge operator rejects and the protocol is not projectable.

Let us explain in more detail how the syntactic projection catches this issue. Recall that $\mathtt{c}$ receives result from $\mathtt{w}_2$ in the left branch, which is also present in the right branch. Let us denote the two branches as follows:

$$\begin{aligned}G_1 &:= \;\mathtt{s}\to\mathtt{w}_1:\mathsf{task}\,.\,\mathtt{w}_1\to\mathtt{c}:\mathsf{result}\,.\,\mathtt{w}_1\to\mathtt{w}_2:\mathsf{task}\,.\,\mathtt{w}_2\to\mathtt{c}:\mathsf{result}\,.\,t, \text{ and}\\ G_2 &:= \;\mathtt{s}\to\mathtt{w}_2:\mathsf{task}\,.\,\mathtt{w}_2\to\mathtt{c}:\mathsf{result}\,.\,t\end{aligned}$$

The first message in G_1 does not involve $\mathtt{c}$, so the projection descends and we compute:

$$\begin{aligned}G'_{\text{LoBa}}{\upharpoonright}_{\mathtt{c}} \;&= \langle \mu t\,.\,(\langle \mathtt{w}_1?\mathsf{result}\,.\,(G'_1{\upharpoonright}_{\mathtt{c}}),\, \text{avail}(\{\mathtt{c}\},\emptyset,G'_1)\rangle\\ &\qquad \sqcap\langle \mathtt{w}_2?\mathsf{result}\,.\,(G'_2{\upharpoonright}_{\mathtt{c}}),\, \text{avail}(\{\mathtt{c}\},\emptyset,G'_2)\rangle),_\rangle\\ \text{where}\quad G'_1 &= \mathtt{w}_1\to\mathtt{w}_2:\mathsf{task}\,.\,\mathtt{w}_2\to\mathtt{c}:\mathsf{result}\,.\,t \text{ and } G'_2 = t.\end{aligned}$$

Source	Name	Impl.	Size	$\|\mathcal{P}\|$	Result	Time	Gen. Proj. Needed
[111]	Instrument Contr. Prot. A	✓	16	3	✓	0.1 ms	×
	Instrument Contr. Prot. B	✓	13	3	✓	< 0.1 ms	×
	OAuth2	✓	7	3	✓	< 0.1 ms	×
[108]	Multi Party Game	✓	16	3	✓	0.1 ms	×
[67]	Streaming	✓	7	4	✓	< 0.1 ms	×
[29]	Non-Compatible Merge	✓	8	3	✓	< 0.1 ms	✓
[113]	Spring-Hibernate	✓	44	6	✓	0.6 ms	✓
New	Group Present	✓	43	4	✓	0.5 ms	✓
	Late Learning	✓	12	4	✓	0.1 ms	✓
	Load Balancer ($n = 10$)	✓	32	12	✓	1.9 ms	✓
	Logging ($n = 10$)	✓	56	13	✓	7.7 ms	✓
Section 3.1.1	2 Buyer Protocol	✓	16	3	✓	0.1 ms	×
Section 3.4	2B-Prot. Omit No	✓	14	3	(×)	< 0.1 ms	-
	2B-Prot. Subscription	✓	35	3	(×)	0.3 ms	-
	2B-Prot. Inner Recursion	✓	12	3	✓	< 0.1 ms	×

Table 3.1: Projecting Global Types with Generalised Projection. For every protocol, we report whether it is implementable ✓ or not ×, as well as the outcome as ✓ for "implementable" and (×) for "not known". We also give the size of the protocol (number of states and transitions) and the number of participants.

For this, we compute both availability annotations: $\mathrm{avail}(\{\mathtt{c}\}, \emptyset, G'_2) = \emptyset$ which is empty because all message exchanges in the recursion initiate from $\mathtt{c}$ which is blocked; and $\mathrm{avail}(\{\mathtt{c}\}, \emptyset, G'_1) = \{\mathtt{w}_2 \triangleleft \mathtt{w}_1 ?\mathsf{task}, \mathtt{c} \triangleleft \mathtt{w}_2 ?\mathsf{result}\}$, which contains both receptions from G'_1 only since no receptions from the recursion are added for the same reason. The message for $\mathtt{c}$ in the second branch is available in the first one: $\mathtt{c} \triangleleft \mathtt{w}_2 ?\mathsf{result} \in \mathrm{avail}(\{\mathtt{c}\}, \emptyset, G'_1)$, not satisfying the condition. Thus, the projection is undefined.

3.2.6 Evaluation

We implemented our generalised projection approach in a prototype tool, which is publicly available [106, 114]. The core functionality is implemented in about 800 lines of Python3 code. Our tool takes as input a global type and outputs its projections (if defined). We consider global types (and communication protocols) from five different sources as well as examples from this work (cf. 1st column of Table 3.1). Our experiments were run on a computer with an Intel Core i5-1335U CPU and used at most 100MB of memory. The results are summarised in Table 3.1. The reported size is the number of states and transitions of the respective state machine. Note that our tool does not check for multiple occurences of the same subterm and, thus, the reported size might be bigger than the actual size.

Our prototype successfully projects global types from the literature, in particular Multi-Party Game, OAuth2, Streaming, and two corrected versions of the Instrument Control Protocol. These existing examples can be projected, but do not require generalised projection.

The Need for Generalised Projection. The remaining examples exemplify when our generalised projection is needed. This can have two causes: a sender can send to different receivers or a receiver can receive from different senders along two paths (immediately or after a sequence of same actions). In both cases, its projection is only defined for the generalised projection operator. The Spring-Hibernate example was obtained by translating a UML sequence diagram [113] to a global type. There, `Hibernate Session` can receive from two different senders along two runs. The Group Present example is a variation of the traditional book auction example [68] and describes a protocol where friends organise a birthday present for someone; in the course of the protocol, some people can be contacted by different people. The Late Learning example models a protocol where a participant submits a request and the server replies either with `reject` or `wait`, however, the last case applies to two branches where the result is served by different participants. The Load Balancing (Section 3.2.1) and Logging examples are simple versions of typical communication patterns in distributed computing. The examples are parameterised by the number of workers, respectively, back-ends that call the logging service, to evaluate the efficiency of projection. For both, we present one instance ($n = 10$) in the table. All new examples are rejected by previous approaches but shown projectable by our generalised projection approach.

Overhead. The generalised projection should not incur any overhead for global types that do not need it. Our implementation computes the sets of available messages lazily, i.e. it is only computed if our message causality analysis is needed. These sets are only needed when merging receptions from different senders. Thus, while the message causality analysis is crucial for our generalised projection operator and hence applicability of MST verification, it should not affect its efficiency.

3.3 Soundness of Generalised Projection: Projectability implies Implementability

In this section, we prove soundness of our generalised projection; roughly, a projectable global type can be implemented by communicating state machines in a distributed way.

Theorem 3.19. If a global type G is projectable, then $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ implements G.

Proving this is far from trivial and we cannot use previous proof ideas from the MST literature.

3.3.1 Implementability = Protocol Fidelity + Deadlock Freedom

The CSM of local types should be deadlock-free and generate the same language as the global type. Intuitively, we prove both by showing two properties: first, it does not remove behaviours from the global type; and second, it does not introduce new behaviours, including deadlocks. We start with the first observation.

3.3.2 Generalised Projection Does Not Remove Behaviours

To start with, recall that our projection operator preserves the local order of events for every participant and does not remove or add any event. Therefore, we conclude that, for each participant, the projected language of the global type is subsumed by the language of the projection.

Proposition 3.20. Let G be a projectable global type and $\mathtt{r} \in \mathcal{P}$ be a participant. Then, every trace w of $\mathsf{GAut}(G)\Downarrow_{\Gamma_{\mathtt{r}}}$ is a trace of $\mathsf{LAut}(G{\upharpoonright}_{\mathtt{r}})$ and vice versa. Hence, it holds that $\mathcal{L}(G)\Downarrow_{\Gamma_{\mathtt{r}}} = \mathcal{L}(G{\upharpoonright}_{\mathtt{r}})$.

We use this observation to prove that the CSM $\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ does not remove behaviours of a global type G. Intuitively, we show that $\mathcal{L}(\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{L}(G)$. For the proof, we strengthen this statement.

Lemma 3.21. Let G be a projectable global type. Then, the following holds:

(1) for every prefix $w \in \operatorname{pref}(\mathcal{L}(G))$, there is a run ρ in $\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with $w \preceq \operatorname{trace}(\rho)$ and for every extension wx of w in $\operatorname{pref}(\mathcal{L}(G))$, ρ can be extended to ρ' so that $wx \preceq \operatorname{trace}(\rho')$, and

(2) $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

Proof. For (1), let $w \in \operatorname{pref}(\mathcal{L}(G))$. We prove the claim by induction on the length of w.

The base case, $w = \varepsilon$, is trivial. For the induction step, we assume that the claim holds for w with run ρ and we extend ρ for $wx \in \operatorname{pref}(\mathcal{L}(G))$. We do a case analysis on x.

First, suppose that $x = \mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$ for some $\mathtt{p}$, $\mathtt{q}$, and $\mathtt{m}$. By Proposition 3.20, we know that $(wx)\Downarrow_{\Gamma_{\mathtt{p}}}$ has a run in $\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})$. Therefore, we can simply extend the run ρ in $\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ to obtain ρ'.

Second, suppose that $x = \mathtt{q}\triangleleft\mathtt{p}?\mathtt{m}$ for some $\mathtt{p}$, $\mathtt{q}$, and $\mathtt{m}$. Again, by Proposition 3.20, we know that $(wx)\Downarrow_{\Gamma_{\mathtt{q}}}$ has a run in $\mathsf{LAut}(G{\upharpoonright}_{\mathtt{q}})$. Any prefix of $\mathcal{L}(G)$ is FIFO-compliant by Proposition 2.25. From Lemma 2.5, we know that $(\mathtt{p}, \mathtt{q})$ is of the form $m \cdot u$. Therefore $\mathtt{q}$ can receive $\mathtt{m}$ from $\mathtt{p}$ and we can extend ρ to obtain ρ' for wx in $\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$.

In both cases, we have extended ρ on $w \in \operatorname{pref}(\mathcal{L}(G))$ to $wx \in \operatorname{pref}(\mathcal{L}(G))$, completing the argument for (1).

For (2), let $w \in \mathcal{L}(G)$. We do a case analysis whether w is finite or infinite.

For finite words $w \in \mathcal{L}(G) \cap \Gamma^*$, we know that w is FIFO-compliant and complete (Proposition 2.25). (1) tells us that there is a run for w in $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ that reaches $(\vec{q}, \xi)$ for some states $\vec{q}$ and channel evaluation ξ. We need to show that $(\vec{q}, \xi)$ is a final configuration, i.e. (a) $\vec{q}_{\mathtt{p}}$ is a final state in $\mathsf{LAut}(G\restriction_{\mathtt{p}})$ and (b) all channels in ξ are empty.

To establish (b), we observe that all channels are empty after a run of a finite complete word w, by Lemma 2.5. We know from Proposition 2.25 that w is complete.

For (a), recall that $w \in \mathsf{GAut}(G)$ and w is finite, so the run with trace w in $\mathsf{GAut}(G)$ ends in a final state. By Definition 2.24, every final state corresponds to 0 and has no outgoing transitions. The projection of $0\restriction_{\mathtt{p}}$ is defined as 0 for every $\mathtt{p}$ and 0 is never merged with local types different from 0. The definition of local types prescribes that all different send, resp. receive, options are distinct for every branching. This implies that $\mathsf{LAut}(G\restriction_{\mathtt{p}})$ is deterministic. Therefore, each state $\vec{q}_{\mathtt{p}}$ corresponds to the local type 0, which are final in $\mathsf{LAut}(G\restriction_{\mathtt{p}})$. Thus, $(\vec{q}, \xi)$ is a final configuration of $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, so $w \in \mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

Suppose that w is infinite, i.e. $w \in \mathcal{L}(G) \cap \Gamma^\omega$. We show that w has an infinite run in $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, analogous to the proof of Lemma 2.10. Consider a tree $\mathcal{T}$ where each node corresponds to a run ρ on some finite prefix $w' \leq w$. The root is labelled by the empty run. The children of a node ρ are runs that extend ρ by a single transition – these exist by (1). Since our CSM, derived from a global type, comprises a finite number of finite state machines, $\mathcal{T}$ is finitely branching. By König's Lemma, there is an infinite path in $\mathcal{T}$ that corresponds to an infinite run for w in $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ so $w \in \mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$. ☐

This shows that the projection does not remove behaviours.

3.3.3 Generalised Projection Does Not Introduce New Behaviours

We proceed with the second observation: no new behaviours are introduced.

Key Ideas. We first give a brief summary of the main ideas. So far, we showed that the projections combine to admit at least the behaviour specified by the global protocol. For the converse direction, we establish a property of the executions of the local types with respect to the global type: during any run of the projections, all the participants agree on some (not necessarily maximal) run of the global type. For this, we introduce the idea of *run mappings*, which maps every participant's observation to a prefix of a run of the global type. While this ensures that no participant has yet diverged from some common run in the global type, it does not guarantee that this run can be extended to be maximal and all participants can catch up. Intuitively, run mappings do only ensure that nothing bad has happened but not that something good will eventually happen, basically distinguishing between safety and liveness. Adding this liveness aspect yields *control flow agreement*. Runs that satisfy control flow agreement also satisfy protocol fidelity and are deadlock-free. The formalisation and proofs of these properties is complicated by

the fact that not all participants learn about a choice at the same time. Some participants can perform actions after the choice has been made and before they learn which branch has been taken. In the extreme case, a participant may not learn at all that a choice happened. The key to control flow agreement is in the definition of the merge operator. We can simplify the reasoning to the following two points.

- **Participants learn choices before performing distinguishing actions**
  When faced with two branches with different actions, a participant that is not making the choice needs to learn the branch by receiving a message. This follows from the definition of the merge operator. This message is also called choice message. Merging branches is only allowed as long as the actions are similar for this participant. When there is a difference between two (or more) branches, an external choice is the only case that allows a participant to continue on distinct branches.
- **Checking available messages ensures no confusion**
  From the possible receives ($\mathtt{q}_i ? \mathsf{m}_i$) in an external choice, any pair of sender and message is unique among this list for the choice. This follows from two facts. First, the projection computes the available messages along the different branches of the choice. Second, merging uses that information to make sure that the choice message of one branch does not occur in another branch as a message independent of that branch's choice messages.

Example 3.22. This example illustrates why this is non-trivial. Consider

$$G := \big(\mathtt{p} \to \mathtt{q} : \mathsf{l} \,.\, \mu t \,.\, \mathtt{r} \to \mathtt{p} : \mathsf{m} \,.\, t\big) + \big(\mathtt{p} \to \mathtt{q} : \mathsf{r} \,.\, \mu s \,.\, \mathtt{r} \to \mathtt{p} : \mathsf{m} \,.\, s\big)$$

with its projections

$$\begin{aligned} G{\upharpoonright}_{\mathtt{p}} &= \big(\mathtt{q}!\mathsf{l}.\, \mu t \,.\, \mathtt{r}?\mathsf{m} \,.\, t\big) \;\oplus\; \big(\mathtt{q}!\mathsf{r}.\, \mu s.\, \mathtt{r}?\mathsf{m} \,.\, s\big) \\ G{\upharpoonright}_{\mathtt{q}} &= \mathtt{p}?\mathsf{l} \,.\, 0 \;\&\; \mathtt{p}?\mathsf{r} \,.\, 0 \\ G{\upharpoonright}_{\mathtt{r}} &= \mu t \,.\, \mathtt{p}!\mathsf{m} \,.\, t \end{aligned}$$

and a trace w of $\{\!\{\mathsf{LAut}(G{\upharpoonright}_{\mathtt{p}})\}\!\}_{\mathtt{p} \in \mathcal{P}}$: $\mathtt{r}{\triangleright}\mathtt{p}!\mathsf{m} \cdot \mathtt{r}{\triangleright}\mathtt{p}!\mathsf{m} \cdot \mathtt{p}{\triangleright}\mathtt{q}!\mathsf{l} \cdot \mathtt{p}{\triangleleft}\mathtt{r}?\mathsf{m} \cdot \mathtt{r}{\triangleright}\mathtt{p}!\mathsf{m}$. For this trace, we check which runs in $\mathsf{GAut}(G)$ each participant could have pursued. In this case, $\mathtt{r}$ is not directly affected by the choice so it can proceed before the $\mathtt{p}$ has even made the choice. As the part of the protocol after the choice is a loop, we cannot bound how far $\mathtt{r}$ can proceed before $\mathtt{p}$ commits a choice. ◀

3.3.4 Family of Run Mappings Exists for Projectable Global Types

Let us define run mappings and families thereof first.

Definition 3.23 (Run mappings)**.** Let G be a global type and $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM. For a trace w of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ and a run ρ in $\mathsf{GAut}(G)$ such that $w \preceq_{\sim} \mathrm{trace}_{\mathsf{GAut}(G)}(\rho)$, a ***run mapping*** $\rho_{\mathcal{P}} \colon \mathcal{P} \to \mathrm{pref}(\rho)$ is a mapping such that $w{\Downarrow}_{\Gamma_{\mathtt{p}}} = \mathrm{trace}_{\mathsf{GAut}(G)}(\rho_{\mathcal{P}}(\mathtt{p})){\Downarrow}_{\Gamma_{\mathtt{p}}}$.

We say that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ has a *family of run mappings* for $\mathsf{GAut}(G)$ iff for every trace w of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$, there is a *witness run* ρ in $\mathsf{GAut}(G)$ with a run mapping $\rho_{\mathcal{P}}$.

When clear from context, we omit the subscript of run mappings $\rho_{\mathcal{P}}(\text{-})$.

Given a global type G and a trace of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$, a run mapping guarantees that there is some common run in $\mathsf{GAut}(G)$ that all participants could have pursued when observing this trace. They might not have processed all their events along this run but they have not diverged. A family of run mappings lifts this to all traces. However, this does not yet ensure that every participant will be able to follow this run to the end. This will be part of *Control Flow Agreement.*

Lemma 3.24 ($\{\mathsf{LAut}(G{\restriction_{\mathtt{p}}})\}_{\mathtt{p}\in\mathcal{P}}$ has a family of run mappings)**.** Let G be a projectable global type. Then, $\{\!\{\mathsf{LAut}(G{\restriction_{\mathtt{p}}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ has a family of run mappings for $\mathsf{GAut}(G)$.

The proof is not trivial because the merge can collapse different branches for a participant so it follows sets of runs. The intersection of all these sets contains (a prefix of) the witness run. In some case, e.g. $\mathtt{r}$ in the Example 3.22, a participant can "overtake" a choice and perform actions that belong to the branches of a choice before the choosing participant has made the choice.

Outline and Motivation of Concepts

The overall proof works by induction on the trace. While the skeleton is quite basic, let us give an intuition which properties are needed for the cases where some participant receives or sends a message in the extension of a trace. Based on these properties, we can explain the different concepts which need to be formalised for these properties. In both cases, we need to argue that the previous run mapping can be kept for all participants but the one that takes a step.

For the receive case, suppose that $\mathtt{p}$ receives from $\mathtt{q}$. On the one hand, we need to argue that the run for $\mathtt{p}$ can actually be extended to match the one of $\mathtt{q}$. On the other hand, we also need to ensure that there is no other message that $\mathtt{p}$ could receive so that it follows a run different from the one for $\mathtt{q}$. While the first is easy to obtain by the definition of projection, we need to argue that the message availability check does indeed compute the available messages when $\mathtt{p}$ has not proceeded with any other event yet.

For the send case, suppose that p sends to q and chooses between different branches, i.e. there actually is a choice between different continuations. Though, there might be participants with (some) actions that do not depend on this choice and we have to ensure that p's run can be extended to some prefix of theirs. For this, we exploit the idea of prophecy variables and assume that all of those participants took the branch that p will choose but we need to ensure that this does not restrain them in any way. So intuitively for each of these participants, we need to compare the executions along the different branches p can choose from and need to prove that they are the same if p has not yet committed this choice yet. (Note the subtlety that only the part of the execution which does not depend on this choice needs to be the same.)

To formalise these ideas, we introduce three concepts.

- *Blocked languages:* We inductively define the (global) language $L^{\mathcal{B}}_{(G' \ldots)}$ that captures all executions that are possible from subterm G' in some global type G, under the assumption that no participant in $\mathcal{B} \subseteq \mathcal{P}$ can take any further step. We call these participants *blocked.* This definition serves for the formalisation of semantic arguments about the availability of messages and the languages along different choice branches.
- *Blocked projection operator:* We introduce a variation of the projection operator that also allows to account for a set of blocked participants $\mathcal{B}$, prove its connection to the above languages, and use the relation to the standard projection operator to show the equality of the blocked languages along different choices.
- *Extended local types:* In contrast to the standard projection operator, the blocked one can declare intermediate syntactic subterms as final states. Standard local types do not allow this so we introduce extended local types.

Blocked Languages – the Language-theoretic Perspective

We will use blocked languages to show that no participant can proceed with an action that is specific to some choice which has not been committed yet.

Definition 3.25 (Blocked language)**.** Let G be a projectable global type and let $\mathcal{B} \subseteq \mathcal{P}$ be a set of participants. Then, $L^{\mathcal{B}}_{(G \ldots)}$ is inductively defined:

$L^{\mathcal{B}}_{(0 \ldots)} := \{\varepsilon\}$ $\qquad L^{\mathcal{B}}_{(t \ldots)} := L^{\mathcal{B}}_{(\mu t \,.\, G \ldots)}$ $\qquad L^{\mathcal{B}}_{(\mu t \,.\, G \ldots)} := L^{\mathcal{B}}_{(G \ldots)}$

$L^{\mathcal{B}}_{(\Sigma_{i \in I} \mathsf{p} \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i \ldots)} := \bigcup_{i \in I} \{ f(\mathcal{B}, \mathsf{p} \to \mathsf{q}_i : \mathsf{m}_i) \cdot w \mid w \in L^{g(\mathcal{B}, \mathsf{p} \to \mathsf{q}_i : \mathsf{m}_i)}_{(G_i \ldots)} \}$ where

$$f(\mathcal{B}, \mathsf{p} \to \mathsf{q}_i : \mathsf{m}_i) := \begin{cases} \mathsf{p} \triangleright \mathsf{q}_i ! \mathsf{m}_i \,.\, \mathsf{q}_i \triangleleft \mathsf{p} ? \mathsf{m}_i & \text{if } \mathsf{p} \notin \mathcal{B} \land \mathsf{q}_i \notin \mathcal{B} \\ \mathsf{p} \triangleright \mathsf{q}_i ! \mathsf{m}_i & \text{if } \mathsf{p} \notin \mathcal{B} \land \mathsf{q}_i \in \mathcal{B} \\ \varepsilon & \text{otherwise} \end{cases}$$

$$g(\mathcal{B}, \mathsf{p} \to \mathsf{q}_i : \mathsf{m}_i) := \begin{cases} \mathcal{B} & \text{if } \mathsf{p} \notin \mathcal{B} \\ \mathcal{B} \cup \{\mathsf{q}_i\} & \text{otherwise} \end{cases}$$

Based on this definition, we compute the set of messages that can occur when some participants, i.e. the ones in $\mathcal{B}$ are blocked:

$$\text{msgs}^{\mathcal{B}}_{(G\ldots)} := \bigcup_{x = x_1 \ldots \in L^{\mathcal{B}}_{(G\ldots)}} \{x_i \mid x_i \in \Gamma_?\} \enspace .$$

If $\mathcal{B}$ is a singleton set, we omit the set notation and write $\text{msgs}^{\mathtt{p}}_{(G\ldots)}$ for $\text{msgs}^{\{\mathtt{p}\}}_{(G\ldots)}$.

Extended Local Types and Blocked Projection Operator – the Type-theoretic Perspective

Standard global and local types have a dedicated syntactic term to indicate termination: 0. For the FSMs of their semantics, this means that a state is final if and only if it has not outgoing transitions. For our proofs, we need to extend local types in a way that they can have intermediate termination.

Definition 3.26 (Extended local types)**.** We extend local types with marks $*$ or $\circ$ for syntactic subterms and obtain *extended local types* with the following inductive rules where $\circledast \in \{\circ, *\}$:

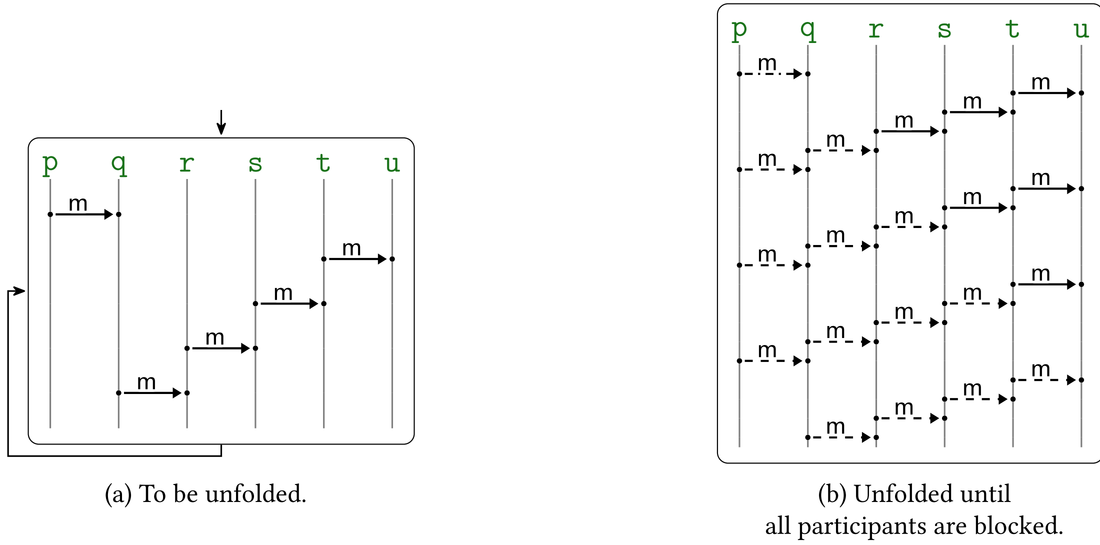
$$EL ::= (\!| 0, \circledast |\!) \;\mid\; (\!| \bigoplus_{i \in I} \mathsf{q}_i ! \mathsf{m}_i \,.\, EL_i, \circledast |\!) \;\mid\; (\!| \mathop{\&}_{i \in I} \mathsf{q}_i ? \mathsf{m}_i \,.\, EL_i, \circledast |\!) \;\mid\; (\!| \mu t \,.\, EL, \circledast |\!) \;\mid\; (\!| t, \circledast |\!)$$

Extended local types can be annotated with availability information about messages as local types. When working with message set annotations, we assume to have extended availability annotated local types of shape $(\!| \langle \text{-}, \text{-} \rangle, \text{-} |\!)$. Thus, we can re-use the merge operator $\sqcap$.

The mark $*$ indicates that this syntactic subterm shall be considered final in the semantics of the extended local types. Any syntactic subterm $(\!| EL', \circledast' |\!)$ of $(\!| EL, \circledast |\!)$ has a syntax tree. Intuitively, $\circledast'$ marks the root of the syntax tree of EL' and tells whether one needs to continue, i.e. $(\!| EL', \circ |\!)$, to obtain an accepting word or may stop, i.e. $(\!| EL', * |\!)$, at this point. Without such an extension, there is no way to specify such behaviour in local types (and global type) since 0 explicitly marks the end of the type. We consider two subterms equal if they agree on all (nested) markings.

Definition 3.27 (Semantics of extended local types)**.** We define the semantics of an extended local type EL analogously to the one for local types, except that there can be multiple final states as explained before. Let EL be an extended local type for $\mathtt{p}$. We construct a state machine $\mathsf{EAut}(EL) = (Q, \Gamma_{\mathtt{p}}, \delta, q_0, F)$ and define the language of EL as language of this state machine: $\mathcal{L}(EL) = \mathcal{L}(\mathsf{EAut}(EL))$.

It is straightforward to turn an extended (availability annotated) local type into a (availability annotated) local type by recursively projecting onto the first component of the extended local type and hence ignoring the markers $*$ and $\circ$. Therefore, we apply this implicit coercion for operations that are only defined on (availability annotated) local types.



(a) To be unfolded.

(b) Unfolded until all participants are blocked.

Figure 3.10: An HMSC and its unfolding where all participants are blocked at the end: if the first (single-dashed) message m is blocked, the ones with solid arrow can still be sent while the dashed ones cannot as their sender is blocked. (Note that, here, line style indicates if a message is blocked, not the message sent.)

Example 3.28. Consider the HMSC in Fig. 3.10a and its unfolding in Fig. 3.10b. The unfolding exemplifies that one needs to unfold the recursion three times until all participants get blocked. Still, if there were two more participants that would send messages to each other independently, one needs to account for these messages with a recursion. ◀

The set of blocked participants does only increase. If some message is blocked, two participants are blocked from the beginning and one iteration of the recursion is always present in any type. In general, a fixed point is reached after $|\mathcal{P}| - 2$ unfoldings.

Definition 3.29 (Blocked projection operator)**.** Let G be some global projectable global type, E a set of recursion variables, $\mathcal{B} \subseteq \mathcal{P}$ a set of participants and n some natural number. Let T be a set of recursion variables and $\eta\colon T \to 2^{\mathcal{P}} \times \mathbb{N}$. We define the blocked projection $\upharpoonright^{E,\mathcal{B}}$ of G parametrised by n onto a participant $\mathtt{r}$ inductively:

$$0\upharpoonright^{E,\mathcal{B}}_{\mathtt{r},n} := (\!| 0\upharpoonright_{\mathtt{r}}, * |\!) \qquad (\mu t\,.\,G')\upharpoonright^{E,\mathcal{B}}_{\mathtt{r},n} := \begin{cases} (\!| \langle \mu t\,.\,(G\upharpoonright^{E\cup\{t\}}_{\mathtt{r}}), \text{avail}(\{R\},\{t\},G)\rangle, \circ |\!) \text{ and } \eta(t) := (\mathcal{B},0) & \\ \qquad \text{if } G\upharpoonright^{E\cup\{t\}}_{\mathtt{r}} \neq (\!| \langle t, _\rangle, _ |\!) \text{ and } t \text{ occurs in } G\upharpoonright^{E\cup\{t\}}_{\mathtt{r}} & \\ \langle (G\upharpoonright^{E\cup\{t\}}_{\mathtt{r}}), \text{avail}(\{R\},\{t\},G)\rangle & \text{if } t \text{ does not occur in } G\upharpoonright^{E\cup\{t\}}_{\mathtt{r}} \\ (\!| \langle 0,\emptyset\rangle, * |\!) & \text{otherwise} \end{cases}$$

$$t\upharpoonright^{E,\mathcal{B}}_{\mathtt{r},n} := \begin{cases} (\!| \langle \mu t'\,.\,((get\mu_G(t))[t'/t])\upharpoonright^{E,\mathcal{B}}_{\mathtt{r},n}, \text{avail}(\{\mathtt{r}\},\{t\},get\mu_G(t))\rangle, \circ |\!) \text{ and } \eta(t') := (\mathcal{B}, \eta_2(t)+1) & \\ \qquad \text{if } \eta_1(t) \neq \mathcal{B} \wedge \eta_2(t) < n & \\ (\!| \langle t, \text{avail}(\{\mathtt{r}\},\{t\},get\mu_G(t))\rangle, \circ |\!) & \text{if } \eta_1(t) = \mathcal{B} \wedge \eta_2(t) \leq n \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\left(\sum_{i\in I} \mathtt{p}\to\mathtt{q}_i : \mathsf{m}_i \,.\, G_i\right)\restriction^{E,\mathcal{B}}_{\mathtt{r},n} := \begin{cases} \llparenthesis\, \langle \oplus_{i\in I}\, \mathtt{q}_i!\mathsf{m}_i \,.\, (G_i\restriction^{E,\mathcal{B}}_{\mathtt{r},n}), \bigcup_{i\in I} \operatorname{avail}(\{R\},\emptyset,G_i)\rangle, \circ \,\rrparenthesis & \\ & \text{if } \mathtt{p}\notin\mathcal{B}\wedge\mathtt{r}=\mathtt{p} \\ \bar{\sqcap}\begin{cases} \llparenthesis\, \langle \&_{i\in I\wedge\mathtt{q}_i=\mathtt{r}\wedge\mathtt{r}\notin\mathcal{B}}\, \mathtt{p}?\mathsf{m}_i \,.\, (G_i\restriction^{E,\mathcal{B}}_{\mathtt{r},n}), \bigcup_{i\in I} \operatorname{avail}(\{\mathtt{r}\},\emptyset,G_i)\rangle, \circ \,\rrparenthesis \\ \bar{\sqcap}_{i\in I\wedge\mathtt{q}_i=\mathtt{r}\wedge\mathtt{r}\in\mathcal{B}}\ \llparenthesis\, \langle 0,\emptyset\rangle, * \,\rrparenthesis \\ \bar{\sqcap}_{i\in I\wedge\mathtt{q}_i\neq\mathtt{r}\wedge\forall t\in E.\, G_i\restriction^{E,\mathcal{B}}_{\mathtt{r},n}\neq\llparenthesis\, \langle t,_\rangle,_\,\rrparenthesis}\ G_i\restriction^{E,\mathcal{B}}_{\mathtt{r},n} \end{cases} & \\ & \text{if } \mathtt{p}\notin\mathcal{B}\wedge\mathtt{r}\neq\mathtt{p} \\ \llparenthesis\, \langle 0,\emptyset\rangle, * \,\rrparenthesis & \text{if } \mathtt{p}\in\mathcal{B}\wedge\mathtt{r}=\mathtt{p} \\ \bar{\sqcap}_{i\in I}\, G_i\restriction^{E,\mathcal{B}\cup\{\mathtt{q}_i\}}_{\mathtt{r},n} & \text{if } \mathtt{p}\in\mathcal{B}\wedge\mathtt{r}\neq\mathtt{p} \end{cases}$$

where $\bar{\sqcap}$ is defined using $\sqcap$:

- for any extended availability annotated local type $\llparenthesis\, \langle EL, Msg\rangle, \circledast \,\rrparenthesis$, it holds that $\llparenthesis\, \langle EL, Msg\rangle, \circledast \,\rrparenthesis \;\bar{\sqcap}\; \llparenthesis\, \langle 0,\emptyset\rangle, * \,\rrparenthesis := \llparenthesis\, \langle EL, Msg\rangle, * \,\rrparenthesis$ for $\circledast \in \{*, \circ\}$, and
- for any two extended availability annotated local types $\llparenthesis\, \langle EL_1, Msg_1\rangle, \circledast_1 \,\rrparenthesis$ and $\llparenthesis\, \langle EL_2, Msg_2\rangle, \circledast_2 \,\rrparenthesis$, it holds that $\llparenthesis\, \langle EL_1, Msg_1\rangle, \circledast_1 \,\rrparenthesis \;\bar{\sqcap}\; \llparenthesis\, \langle EL_2, Msg_2\rangle, \circledast_2 \,\rrparenthesis := \llparenthesis\, \langle (EL_1 \sqcap EL_2), Msg\rangle, \circledast' \,\rrparenthesis$ where $\circledast_1, \circledast_2, \circledast' \in \{*, \circ\}$, $\circledast' = *$ iff $\circledast_i = *$ for any $i \in \{1,2\}$ and $\langle _, Msg\rangle := \langle EL_1, Msg_1\rangle \sqcap \langle EL, Msg_2\rangle$.

By assumption, every variable $t \in T$ is only bound once and is bound before use in any global type. Therefore $\eta(t)$ is always defined. The blocked projection returns an extended availability annotated local type. Erasing the annotations yields an extended local type.

The special rule for recursion ensures that we unfold the definition (equi-recursively) if the set $\mathcal{B}$ changed since the recursion variable has been bound. As for the standard projection, we define $G\restriction^{\mathcal{B}}_{\mathtt{r},n} := G\restriction^{\emptyset,\mathcal{B}}_{\mathtt{r},n}$.

When we want to emphasise the difference, we call projection $\restriction__$ the *standard* projection, in contrast to the *blocked* projection. The blocked projection ignores all actions that depend on actions by any participant in $\mathcal{B}$. In the course of computation, $\mathcal{B}$ grows and represents the set of participants that are not able to proceed any further since their actions depend on some previous action from a blocked participant. With each unfolding, the set either grows or there is no unfolding necessary anymore. Therefore, it holds that $\forall n \geq |\mathcal{P}| - |\mathcal{B}|.\ G\restriction^{\mathcal{B}}_{\mathtt{r},n} = G\restriction^{\mathcal{B}}_{\mathtt{r},n+1}$. In case n is greater than this threshold, we omit it for readability.

By definition, the blocked projection operator mimcs the standard projection operator for $\mathcal{B} = \emptyset$ and $n = 0$.

Proposition 3.30. For any global type G and $\mathtt{r}$, it holds that $G\restriction^{\emptyset}_{\mathtt{r},0} = G\restriction_{\mathtt{r}}$.

Remark 3.31 (Different use of blocked participants)**.** The blocked projection operator and the message causality analysis both share the concept of a growing set of blocked participants. Despite, it does not seem beneficial to unify both approaches. First, we would establish a cyclic dependency between the projection operator and the message causality analysis used when merging. Second, the message availability analysis fixes a participant $\mathtt{r}$ that gets blocked. However, the blocked merge operator would return 0 as soon as it encounters $\mathtt{r}$ again which will not yield the desired information about the channel contents.

Correspondence of Blocked Projection Operator and Blocked Languages

Lemma 3.32 (Correspondence of blocked projection operator and blocked languages)**.** Let G be a projectable global type, G' a syntactic subterm of G, and $\mathtt{r} \in \mathcal{P}$ be a participant. Then, for every subset of participants $\mathcal{B} \subseteq \mathcal{P}$, it holds that

$$L^{\mathcal{B}}_{(G' \ldots)} \Downarrow_{\Gamma_{\mathtt{r}}} = \mathcal{L}(G' \upharpoonright^{\mathcal{B}}_{\mathtt{r}}) \ .$$

Proof. Let T' be all recursion variables in G' that are not bound in G'. We define $\widehat{G'}$ to be the type that is obtained by substituting every $t \in T'$ by $get\mu_G(t)$. By construction, $\widehat{G'}$ is a global type (and not only a syntactic subterm of one). Then, it suffices to show that, for every subset of participants $\mathcal{B} \subseteq \mathcal{P}$, $L^{\mathcal{B}}_{(\widehat{G'} \ldots)} \Downarrow_{\Gamma_{\mathtt{r}}} = \mathcal{L}(\widehat{G'} \upharpoonright^{\mathcal{B}}_{\mathtt{r}})$ where $\widehat{G'}$ can be considered to be the overall global type for $\mathsf{GAut}(\widehat{G'})$ as well. We prove this claim by induction on $\widehat{G'}$.

There are two base cases. For 0, the claim trivially holds and t is no global type which $\widehat{G'}$ is by assumption.

For the first induction step, let $\widehat{G'} = \mu t \,.\, \widehat{G''}$ and the induction hypothesis holds for $\widehat{G''}$. Both definitions agree on the fact that μt itself does not carry any event. The blocked projection ensures to only bind a recursion variable if $\mathtt{r}$ does use it later on. There is no such need in the blocked language along paths. We do the same case analysis as the blocked projection operator. First, if t is used after some actual event, t is bound and the induction hypotheses concludes the claim. Second, if t is never used, t is not bound and the blocked language along the path yields the same language by induction hypothesis. Third, otherwise, i.e. if the projection of the continuation yields $\langle 0, _ \rangle$, the blocked language along path enters an empty loop and the claim follows.

For the second induction step, let $\widehat{G'} = \Sigma_{i \in I} \mathtt{p} \to \mathtt{q}_i : \mathsf{m}_i \,.\, \widehat{G'_i}$ and the induction hypothesis holds for $\widehat{G'_i}$ for every $i \in I$. As for the previous case, we apply the same case analysis as for the blocked projection operator: to be more precise, a combined analysis if $\mathtt{p} \in \mathcal{B}$ and if $\mathtt{r} = \mathtt{p}$.

(i) Suppose that $\mathtt{p} \notin \mathcal{B}$ and $\mathtt{p} = \mathtt{r}$. On both sides, $\mathcal{B}$ does not change in the recursive calls. Thanks to the projection $\Downarrow_{\Gamma_{\mathtt{r}}}$, the same event $\mathtt{p} \triangleright \mathtt{q}_i ! \mathsf{m}_i$ is added

before recursing. (Due to the projection for the blocked language along a path, there is no need for a case analysis for $q_i \in \mathcal{B}$).

(ii) Suppose that $p \notin \mathcal{B}$ and $p \neq r$.
For this case, we consider each branch $i \in I$ individually and do another case analysis whether $r = q_i$.
For $r \neq q_i$, no events are added for the blocked projection operator (and branches of empty loops, yielding empty languages along the path, ignored for further projection) and the ones added for the blocked language along a path are projected away.
For $r = q_i$, we need another case split and check whether $r \in \mathcal{B}$.
If so, the blocked language along a path recurses and produces a language $\{\varepsilon\}$ with this, the blocked projection operator stops immediately and returns $\langle (\!| 0, \emptyset |\!), * \rangle$ which produces the same language.
If not, both sides add the same receive event for r (and the send event is projected away).

(iii) Suppose that $p \in \mathcal{B}$ and $p = r$. On both sides, the result is $\{\varepsilon\}$.

(iv) Suppose that $p \in \mathcal{B}$ and $p \neq r$. On both sides, no event is added but the recursions add q_i to the set $\mathcal{B}$. These recursions are also covered by the induction hypotheses because we quantified over $\mathcal{B}$.

□

Properties of Projectable Global Types

In this section, we establish properties that we will use in the induction step of the proof for Lemma 3.24 – one for the send and one for the receive case.

Lemma 3.33 (Property of Projectable Global Types I – Send)**.** Let G be a projectable global type and $G = \Sigma_{i \in I}\, p \to q_i : m_i \,.\, G_i$ be some syntactic subterm. Then, for all $i, j \in I$ and $r \in \mathcal{P}$, it holds that $L^{\{p,q_i\}}_{(G_i \ldots)} \Downarrow_{\Gamma_r} = L^{\{p,q_j\}}_{(G_j \ldots)} \Downarrow_{\Gamma_r}$.

Proof. With Lemma 3.32, it suffices to show that $G_i \upharpoonright^{\{p,q_i\}}_r = G_j \upharpoonright^{\{p,q_j\}}_r$ for all i, j and r. First, we take care of the two special cases where r is either p or q_i for some $i \in I$.

On the one hand, if $r = p$, it is straightforward that $(\!| \langle 0, \emptyset \rangle, * |\!)$ is the result of both projections. On the other hand, assume without loss of generality $r = q_i$ for some $i \in I$. Then, it holds that $G_i \upharpoonright^{\{p,q_i\}}_r = (\!| \langle 0, \emptyset \rangle, * |\!)$. Towards a contradiction, assume there is a $j \in I$ such that $G_i \upharpoonright^{\{p,q_i\}}_r \neq G_j \upharpoonright^{\{p,q_j\}}_r$. Since G is projectable, we know that the $G_i \upharpoonright_r \sqcap G_j \upharpoonright_r$ is defined and therefore, r cannot send as first action in the projection for G_j. Therefore, it needs to receive. However, then this message must be unique for this branch and by definition, the choice which branch to take is blocked in $G_j \upharpoonright^{\{p,q_j\}}_r$ and such a message cannot exist in the blocked projection. Therefore, both projections are $(\!| \langle 0, \emptyset \rangle, * |\!)$.

For the case where $\mathtt{r} \neq \mathtt{p}$ and $\mathtt{r} \neq \mathtt{q}_i$ for all $i \in I$, we introduce a slight variation $\dot{\sqcap}$ of $\sqcap$ to state a sufficient claim. The merge operator $\dot{\sqcap}$ is defined as $\sqcap$ but does only merge $\circledast_1$ and $\circledast_2$ if $\circledast_1 = \circledast_2$ and does require that $I = J$ when merging receives. Then, $\dot{\sqcap}$ does only merge extended local types that are exactly the same by definition.

Hence, it suffices to show that $G_i \upharpoonright^{\{\mathtt{p},\mathtt{q}_i\}}_{\mathtt{r}} \dot{\sqcap}\, G_j \upharpoonright^{\{\mathtt{p},\mathtt{q}_j\}}_{\mathtt{r}}$ is defined for all $i, j,$ and $\mathtt{r}$.

For this, we first show that the set of blocked participants $\mathcal{B}$ in two branches will be the same if some recursion variable is merged with $\dot{\sqcap}$. Intuitively, this is true because the $\sqcap$-operator does not unfold recursions and a participant is either agnostic to some choice (merging two recursion variables) or has learnt about the choice before encountering the recursion variable.

Claim I: For all i, j, and $\mathtt{r}$, if $t\upharpoonright^{\mathcal{B}_i}_{\mathtt{r}} \dot{\sqcap}\, t\upharpoonright^{\mathcal{B}_j}_{\mathtt{r}}$ occurs in the computation of the above merge with $\dot{\sqcap}$, then $\mathcal{B}_i = \mathcal{B}_j$.

Proof of Claim I.
(Note that in the presence of empty loops, the definition prevents such kinds of merge by definition.) Recall that G_i and G_j are two branches of the same choice in which $\mathtt{r}$ is not involved. Towards a contradiction, suppose that there are i, j and $\mathtt{r}$ such that $\mathcal{B}_i \neq \mathcal{B}_j$. Without loss of generality, there must be some participant $\mathtt{s}$ such that $\mathtt{s} \in \mathcal{B}_i$ and $\mathtt{s} \notin \mathcal{B}_j$. It is obvious that $\mathtt{s} \notin \{\mathtt{p}, \mathtt{q}_j\}$. We do a case analysis whether $\mathtt{s} = \mathtt{q}_i$.

If so, $\mathtt{s}$ receives in branch i and since $\sqcap$ is defined, it must also receive in every continuation specified by G_j – otherwise t would be merged with a receive which is undefined. (There could be subsequent branches in G_j and therefore $\mathtt{s}$ could potentially receive different messages but at least one in every continuation.) By definition, these possible receives must be unique and therefore distinct from m_i by $\mathtt{p}$ and all possible messages for $\mathtt{s}$ in the continuation of G_j, ensured by the available messages. Because of this uniqueness, the send event for any receive event in G_j must not be possible in G_i. But for this, the send event must depend on the decision by $\mathtt{p}$ and therefore $\mathtt{s} \in \mathcal{B}_j$ which yields a contradiction.

Suppose that $\mathtt{s} \neq \mathtt{q}_i$. By assumption, $\mathtt{s} \in \mathcal{B}_i$ so there must be some receive for $\mathtt{s}$ for which the corresponding send event depends on $\mathtt{p}\triangleright\mathtt{q}_i!\mathsf{m}_i$. From here, the reasoning is analogous to the previous case.

End Proof of Claim I.

In Claim I, $\mathcal{B}_i$ and $\mathcal{B}_j$ capture all participants that depend on the choice by $\mathtt{p}$ (without unfolding the recursion). The standard projection and merge operator $\sqcap$ do not unfold any recursions and $t \sqcap t$ is the only rule to merge recursion variables. Therefore, a participant either has to learn about a choice by receiving some unique message before recursing or will never learn about it (since t maps back to the same continuation). Combined with Claim I, it suffices to show that $G_i \upharpoonright^{\{\mathtt{p},\mathtt{q}_i\}}_{\mathtt{r}} \dot{\sqcap}\, G_j \upharpoonright^{\{\mathtt{p},\mathtt{q}_j\}}_{\mathtt{r}}$ does not return undefined until the first recursion variable is encountered, for all i, j, and $\mathtt{r}$. Towards a contradiction, suppose it does return undefined for some i, j, and $\mathtt{r}$.

We check all possibilities why $\dot{\sqcap}$ can be undefined.

To start with, either of both projections $\upharpoonright$ can return undefined when the parameter n is too small but we choose a sufficiently big enough n, making this impossible.

Let us define some notation: for $l \in \{i, j\}$, let $(\!|\langle EL_l, Msg_l\rangle, \circledast_l|\!) = G_l \upharpoonright_{\mathtt{r}}^{\{\mathtt{p}, \mathtt{q}_i\}}$. Then, $(\!|\langle EL_i, Msg_i\rangle, \circledast_i|\!) \sqcap\!\!\!\!\sqcap (\!|\langle EL_j, Msg_j\rangle, \circledast_j|\!)$ is undefined if $\circledast_i \neq \circledast_j$ (first case) or $EL_i \sqcap\!\!\!\!\sqcap EL_j$ (second case) is undefined.

The **first case** can only occur if $(\!|\langle 0, \emptyset\rangle, *|\!)$ is merged on top level in the projection of one branch and not the other. Without loss of generality, let $G_i \upharpoonright_{\mathtt{r}}^{\{\mathtt{p}, \mathtt{q}_i\}}$ be the branch where this happens. On the one hand, this occurs when $\mathtt{r}$ receives from a participant in $\mathcal{B}_i$ and since $G_i \upharpoonright_{\mathtt{r}} \sqcap G_j \upharpoonright_{\mathtt{r}}$ is defined by assumption, $\mathtt{r}$ also receives in the branch of j by definition of $\sqcap$. For the standard merge $\sqcap$, both messages are checked to be unique for both branches. However, if $\mathtt{r}$ can still receive the message in $G_j \upharpoonright_{\mathtt{r}}^{\mathcal{B}_j}$, this cannot depend on the choice by $\mathtt{p}$ and hence must also be possible in G_i which yields a contradiction. On the other hand, $\langle 0, \emptyset\rangle$ can also be the result of projecting empty loops (after some choice operation). Then, however, since $G_i \upharpoonright_{\mathtt{r}}$ is defined, all other branches also return $\langle 0, \emptyset\rangle$ for the standard projection to be defined. In turn, since $G_i \upharpoonright_{\mathtt{r}} \sqcap G_j \upharpoonright_{\mathtt{r}}$ is defined, $G_j \upharpoonright_{\mathtt{r}}$ is $\langle 0, \emptyset\rangle$ and hence the above projection cannot be undefined.

For the **second case**, we first observe that both $\sqcap$ and $\sqcap\!\!\!\!\sqcap$ agree on the type of events that can be merged, e.g. only send events can merge with send events. This is why we do not consider all the different combinations, e.g. of merging send events with recursion binders, but only the ones that are defined in both cases. Note that $EL_i \sqcap\!\!\!\!\sqcap EL_j$ is defined for cases where the rules of $\sqcap\!\!\!\!\sqcap$ mimic the ones of $\sqcap$ literally. (Recall that we do only need to check until the first recursion variable thanks to Claim I.) In this sense, both definitions $\sqcap\!\!\!\!\sqcap$ and $\sqcap$ agree for merging 0, recursion variables t, recursion binders μt and send events (internal choice $\oplus$). For receive events (external choice $\&$), they do not agree. The variant $\sqcap\!\!\!\!\sqcap$ requires that both sets I and J are the same. Since $\circledast_i = \circledast_j$, there needs to be some message exchange in one branch that cannot happen in the other branch. However, analogous to the previous case, the send event of such a receive event must not depend on the choice of $\mathtt{p}$ which branch to take. However, this is not possible with the blocked projection operator so $\sqcap$ would also be undefined in this case, yielding a contradiction. $\square$

Lemma 3.34 (Property of Projectable Global Types II – Receive)**.** Let G be some global type and G' be some syntactic subterm of G. For every subset of participants $\mathcal{B} \subseteq \mathcal{P}$, it holds that $\mathrm{msgs}^{\mathcal{B}}_{(G' \ldots)} = \mathrm{avail}(\mathcal{B}, \emptyset, G')$

Proof. As for the proof of Lemma 3.33, we construct a global type $\widehat{G'}$ that specifies the same protocol as G' in the context of G. Let T' be all recursion variables in G' that are not bound in G'. We define $\widehat{G'}$ to be the type that is obtained by substituting every $t \in T'$ by $get\mu_G(t)$. By construction, $\widehat{G'}$ is a global type (and not only a syntactic subterm of one).

Let T be the set of all recursion variables in $\widehat{G'}$. Since we unfolded the unbound recursion variables up-front in $\widehat{G'}$, it is straightforward, from the definition of $\mathrm{avail}(\text{-}, \text{-}, \text{-})$, that the following holds:

$$\mathrm{avail}(\mathcal{B}, \emptyset, G') = \mathrm{avail}(\mathcal{B}, T, \widehat{G'}) \enspace .$$

By construction of $\widehat{G'}$, the computation of $\mathrm{avail}(\mathcal{B}, T, \widehat{G'})$ will never apply the second case which is the only one where we do not simply descend in the structure of the third parameter so induction hypotheses will apply. By the equi-recursive view on global types, it holds that $\mathrm{msgs}^{\mathcal{B}}_{(G' \dots)} = \mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'} \dots)}$ and therefore it suffices to show that $\mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'} \dots)} = \mathrm{avail}(\mathcal{B}, T, \widehat{G'})$.

We do induction on the structure of $\widehat{G'}$ to prove the claim.

For the base case where $\widehat{G'} = 0$, the claim follows trivially.

For $\widehat{G'} = t$, we know that either $t \in T'$ has been substituted by its definition using $get\mu_G(\text{-})$ or it was bound in G'. Normally, one would need to substitute t again and consider the messages. However, in both cases, we know that we traversed all branches from the binder to the variable at least once and unfolding further does not change the set of available messages, which only consist the first possible message for every channel.

For the induction step, suppose that $\widehat{G'} = \mu t \,.\, \widehat{G''}$ first. By induction hypothesis, we know that the claim holds for $\widehat{G''}$ and the binder for recursion variable t does not add any message to the set of messages: $\mathrm{msgs}^{\mathcal{B}}_{(\mu t \,.\, \widehat{G''} \dots)} = \mathrm{msgs}^{\mathcal{B}}_{(\widehat{G''} \dots)}$. By definition, we have that $\mathrm{avail}(\mathcal{B}, T, \mu t \,.\, \widehat{G''}) = \mathrm{avail}(\mathcal{B}, T \cup \{t\}, \widehat{G''}) = \mathrm{avail}(\mathcal{B}, T, \widehat{G''})$. By induction hypothesis, it holds that $\mathrm{msgs}^{\mathcal{B}}_{(\widehat{G''} \dots)} = \mathrm{avail}(\mathcal{B}, T, \widehat{G''})$, which proves this case.

Last, let $\widehat{G'} = \Sigma_{i \in I}\, \mathtt{p} \to \mathtt{q}_i : \mathsf{m}_i \,.\, \widehat{G'_i}$. We do a case analysis if $\mathtt{p} \notin \mathcal{B}$.

Suppose that $\mathtt{p} \notin \mathcal{B}$. Then, all receives are added by definition of $\mathrm{msgs}^{\mathcal{B}}_{\text{-}}$:

$$\mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'} \dots)} = \{\mathtt{q} \triangleleft \mathtt{p}?\mathsf{m}_i \mid i \in I\} \cup \bigcup_{i \in I} \mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'_i} \dots)} \enspace .$$

For $\mathrm{avail}(\mathcal{B}, T, \widehat{G'})$, the fourth case in the definition applies:

$$\mathrm{avail}(\mathcal{B}, T, \widehat{G'}) = \{\mathtt{q} \triangleleft \mathtt{p}?\mathsf{m}_i \mid i \in I\} \cup \bigcup_{i \in I} \mathrm{avail}(\mathcal{B}, T, \widehat{G'_i}) \enspace .$$

The first part of the equations are identical and combining the induction hypotheses for each $\widehat{G'_i}$ proves the claim.

Suppose that $\mathtt{p} \in \mathcal{B}$. Then, by definition, it holds that $\mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'}\ldots)}$ does not contain any of the receipts $\mathtt{q}_i \triangleleft \mathtt{p}?_$. Additionally, $\mathtt{q}_i$ is added to $\mathcal{B}$ by definition: $\mathrm{msgs}^{\mathcal{B}}_{(\widehat{G'}\ldots)} = \bigcup_{i\in I} \mathrm{msgs}^{\mathcal{B}\cup\{\mathtt{q}_i\}}_{(\widehat{G'_i},\ldots)}$. For $\mathrm{avail}(\mathcal{B}, T, \widehat{G'})$, the fifth case applies:

$$\mathrm{avail}(\mathcal{B}, T, \widehat{G'}) = \bigcup_{i\in I} \mathrm{avail}(\mathcal{B} \cup \{\mathtt{q}_i\}, T, \widehat{G'_i}) \ .$$

Combining the induction hypotheses for each $\widehat{G'_i}$ proves the claim. □

Existence of Family of Run Mappings for Projectable Global Types

Equipped with these properties, we can prove the existence of run mappings.

Lemma 3.24 ($\{\mathsf{LAut}(G{\restriction_{\mathtt{p}}})\}_{\mathtt{p}\in\mathcal{P}}$ has a family of run mappings)**.** Let G be a projectable global type. Then, $\{\!\{\mathsf{LAut}(G{\restriction_{\mathtt{p}}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ has a family of run mappings for $\mathsf{GAut}(G)$.

Proof. Let w be a prefix of an execution of $\{\!\{\mathsf{LAut}(G{\restriction_{\mathtt{p}}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$. We prove the claim by induction on the length of w. The base case where $w = \varepsilon$ is trivial: $\rho = \varepsilon$ and $\rho(\mathtt{p}) = \varepsilon$ for all $\mathtt{p}$.

For the induction step, we append x to obtain wx for which $x = \varepsilon$, $x = \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m}$ or $x = \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$ for some $\mathtt{p}, \mathtt{q}$ and m. As induction hypothesis, we assume that there is some run ρ in $\mathsf{GAut}(G)$ such that $w' \preceq_{\sim} \mathrm{trace}(\rho)$ and a run mapping $\rho(\text{-})$ such that $w{\Downarrow_{\Gamma_{\mathtt{p}}}} = \mathrm{trace}(\rho(\mathtt{p})){\Downarrow_{\Gamma_{\mathtt{p}}}}$ for every $\mathtt{p}$.

For all cases, we re-use the witness run ρ by extending it to ρ' when necessary and re-use the run mapping $\rho(\mathtt{p})$ except for at most one participant.

First, Suppose that $x = \varepsilon$. Then, the claim follows with the same run ρ and mapping $\rho(\text{-})$. It is straightforward that $w{\Downarrow_{\Gamma_{\mathtt{p}}}} = (w\varepsilon){\Downarrow_{\Gamma_{\mathtt{p}}}}$ for every $\mathtt{p}$ holds. Let $\mathtt{p}$ be the participant which takes an ε-transition and hence leads to the extension by ε. By the definition of semantics for local types (Definition 3.3), we know that a state can only have a single ε-transition and hence the trace $w\varepsilon$ can be extended in the same way as w is extended to w' to obtain w'' for $w\varepsilon$. Since w' and w'' only differ by one ε, $w'' \sim w' \sim \mathrm{trace}(\rho)$ and the claim follows by transitivity of $\sim$.

Second, suppose that $x = \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m}$.
Claim I: It holds that $\rho(\mathtt{p}) \leq \rho(\mathtt{q})$.

Proof of Claim I. We know $\mathrm{trace}(\rho(\mathtt{p})){\Downarrow_{\Gamma_{\mathtt{p}}}} = w{\Downarrow_{\Gamma_{\mathtt{p}}}}$ and $\mathrm{trace}(\rho(\mathtt{q})){\Downarrow_{\Gamma_{\mathtt{q}}}} = w{\Downarrow_{\Gamma_{\mathtt{q}}}}$. By definition of $\mathsf{GAut}(G)$, we know that $\mathtt{p}$'s and $\mathtt{q}$'s common actions always happen in pairs of sending and receiving a message. Since there is no out-of-order execution, we know that $\mathtt{p}$ cannot have received m from $\mathtt{q}$ yet. Hence, $\rho(\mathtt{p}) \leq \rho(\mathtt{q})$.

End Proof of Claim I.
For every $\mathtt{r} \neq \mathtt{p}$, we define $\rho'(\mathtt{r}) = \rho(\mathtt{r})$. We know that $(wx){\Downarrow_{\Gamma_{\mathtt{r}}}} = w{\Downarrow_{\Gamma_{\mathtt{r}}}}$ for every $\mathtt{r} \neq \mathtt{p}$ so the conditions on the run mapping is satisfied for all $\mathtt{r} \neq \mathtt{p}$.

For p, we extend $\rho(\mathsf{p})$ to be longest, i.e. extending the run further would render one of the conditions for run mappings unsatisfied, to obtain a set of possible extended runs. For every run, either the last or second-last state of these runs corresponds to some syntactic subterm of G. One of them is a prefix of the sender's run $\rho(\mathsf{q})$. In case of empty loop branches, the set of runs might be infinite, however, only one is a prefix of the sender's run $\rho(\mathsf{q})$. We denote its syntactic subterm by G' and we will show that only the option taken by q can be pursued.

Since p can receive m from q, we know that the current local type L of p is the result of (merging) the projection of some type(s) of shape $\Sigma_\ _\!\rightarrow\!_ : _ . _$ and all these syntactic subterms are of this form. Thus, the local type L must be the result of some merge applying the receiving rule last to compute

$$\langle L, _ \rangle = \langle L_1, Msg_1 \rangle \sqcap \ldots \sqcap \langle L_n, Msg_n \rangle$$

for some n and $L_l = \&_{i \in I_l} \mathsf{q}_l ? \mathsf{m}_i . AL_{l,i}$ for every $l \in \{1, \ldots, n\}$ and for some $l \in \{1, \ldots, n\}$, it holds that $\langle L_l, Msg_l \rangle = G' {\upharpoonright}_{\mathsf{p}}$. The other local types correspond to the branches p explores concurrently while processing $w {\Downarrow}_{\Gamma_{\mathsf{p}}}$. We unrolled the binary definition of merge but since $\sqcap$ is associative and commutative, we know that $\langle L_l, Msg_l \rangle \sqcap \langle L_{l'}, Msg_{l'} \rangle$ is defined for all $l, l' \in \{1, \ldots, n\}$. Therefore it holds, for any $l, l' \in \{1, \ldots, n\}$, that

$$\forall i \in I_l \setminus I_{l'} .\, \mathsf{p} \triangleleft \mathsf{q}_l ? \mathsf{m}_i \notin Msg_{l'}, \text{ and}$$
$$\forall i \in I_{l'} \setminus I_l .\, \mathsf{p} \triangleleft \mathsf{q}_{l'} ? \mathsf{m}_i \notin Msg_l .$$

Without loss of generality, let $k \in \bigcup_{1 \leq l \leq n} I_l$ be the index for which $\mathsf{q} = \mathsf{q}_k$ and $\mathsf{m} = \mathsf{m}_k$. Therefore, for any l with $k \notin I_l$, it holds that $\mathsf{p} \triangleleft \mathsf{q}_k ? \mathsf{m}_k \notin Msg_l$.

Let l be some index. To obtain every individual availability annotated local type $\langle L_l, Msg_l \rangle$, the following case of the projection has been applied:

$$\langle L_l, Msg_l \rangle = \langle \mathop{\&}_{j \in J_l} \mathsf{q}_l ? \mathsf{m}_j . (G_j {\upharpoonright}_{\mathsf{p}}), \bigcup_{j \in J} \operatorname{avail}(\{\mathsf{p}\}, \emptyset, G_j) \rangle$$

for some index sets J_l and syntactic subterms G_j.
From Lemma 3.34, we have that, for every $l \in \{1, \ldots, n\}$, $j \in I_l$ and $\mathcal{B} \subseteq \mathcal{P}$, it holds that $\operatorname{msgs}^{\mathcal{B}}_{(G_j \ldots)} = \operatorname{avail}(\mathcal{B}, \emptyset, G_j)$ (*). Recall that $\operatorname{msgs}^{\mathcal{B}}_{(G_j \ldots)}$ is defined as all messages that can be sent in any run starting with branch G_j when no participant in $\mathcal{B}$ can take any further step.

By definition of $\sqcap$, all the first actions $\mathsf{q}_k ? \mathsf{m}_k$ will be merged together to one branch in the local type L so p will continue with this branch of L. Let us denote this branch by B_k for now. By instantiating (*) with $\mathcal{B} = \{\mathsf{p}\}$ whose next action is to receive from q, we know that the message m_k by participant q_k cannot occur in the channel (q, p) in any branch of L different from B_k so p cannot diverge from the run $\rho(\mathsf{q})$ by receiving

m_k out of order. Hence, p follows the run of q. By assumption p can receive from q, so q must have taken one of the runs that corresponds to this branch of L so we can choose $\rho'(\mathtt{p})$ such that $\rho'(\mathtt{p}) \leq \rho(\mathtt{q})$ which proves the claim.

Third, suppose that $x = \mathtt{p} \triangleright \mathtt{q}!\mathsf{m}$.

We collect the participants with longer runs than p:

$$\mathcal{S} := \{\mathtt{r} \mid \rho(\mathtt{p}) \leq \rho(\mathtt{r}) \land \rho(\mathtt{p}) \neq \rho(\mathtt{r})\} \ .$$

It is obvious that $\mathtt{p} \notin \mathcal{S}$. We assume $\mathcal{S}$ is not empty as the claim follows trivially if it is: we could simply extend ρ, keep the mapping for all other participants and extend it for p accordingly.

As before, for p, we extend $\rho(\mathtt{p})$ to be longest, i.e. extending the run further would render one of the conditions unsatisfied, to obtain a set of possible runs. In the presence of empty loop branches, this set could be infinite. Here, it suffices to obtain some valid run mapping though. For every run in this set, either the last or second-last state of these runs corresponds to some syntactic subterm of G. Since p can send m to q, we know that the current local type L of p is the result of (merging) the projection of some type(s) of shape $\Sigma_ \to _ : _._$ and all these syntactic subterms are of this form. The local type is also the result of merging of this shape since p sends at this step:

$$L = \bigsqcap_{i \in I} \big(\bigoplus_{j \in J} \mathtt{q}_j!\mathsf{m}_j.(G_{(i,j)}\upharpoonright_{\mathtt{p}}) \big) = \bigoplus_{j \in J} \mathtt{q}_j!\mathsf{m}_j. \bigsqcap_{i \in I} (G_{(i,j)}\upharpoonright_{\mathtt{p}})$$

for some index sets I and J, $\mathtt{q}_j$ as well as m_j and $G_{(i,j)}$ for every $j \in J$ and $i \in I$.

So the merge $\sqcap$ ensures that p has the same options to send after processing $w\Downarrow_{\Gamma_{\mathtt{p}}}$, no matter which run in $\mathsf{GAut}(G)$ was pursued. This ensures that we are able to adapt the mapping from the induction hypothesis such that $\mathrm{trace}(\rho'(\mathtt{p})) = (wx)\Downarrow_{\Gamma_{\mathtt{p}}}$.

Inspired by prophecy variables [1], we assume that each participant r in $\mathcal{S}$ followed the run that p will choose to take, i.e. $\rho'(\mathtt{p}) \leq \rho(\mathtt{r})$. Assuming this, we can re-use the same mapping for them: $\rho'(\mathtt{r}) = \rho(\mathtt{r})$ and the overall claim follows.

It remains to show that the mappings $\rho(\mathtt{r})$ for $\mathtt{r} \in \mathcal{S}$ can be chosen in such a way without prohibiting any of the participants to proceed with some events. We will do so by applying Lemma 3.33.

Participant p might be pursuing more than one run in $\mathsf{GAut}(G)$. Since all first send options for p could be merged, we know that they are the same over all possible continuations. Therefore, we show that no matter which option p chooses, no participant $\mathtt{r} \in \mathcal{S}$ can have processed an action that is incompatible with this branch yet, provided that neither the sender p nor each receiver $\mathtt{q}_i$ has processed some event in the continuations of the branch of option i.

For this, we show for all $\mathtt{r} \in \mathcal{S}, i \in I$ and $j_1, j_2 \in J$ that

$$L^{\{\mathtt{p},\mathtt{q}_i\}}_{(G_{(i,j_1)}\dots)}\Downarrow_{\Gamma_{\mathtt{r}}} = L^{\{\mathtt{p},\mathtt{q}_j\}}_{(G_{(i,j_2)}\dots)}\Downarrow_{\Gamma_{\mathtt{r}}}.$$

Since we do only compare runs that start at the same state, i.e. the syntactic subterm $\oplus_{j\in J}\, \mathsf{q}_j ! \mathsf{m}_j . G_{(i,j)}$, this claim follows from Lemma 3.33. It suffices to compare those since p can not and does not dictate which of the branches of I were taken, but the ones in J. It might happen that some choice with regard to the options in I has already been committed but p does not know about this at this stage yet. Still, all participants in $\mathcal{S}$ have the same options along all the options in J and therefore the use of prophecy variables does not prohibit them from proceeding with any action that is possible in the execution.

This concludes the induction and, thus, the proof. □

3.3.5 From Run Mappings via Control Flow Agreement to Implementability

To show protocol fidelity and deadlock freedom, we use a strengthening of the run mappings by a progress condition. Any trace can be extended to match all actions in the run given by the run mapping.

Definition 3.35 (Control Flow Agreement)**.** Let G be a global type. A CSM $\{\!\{A_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ satisfies *Control Flow Agreement* (CFA) for $\mathsf{GAut}(G)$ iff for every finite trace w of $\{\!\{A_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$, there is a run ρ in $\mathsf{GAut}(G)$ such that the following holds:

- $w{\Downarrow_{\Gamma_\mathsf{p}}}$ is a prefix of $\mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathsf{p}}}$ for every participant p, and
- w can be extended (in $\{\!\{A_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$) to w' such that $w' \sim \mathrm{trace}(\rho)$.

Run mappings do only show that nothing did go wrong yet because there is a global run all participants could have followed up to some point. With the following lemma, we show that all participants can catch up to the end of this global run, a first step towards control flow agreement.

Lemma 3.36 (Runs following a common path are extendable)**.** Let G be a projectable global type, w a trace of $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$, and ρ a finite run in $\mathsf{GAut}(G)$ such that $w{\Downarrow_{\Gamma_\mathsf{p}}} \leq \mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathsf{p}}}$ for every $\mathsf{p}\in\mathcal{P}$. Then, there is a trace w' of $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ such that $w \leq w'$ and $w' \sim \mathrm{trace}(\rho)$.

Proof. By definition, it holds that $\mathrm{trace}(\rho) \in \mathrm{pref}(\mathcal{L}(G))$. (The run ρ might not be maximal and we, thus, use $\mathrm{pref}(\text{-})$.) From Lemma 3.21, we know that every prefix of $\mathcal{L}(G)$ is also the trace of some run in $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$. Thus, there is a run for $\mathrm{trace}(\rho)$ in $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$. With Lemma 2.10, which shows that languages of CSMs are closed under $\sim$, it follows that every trace in $\mathcal{C}^{\sim}(\{\mathrm{trace}(\rho)\})$ has a run in $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$.

We prove the claim by finding w' such that $w \leq w'$ and $w' \sim \mathrm{trace}(\rho)$.

By Lemma 2.9, it suffices to find a FIFO-compliant trace w' such that $w'{\Downarrow_{\Gamma_\mathsf{p}}} = \mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathsf{p}}}$ for every participant p.

We know that, for every $\mathtt{p}$, there is $y_{\mathtt{p}}$ such that $(w\Downarrow_{\Gamma_{\mathtt{p}}}) \cdot y_{\mathtt{p}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$. We show that there is a FIFO-compliant $w' = wy$ such that $y\Downarrow_{\Gamma_{\mathtt{p}}} = y_{\mathtt{p}}$ for every $\mathtt{p}$. Since w is a trace of a CSM, w is FIFO-compliant (Lemma 2.5). We need to construct y and start with $y = \varepsilon$. We extend y by consuming prefixes of $y_{\mathtt{p}}$ and do so as long as some $y_{\mathtt{p}}$ is not empty. At all times, we preserve the following invariant for every $\mathtt{p}$: $(wy)\Downarrow_{\Gamma_{\mathtt{p}}} \cdot y_{\mathtt{p}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$.

If there is any $y_{\mathtt{p}} = (\mathtt{p}\triangleright\mathtt{q}!\mathsf{m}) \cdot y'_{\mathtt{p}}$, we extend $y = y \cdot (\mathtt{p}\triangleright\mathtt{q}!\mathsf{m})$ and set $y_{\mathtt{p}} = y'_{\mathtt{p}}$. This preserves the invariant and appending events of shape $_\triangleright_!_$ to y does preserve FIFO-compliancy of wy. If there is no such $y_{\mathtt{p}}$ (and not all $y_{\mathtt{p}}$ are empty), there is a set of participants $\mathcal{S}$ such that $y_{\mathtt{p}} = (\mathtt{p}\triangleleft_?_) \cdot y'_{\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{S}$. Intuitively, we define the shortest prefix ρ' of ρ such that there is some participant whose trace is included in the projection of $\operatorname{trace}(\rho')$:

$$\rho' := \min_{\leq}\{\max_{\leq}\{\rho' \mid \exists \mathtt{p} \in \mathcal{S}.\, (wy)\Downarrow_{\Gamma_{\mathtt{p}}} = \operatorname{trace}(\rho')\Downarrow_{\Gamma_{\mathtt{p}}} \text{ for } \rho' \leq \rho\}\}.$$

We are interested in the next non-ε transition label of the run ρ' and, thus, use $\max_{\leq}$ to have such a label, making it technically not the shortest run prefix. Let $\mathtt{p} \in \mathcal{S}$ such that $(wy)\Downarrow_{\Gamma_{\mathtt{p}}} = \operatorname{trace}(\rho')\Downarrow_{\Gamma_{\mathtt{p}}}$. Let $\rho' = \rho'' \xrightarrow{q}\xrightarrow{x} q'$. By construction, it holds that $x \in \Gamma_{\mathtt{p}}$. Therefore, the choice of $\mathtt{p}$ is unique and for all other participants $\mathtt{q} \neq \mathtt{p}$, it holds that $(wy)\Downarrow_{\Gamma_{\mathtt{q}}} \neq \operatorname{trace}(\rho')\Downarrow_{\Gamma_{\mathtt{q}}}$. By assumption, it holds that $(wy)\Downarrow_{\Gamma_{\mathtt{q}}} \leq \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{q}}}$ and hence $\operatorname{trace}(\rho')\Downarrow_{\Gamma_{\mathtt{q}}} \leq (wy)\Downarrow_{\Gamma_{\mathtt{q}}}$ because $\rho' \leq \rho$ for all $\mathtt{q}$ with $\mathtt{q} \neq \mathtt{p}$ (*).

Let $y_{\mathtt{p}} = (\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m}) \cdot y'_{\mathtt{p}}$ for some $\mathtt{q}$ and m. We append the first event $\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m}$ from $y_{\mathtt{p}}$ to y, i.e. $y := y \cdot (\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m})$ and $y_{\mathtt{p}} = y'_{\mathtt{p}}$. It is straightforward that the invariant is preserved. For FIFO-compliancy of wy, we need to show that

$$\mathcal{V}((wy)\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_}) \leq \mathcal{V}((wy)\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}) \ .$$

From the invariant, we know that $(wy)\Downarrow_{\Gamma_{\mathtt{p}}} \cdot y_{\mathtt{p}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$ and $(wy)\Downarrow_{\Gamma_{\mathtt{q}}} \cdot y_{\mathtt{q}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{q}}}$, so it holds that $(w \cdot y \cdot y_{\mathtt{p}})\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_} = \operatorname{trace}(\rho)\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}$ and $(w \cdot y \cdot y_{\mathtt{q}})\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_} = \operatorname{trace}(\rho)\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_}$. Global types specify FIFO-compliant words (Proposition 2.25), which is a property that is preserved by $\operatorname{pref}(\text{-})$. Thus, $\operatorname{trace}(\rho)$ is FIFO-compliant, yielding $\mathcal{V}(\operatorname{trace}(\rho)\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}) = \mathcal{V}(\operatorname{trace}(\rho)\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_})$. This ensures that the next message in $(\mathtt{q}, \mathtt{p})$ is m if it exists. By definition of Γ and $\mathsf{GAut}(G)$, the send and receive event of $\mathtt{r}\to\mathtt{s}:\mathsf{m}$ always occur in pairs on the run ρ. From (*), we know that $\mathtt{q}\triangleright\mathtt{p}!\mathsf{m}$ must be the next action of shape $\mathtt{q}\triangleright\mathtt{p}!_$ with unmatched receive in wy. Channels preserve FIFO order so the next message in channel $(\mathtt{q}, \mathtt{p})$ is m.

Overall, this procedure ensures that we can always extend wy by appending prefixes of $y_{\mathtt{p}}$ in a way that preserves FIFO-compliancy of wy and $(wy)\Downarrow_{\Gamma_{\mathtt{p}}} \cdot y_{\mathtt{p}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$ for every $\mathtt{p}$. Note that ρ is finite and hence each $y_{\mathtt{p}}$ is finite so the procedure terminates. For w', we can choose the result of the procedure wy and it holds, for every $\mathtt{p}$, that $(wy)\Downarrow_{\Gamma_{\mathtt{p}}} = \operatorname{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$ which proves the claim. □

We use the previous result to show that the existence of a family of run mappings entails control flow agreement for the CSM of local types.

Lemma 3.37 (Run mappings entail CFA)**.** Let G be a projectable global type, If $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ has a family of run mappings for $\mathsf{GAut}(G)$, then $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ satisfies CFA for $\mathsf{GAut}(G)$.

Proof. Let w be a trace of $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with witness run ρ and its run mapping $\rho(\text{-})$. We can assume that ρ is finite since w is a finite trace. We know that $\mathrm{trace}(\rho(\mathtt{p}))\Downarrow_{\Gamma_{\mathtt{p}}} = w\Downarrow_{\Gamma_{\mathtt{p}}}$ and $\rho(\mathtt{p}) \leq \rho$ for every $\mathtt{p}$. Hence, for every $\mathtt{p}$, it holds that $w\Downarrow_{\Gamma_{\mathtt{p}}} \leq \mathrm{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$, which is exactly the first condition of CFA.

The second condition requires that we can extend w to w' in $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that $w' \sim \mathrm{trace}(\rho)$. Since ρ is finite, we can apply Lemma 3.36 and obtain an extension w' of w in $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that $w' \sim \mathrm{trace}(\rho)$. □

We want to use control flow agreement to show that the CSM of local projections does not introduce new behaviour. In addition, we use a previous result and show both ingredients for implementability: first, protocol fidelity and second, deadlock freedom.

Lemma 3.38 (CFA for $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ entails protocol fidelity)**.** Let G be a projectable global type. If the CSM $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ satisfies CFA for $\mathsf{GAut}(G)$, then $\mathcal{C}^{\sim}(\mathcal{L}(G)) = \mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

Proof. We prove the claim by two inclusions.

First, Lemma 3.21 yields that $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$. The language of a CSM is closed under $\sim$ (Lemma 2.10), hence the first inclusion holds.

It remains to prove the second inclusion: $\mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(G))$.

Let $w \in \mathcal{L}(\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}})$. For both the finite and infinite case, we will use Definition 3.35 to obtain a maximal run ρ in $\mathsf{GAut}(G)$ for which $w\Downarrow_{\Gamma_{\mathtt{p}}} \leq \mathrm{trace}(\rho)\Downarrow_{\Gamma_{\mathtt{p}}}$ for every $\mathtt{p}$.

We do a case split whether w is finite or infinite.

First, suppose that w is finite. We use Definition 3.35 to find the corresponding run ρ in $\mathsf{GAut}(G)$, show that it is maximal in $\mathsf{GAut}(G)$ and that we do not need to extend w for any participant to reach the end of ρ.

We show that ρ ends in a final state q and is hence maximal in $\mathsf{GAut}(G)$. Let $(\vec{q}, \xi)$ be the final configuration of $\{\!\{\mathsf{LAut}(G\restriction_{\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ that has been reached with trace w. Since $(\vec{q}, \xi)$ is final, every $\vec{q}_{\mathtt{p}}$ is final in $\mathsf{LAut}(G\restriction_{\mathtt{p}})$. Then, we know that $\vec{q}_{\mathtt{p}}$ corresponds to 0 by definition of $\mathsf{LAut}(G\restriction_{\mathtt{p}})$. By construction of the FSM for the semantics of local types (Definition 3.3), $\vec{q}_{\mathtt{p}}$ does not have any outgoing transitions. Proposition 3.20 states that every run in $\mathsf{GAut}(G)\Downarrow_{\Gamma_{\mathtt{p}}}$ is also possible in $\mathsf{LAut}(G\restriction_{\mathtt{p}})$. Therefore, by contradiction, it follows that a run in $\mathsf{GAut}(G)\Downarrow_{\Gamma_{\mathtt{p}}}$ with trace $w\Downarrow_{\Gamma_{\mathtt{p}}}$ cannot be extended to a run with a longer trace for all $\mathtt{p}$. Hence, the run of $\mathrm{trace}(\rho)$ in $\mathsf{GAut}(G)$ cannot be extended to a run with a longer trace, i.e. there can only be ε-transitions. By definition of $\mathsf{GAut}(G)$, ε-transitions occur from states corresponding to t and $\mu t \,.\, G'$. We claim that q does

not correspond to any t or syntactic subterm of shape $\mu t \,.\, G'$. Towards a contradiction, assume that q corresponds to some t or subterm of shape $\mu t \,.\, G'$. Global types have guarded recursion, i.e. G' cannot be 0 for any $\mu t \,.\, G'$. Then there is some non-ε-transition from q, which is a contradiction.

Hence, there is also no ε-transition from q and q does not have any outgoing transitions. By construction of the semantics of global types (Definition 2.24), q corresponds to 0, is final in $\mathsf{GAut}(G)$, and ρ is maximal in $\mathsf{GAut}(G)$.

It remains to show that we do not need to extend w to w' for any participant – in other words, that every participant followed the run ρ all the way to the end. We define $\mathcal{S} := \{\mathtt{p} \mid w\Downarrow_{\Gamma_\mathtt{p}} \neq \mathrm{trace}(\rho)\Downarrow_{\Gamma_\mathtt{p}}\}$. We claim that $\mathcal{S} = \emptyset$. Towards a contradiction, let us assume that $w\Downarrow_{\Gamma_\mathtt{p}} \neq \mathrm{trace}(\rho)\Downarrow_{\Gamma_\mathtt{p}}$ for $\mathtt{p} \in \mathcal{S}$. Since we reached the final state q with ρ in $\mathsf{GAut}(G)$ which corresponds to 0, there is no successor in $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for $\vec{q}_\mathtt{p}$ either and there is no possibility to extend w for any $\mathtt{p}$ but this contradicts the second condition of CFA and we do not need to extend w' for any participant.

So it holds that $w \sim \mathrm{trace}(\rho)$. Because ρ is maximal, we have that $\mathrm{trace}(\rho) \in \mathcal{L}(G)$ and the claim (for finite w) follows since $\mathcal{L}(G)$ is closed under $\sim$ by definition.

Second, suppose that w is infinite. We show that there is an infinite run in $\mathsf{GAut}(G)$ that matches with w when closing under $\sim$.

From Definition 3.35, we can obtain a finite run ρ for every prefix u of w such that the conditions for CFA hold, i.e. there is an extension u' of u with $u' \sim \mathrm{trace}(\rho)$. To simplify the argument, we use an idea similar to prophecy variables [1] as in the proof of Lemma 3.24. Here, we can use a similar oracle that tells which witness run as well as run mapping to use for every prefix of w. This does not restrict the participants in any way: all conditions of CFA hold for the prefix and this witness run since they hold for longer prefixes of w. By this, we ensure that the run ρ can always be extended for longer prefixes of w $(*)$.

We prove the existence of an infinite run similarly to Lemma 3.21. Consider a tree $\mathcal{T}$ where each node corresponds to a run ρ on some finite prefix $w' \leq w$. The root is labelled by the empty run. The children of a node ρ are runs that extend ρ by a single transition – these exist by $(*)$. Since our CSM, derived from a global type, is a finite number of finite state machines, $\mathcal{T}$ is finitely branching. By König's Lemma, there is an infinite path in $\mathcal{T}$ that corresponds to an infinite run ρ in $\mathsf{GAut}(G)$ for which $w \preceq^\omega_\sim \mathrm{trace}(\rho)$. By definition of $\mathcal{C}^\sim(\text{-})$ for infinite traces, this yields $w \in \mathcal{C}^\sim(\mathcal{L}(G))$. □

We prove the same property for deadlock freedom, the second ingredient to implementability.

Lemma 3.39 (CFA for $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ entails deadlock freedom)**.** Let G be a projectable global type. If the CSM $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ satisfies CFA for $\mathsf{GAut}(G)$, then $\{\!\{\mathsf{LAut}(G{\upharpoonright_\mathtt{p}})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free.

Proof. Towards a contradiction, assume that $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ has a deadlock. Then, there is a trace w for which $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ reaches a non-final configuration $(\vec{q}, \xi)$ and no transition is possible. Let ρ be the maximal run from Definition 3.35 for which the conditions of CFA hold. Since $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ cannot make another step, the second condition can only apply for $w' = w$, so all events along ρ have been processed by all participants, i.e. $w\Downarrow_{\Gamma_{\mathrm{p}}} = \mathrm{trace}(\rho)\Downarrow_{\Gamma_{\mathrm{p}}}$ for every p, by Lemma 2.9.

Claim I: We claim the last state q of ρ corresponds to 0.

Proof of Claim I. Towards a contradiction, assume that ρ can be extended, i.e. q has a successor. Since ρ is maximal, there cannot be any ε-transitions from q but only some transition labelled different from ε. Since all participants have processed all events along ρ and send and receive events are always jointly specified, such a transition can only be labelled by $\mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$ for some p, q, and m. Hence, there is a run for $(w\Downarrow_{\Gamma_{\mathrm{p}}}) \cdot \mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$ in $\mathsf{GAut}(G)\Downarrow_{\Gamma_{\mathrm{p}}}$. By Proposition 3.20, there is a run for $(w\Downarrow_{\Gamma_{\mathrm{p}}}) \cdot \mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$ in $\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})$. Participant p can hence take another transition which contradicts the assumption that $(\vec{q}, \xi)$ is a deadlock.

End Proof of Claim I.

So we know that the last state q of ρ corresponds to 0. Again, by Proposition 3.20, there is no transition for any p from $\vec{q}_{\mathrm{p}}$ in $\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})$ because $\mathsf{GAut}(G)\Downarrow_{\Gamma_{\mathrm{p}}}$ has no transition for any p. All channels in ξ are empty and $w\Downarrow_{\Gamma_{\mathrm{p}}} = \mathrm{trace}(\rho)\Downarrow_{\Gamma_{\mathrm{p}}}$ for every p. By construction of the FSM for the semantics of global types (Definition 2.24), a state without outgoing transitions corresponds to 0 and is thus final. But then, all states in $\vec{q}_{\mathrm{p}}$ are final (and all channels in ξ are empty) so $(\vec{q}, \xi)$ is a final configuration, which yields the desired contradiction. □

3.3.6 Wrapping Up: Projectable Global Types are Implementable

Now, we can prove our main result.

Theorem 3.19. If a global type G is projectable, then $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ implements G.

Proof of Theorem 3.19. Let $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ be the CSM of local types. Then, we know that $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ has a family of run mappings for $\mathsf{GAut}(G)$ by Lemma 3.24. This, in turn, entails that $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ satisfies control-flow agreement for $\mathsf{GAut}(G)$ by Lemma 3.37. This satisfies the condition for Lemmas 3.38 and 3.39, giving protocol fidelity and deadlock freedom. □

In addition, we know that feasible eventual reception can be achieved. With control flow agreement, there is a run in the global type's state machine that all participants' views agree with. We argued before that all participants can catch up to the participant that progressed furthest in this run and then, by definition of global types, all channels are empty.

Corollary 3.40 (Feasible eventual reception)**.** The CSM $\{\!\{\mathsf{LAut}(G\!\upharpoonright_{\mathrm{p}})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ for a global type G satisfies feasible eventual reception if it is defined.

3.4 Incompleteness of Classical Projection Approaches

In the previous section, we presented our generalised projection operator, which is based on the classical MST projection approach. It can handle more communication patterns than previous projection operators. It is still not complete, though: it rejects implementable protocols. In this section, we exemplify such shortcomings of the classical projection approach. For this, we present two implementable variations of the two buyer protocol (cf. Sec. 3.1.1) and an example where a participant learns about a choice through the parity of number of messages received.

Example 3.41. We obtain an implementable variant by omitting both message interactions $\mathsf{a} \to \mathsf{s} : \mathsf{no}$ with which buyer a notifies seller s that they will not buy the item:

$$\mu t\,.\, + \begin{cases} \mathsf{a}\to\mathsf{s}:\mathsf{query}\,.\,\mathsf{s}\to\mathsf{a}:\mathsf{price}\,.\,(\mathsf{a}\to\mathsf{b}:\mathsf{split}\,.\,(\mathsf{b}\to\mathsf{a}:\mathsf{yes}\,.\,\mathsf{a}\to\mathsf{s}:\mathsf{buy}\,.\,t + \mathsf{b}\to\mathsf{a}:\mathsf{no}\,.\,t) + \mathsf{a}\to\mathsf{b}:\mathsf{cancel}\,.\,t) \\ \mathsf{a}\to\mathsf{s}:\mathsf{done}\,.\,\mathsf{a}\to\mathsf{b}:\mathsf{done}\,.\,0 \end{cases} .$$

This global type cannot be projected onto seller s. The merge operator would need to merge a recursion variable with an external choice. Visually, the classical merge operator does not allow to unfold the variable t and try to merge again. However, there is a local type for seller s:

$$\mu t_1\,.\, \& \begin{cases} \mathsf{a}?\mathsf{query}\,.\,\mu t_2\,.\,\mathsf{a}!\mathsf{price}\,.\,(\mathsf{a}?\mathsf{buy}\,.\,t_1 \,\&\, \mathsf{a}?\mathsf{query}\,.\,t_2 \,\&\, \mathsf{a}?\mathsf{done}\,.\,0) \\ \mathsf{a}?\mathsf{done}\,.\,0 \end{cases} .$$

The local type has two recursion variable binders while the global type only has one. Classical projection operators can never yield such a structural change: the merge operator can only merge states but not introduce new ones or introduce new backward transitions. ◀

Example 3.42 (Two Buyer Protocol with Subscription)**.** In this variant, buyer a first decides whether to subscribe to a yearly discount offer or not — before purchasing the sequence of items — and notifies buyer b if it does so:

$$G_{\mathrm{2BPWS}} := +\begin{cases} \mathsf{a}\to\mathsf{s}:\mathsf{login}\,.\,G_{\mathrm{2BP}} \\ \mathsf{a}\to\mathsf{s}:\mathsf{subscribe}\,.\,\mathsf{a}\to\mathsf{b}:\mathsf{subscribed}\,.\,G_{\mathrm{2BP}} \end{cases} .$$

The merge operator needs to merge a recursion variable binder μt with the external choice $\mathsf{b}\triangleleft\mathsf{a}?\mathsf{subscribed}$. Still, there is a local type L_{b} for b such that $\mathcal{L}(L_{\mathsf{b}}) = \mathcal{L}(G_{\mathrm{2BPWS}})\Downarrow_{\Gamma_{\mathsf{b}}}$:

$$L_{\mathsf{b}} := \& \begin{cases} \mathsf{a}?\mathsf{split}\,.\,(\mathsf{a}!\mathsf{yes}\,.\,L(t_1) \oplus \mathsf{a}!\mathsf{no}\,.\,L(t_2)) \\ \mathsf{a}?\mathsf{cancel}\,.\,L(t_3) \\ \mathsf{a}?\mathsf{done}\,.\,0 \\ \mathsf{a}?\mathsf{subscribed}\,.\,L(t_4) \end{cases} \quad \text{where} \quad L(t) := \mu t\,.\, \& \begin{cases} \mathsf{a}?\mathsf{split}\,.\,(\mathsf{a}!\mathsf{yes}\,.\,t \oplus \mathsf{a}!\mathsf{no}\,.\,t) \\ \mathsf{a}?\mathsf{cancel}\,.\,t \\ \mathsf{a}?\mathsf{done}\,.\,0 \end{cases} .$$

In fact, one can also rely on the fact that buyer a will comply with the intended protocol. Then, it suffices to introduce one recursion variable t in the beginning and substitute every $L(\text{-})$ with t, yielding a local type $L'_{\mathtt{b}}$ with $\mathcal{L}(L_{\mathtt{b}}) \subseteq \mathcal{L}(L'_{\mathtt{b}})$. ◀

Both examples show how brittle the classical projection approach is. In fact, one could unfold recursion variables, obtaining an equivalent global type that is then projectable. However, this is quite cumbersome and it will still be the case that the projected local types are structurally similar to the (then new) global type. Such tricks can only mitigate syntactic issues and can easily yield non-effective procedures that might not terminate. For instance, this treatment will most likely not terminate for global types where choices can be disambiguated with semantic properties, e.g. counting modulo a constant, like the following example.

Example 3.43 (Odd-even)**.** Consider the following global type G_{oe}:

$$+\begin{cases} \mathtt{p}\to\mathtt{q}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,\mu t_1\,.\,(\mathtt{p}\to\mathtt{q}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,t_1 \;+\; \mathtt{p}\to\mathtt{q}:\mathtt{b}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{b}\,.\,\mathtt{r}\to\mathtt{p}:\mathtt{o}\,.\,0) \\ \mathtt{p}\to\mathtt{q}:\mathtt{m}\,.\,\mu t_2\,.\,(\mathtt{p}\to\mathtt{q}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,t_2 \;+\; \mathtt{p}\to\mathtt{q}:\mathtt{b}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{b}\,.\,\mathtt{r}\to\mathtt{p}:\mathtt{m}\,.\,0) \end{cases} .$$

Figure 3.11 visualises G_{oe} as an HMSC. The left and right subprotocols respectively correspond to the top and bottom branches of the protocol. Participant p chooses a branch by sending either o or m to q. On the left, q echoes this message to r. Both branches continue in the same way: p sends an arbitrary number of o messages to q, each of which is forwarded twice from q to r. Participant p signals the end of the loop by sending b to q, which q forwards to r. Finally, depending on the branch, r must send o or m to p.

Figs. 3.12a and 3.12b depict the structural similarity between the global type G_{oe} and the implementations for p and q. For the "choicemaker" participant p, the reason is evident. Participant q's implementation collapses the continuations of both branches in the protocol into a single subcomponent. For r (Fig. 3.12c), the situation is more complicated. Participant r does not decide on or learn directly which branch is taken, but can deduce it from the parity of the number of o messages received from q: odd means left and even means right. The resulting local implementation features transitions going back and forth between the two branches that do not exist in the global type. Classical projection operators fail to create such transitions and unfolding recursion will not help. ◀

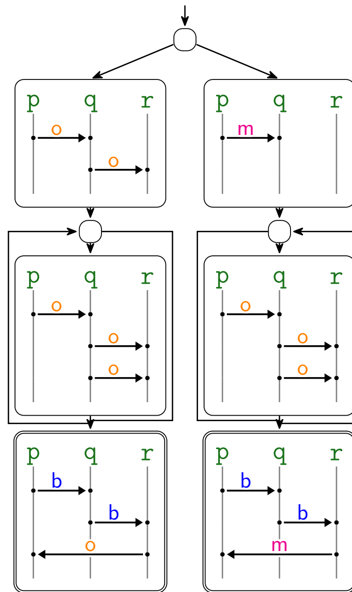
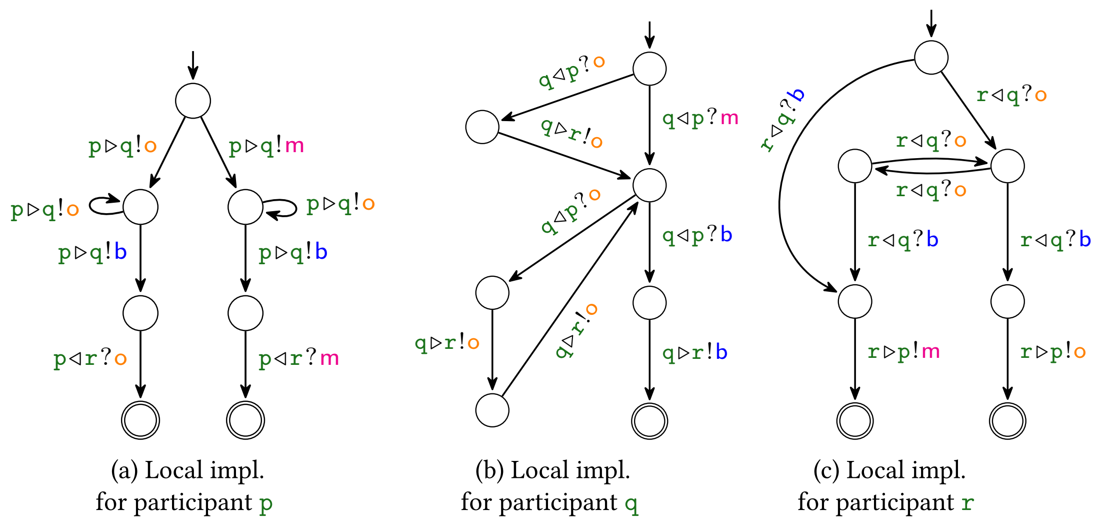


Figure 3.11: Odd-even Protocol:
implementable but not (yet) projectable.



(a) Local impl.
for participant p

(b) Local impl.
for participant q

(c) Local impl.
for participant r

Figure 3.12: Local implementations for the odd-even protocol.

Chapter 4

Building a Bridge from Multiparty Session Types to High-level Message Sequence Charts

In the previous chapter, we presented a generalised classical MST projection operator that was designed to support sender-driven choice. Despite, the classical projection approach is inherently incomplete, rejecting implementable protocols. In this chapter, we first establish a formal connection between global types and high-level message sequence charts. We use this to prove decidability of the implementability problem for sender-driven global types, under the mild assumption that every partial execution of the protocol can be extended to a finite completed execution. In addition, we investigate how results from the HMSC domain can be used for MSTs. Last, we propose a performance-oriented relaxation of the indistinguishability relation $\sim$ and show that the corresponding implementability problem is undecidable.

4.1 Encoding Global Types from MSTs as HMSCs

Non-deterministic global types can be turned into HMSCs while preserving the protocol they specify. The main difference between the automata-based semantics of global types from MSTs and the semantics of HMSCs is that an automaton carries the events on the edges and an HMSC carries events as labels of the event nodes in its BMSCs.

Definition 4.1 (HMSC encoding of global types)**.** In the translation, we use the following notation: $M_\emptyset$ is the empty BMSC ($N = \emptyset$) and $M(\mathtt{p} \to \mathtt{q} : \mathsf{m})$ is the BMSC with two event nodes e_1 and e_2 with $f(e_1) = e_2$, $l(e_1) = \mathtt{p} \triangleright \mathtt{q} ! \mathsf{m}$, and $l(e_2) = \mathtt{q} \triangleleft \mathtt{p} ? \mathsf{m}$. From a non-deterministic global type G, we construct an HMSC $H(G) = (V, E, v^I, V^T, \mu)$ as follows:

$$V := \{G' \mid G' \text{ is a subterm of } G\} \cup \\ \{(\sum_{i \in I} \mathtt{p}_i \to \mathtt{q}_i : \mathsf{m}_i \,.\, G_i, j) \mid \sum_{i \in I} \mathtt{p}_i \to \mathtt{q}_i : \mathsf{m}_i \,.\, G_i \text{ occurs in } G \text{ and } j \in I\}$$

$$
\begin{aligned}
E := &\{(\mu t \,.\, G',\, G') \mid \mu t \,.\, G' \text{ occurs in } G\} \cup \{(t,\, \mu t \,.\, G') \mid t \text{ and } \mu t \,.\, G' \text{ occur in } G\} \\
&\cup \{(\textstyle\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i,\, (\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i, j)) \mid (\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i, j) \in V\} \\
&\cup \{((\textstyle\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i, j),\, G_j) \mid (\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i, j) \in V\}
\end{aligned}
$$

$$
v^I := G \qquad V^T := \{0\}
$$

$$
\mu(v) := \begin{cases} M(\mathsf{p}_j \to \mathsf{q}_j : \mathsf{m}_j) & \text{if } v = (\sum_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \,.\, G_i, j) \\ M_\emptyset & \text{otherwise} \end{cases}
$$

This translation does not yield the HMSC with the least number of vertices since vertices with a single successor could be merged to form larger BMSCs. Here, every BMSC contains at most one message exchange.

Correctness of the Encoding

The correctness of the encoding of non-deterministic global types to HMSCs can be stated as follows.

Theorem 4.2. Let G be a non-deterministic global type. Then, the following holds:

(1) $\mathcal{L}(\mathsf{GAut}(G)) \subseteq \mathcal{L}(H(G))$ and
(2) $\mathcal{L}(G) = \mathcal{L}(H(G))$.

We prove this correspondence between a non-deterministic global type and its HMSC encoding as follows. First, we observe that all HMSC encodings have at most one message exchange in every BMSC, which we call 1-*HMSCs*. We can, thus, translate such an HMSC into an FSM. We relate this FSM with the one for the non-deterministic global type with a weak bisimulation. Throughout this subsection, we only consider words where send $\mathsf{p} \triangleright \mathsf{q} ! \mathsf{m}$ and receive events $\mathsf{q} \triangleleft \mathsf{p} ? \mathsf{m}$ happen right after each other. Thus, we treat words over Γ as words over Σ for conciseness, which we defined as syntactic sugar.

Let us first define 1-HMSCs.

Definition 4.3. We say that an HMSC $H = (V, E, v^I, V^T, \mu)$ is a 1-*HMSC* iff every BMSC in $\mu(V)$ consists of at most one send and one receive event.

1-HMSCs can be translated to FSMs as follows.

Definition 4.4 (Quasi-optimal translation of H)**.** Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC such that for every $v \in V$, $\mu(v) = M_\emptyset$ or $\mu(v) = M(\mathsf{p} \to \mathsf{q} : \mathsf{m})$ for some p, q and m. We define the function $\mathrm{qOPT}(M)$ if M is either $M_\emptyset$ or $M(\mathsf{p} \to \mathsf{q} : \mathsf{m})$:
$\mathrm{qOPT}(M_\emptyset) = \varepsilon$ and $\mathrm{qOPT}(M(\mathsf{p} \to \mathsf{q} : \mathsf{m})) = \mathsf{p} \to \mathsf{q} : \mathsf{m}$.
The state machine $\mathrm{qOPT}(H(G)) = (Q, \Sigma, \delta, q_0, F)$ is defined as follows:

$$
Q = \{v^1, v^2 \mid v \in V\} \qquad q_0 = \{v^1 \mid v = v^I\} \qquad F = \{v^2 \mid v \in V^T\}
$$

$$
\delta = \{(v^1, \mathrm{qOPT}(\mu(v)), v^2) \mid v \in V \wedge \mathrm{qOPT}(\mu(v)) \in \Sigma \cup \{\varepsilon\}\} \cup \{(v^2, \varepsilon, u^1) \mid (v, u) \in E\}
$$

Note that the HMSC $H(G)$ for any non-deterministic global type G qualifies for a quasi-optimal translation: each BMSC contains either no events or two events of corresponding send and receive.

Lemma 4.5 (Correctness of Definition 4.4)**.** Let H be a 1-HMSC. Then,

(1) $\mathcal{L}(\text{qOPT}(H)) \subseteq \mathcal{L}(H)$ and
(2) $\mathcal{C}^{\sim}(\mathcal{L}(\text{qOPT}(H))) = \mathcal{L}(H)$.

Proof. For (1), we prove the following:

Claim I: Let $v_1^1, v_1^2, v_2^1, v_2^2, \ldots, v_n^1, v_n^2$ be some run in $\text{qOPT}(H)$ with trace w. We claim that $w \in \mathcal{L}(\mu(v_1)\mu(v_2)\cdots\mu(v_n))$ with the same vertices.

Proof of Claim I. We prove this by induction on the number of vertices n. Note that, as $\mu(v_i)$ can be $M_\emptyset$ for some i, this is not necessarily an induction on $|w|$. For the base case, let $n = 0$. Then, the run is empty and the claim trivially holds. For the induction step, let us assume that the claim holds for n and w. We prove it for $n+1$ and w'. By construction $w' = wx$ where x is either ε or $\mathtt{p} \to \mathtt{q} : \mathtt{m}$ (technically $\mathtt{p} \triangleright \mathtt{q} ! \mathtt{m} \cdot \mathtt{q} \triangleleft \mathtt{p} ? \mathtt{m}$ but the previous is more concise). The run in $\text{qOPT}(H)$ is $v_1^1, v_1^2, \ldots, v_n^1, v_n^2, v_{n+1}^1, v_{n+1}^2$. By construction, $x \in \mathcal{L}(\mu(v_{n+1}))$. Hence, $wx \in \mathcal{L}(\mu(v_1)\cdots\mu(v_{n+1}))$, proving the claim.

End Proof of Claim I.

By construction, qOPT(-) translates final vertices to final states. Thus, Claim I shows the inclusion for the finite case when v_n is final:

$$\mathcal{L}(\text{qOPT}(H)) \cap \Sigma^* \subseteq \mathcal{L}(H) \cap \Sigma^*.$$

It remains to show the infinite case: $\mathcal{L}(\text{qOPT}(H)) \cap \Sigma^\omega \subseteq \mathcal{L}(H) \cap \Sigma^\omega$.

Let $w \in \mathcal{L}(\text{qOPT}(H)) \cap \Sigma^\omega$ be an infinite word. We show that there is some infinite path $v_1, v_2, \ldots$ in H for which one linearisation is w. Consider a tree $\mathcal{T}$ where each node corresponds to some path π in H whose linearisations are prefixes w' of w. The root is labelled by the empty path. The children of a node π are paths that extend π by a single node – these exist from the reasoning in the induction step for *Claim I*. HMSC H is finitely branching and so is $\mathcal{T}$. By König's Lemma, there is an infinite path in $\mathcal{T}$ whose linearisation is w, which concludes the proof of the first claim.

We prove (2) by proving two inclusions. The first inclusion $\mathcal{C}^{\sim}(\mathcal{L}(\text{qOPT}(H))) \subseteq \mathcal{L}(H)$ follows from the first claim and the fact that HMSCs are closed under $\sim$ (Lemma 2.21). For the second inclusion $\mathcal{L}(H) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(\text{qOPT}(H)))$, we first establish the following fact:

Claim II: Let $\pi = v_1, \ldots, v_n$ be some path in H. For all words

$$w \in \mathcal{L}(\mu(v_1)\cdots\mu(v_n)),$$

there is some $w' \in \mathcal{L}(\text{qOPT}(H))$ with some path $v_1^1, v_1^2, \ldots, v_n^1, v_n^2$ such that $w \sim w'$.

Proof of Claim II. We prove this by induction on the number of vertices n. For the base case, let $n = 0$. Then, the word is actually empty and the claim trivially holds. For the induction step, let us assume that the claim holds for n with w_n and w'_n and we prove it for $n+1$ with w_{n+1} and w'_{n+1}. We know that $w_{n+1} \in \mathcal{L}(\mu(v_1) \cdots \mu(v_{n+1}))$. By definition of the semantics for HMSCs, $w_{n+1} \sim w_n \cdot x$ $(*)$ such that $w_n \in \mathcal{L}(\mu(v_1) \cdots \mu(v_n))$ and $x \in \mathcal{L}(\mu(v_{n+1}))$. For w_n, there is w'_n with the run $v_1^1, v_1^2, \ldots, v_n^1, v_n^2$ in $\text{qOPT}(H)$. By construction, we can extend this run for $w'_n \cdot x$ as follows: $v_1^1, v_1^2, \ldots, v_n^1, v_n^2, v_{n+1}^1, v_{n+1}^2$. By the induction hypothesis and $(*)$, $w_{n+1} \sim w'_n \cdot x$, proving the claim.

End Proof of Claim II.

We prove that $\mathcal{L}(H) \subseteq \mathcal{C}^{\sim}(\mathcal{L}(\text{qOPT}(H)))$. Let $w \in \mathcal{L}(H)$.

In case w is finite, there is a finite path $\pi = v_1, \ldots, v_n$ with $w \in \mathcal{L}(\mu(v_1), \ldots, \mu(v_n))$ where v_n is final. We apply Claim II and know that v_n^2 is also final. We also know that $w \sim w'$ for some w' with a run through the corresponding states in the automaton $\text{qOPT}(H)$, proving the claim.

In case w is infinite, there is an infinite path $\pi = v_1, \ldots$ with $w \in \mathcal{L}(\mu(v_1) \ldots)$. For every prefix u of w, there is n such that $u \in \text{pref}(\mathcal{L}(\mu(v_1) \cdots \mu(v_n)))$. By Claim II, there is $u' \in \mathcal{L}(\text{qOPT}(H))$ with run $v_1^1, v_1^2, \ldots, v_n^1, v_n^2$ such that $u \sim u'$, entailing $u \preceq_{\sim} u'$. Consider a tree $\mathcal{T}$ where each node corresponds to a prefix of the run $v_1^1, v_1^2, \ldots$ in $\text{qOPT}(H)$. The root is labelled by the empty run. The children of a node ρ are the runs that extends ρ by a single transition – which exist by the above reasoning. Since $\mathcal{T}$ is finitely branching, there is an infinite path in $\mathcal{T}$ that corresponds to an infinite run ρ in $\text{qOPT}(H)$ for which $w \preceq^{\omega}_{\sim} \text{trace}(\rho)$, which concludes the proof. □

With weak bisimulations, we show that $\mathcal{L}(\mathsf{GAut}(G))$ and $\mathcal{L}(\text{qOPT}(H(G)))$ yield the same language for any non-deterministic global type G.

Definition 4.6 (Weak bisimulation)**.** Let

$$A = (Q_A, \Delta, \delta_A, q_{0,A}, F_A) \text{ and } B = (Q_B, \Delta, \delta_B, q_{0,B}, F_B)$$

be two state machines over some alphabet Δ. We say that $R \subseteq Q_A \times Q_B$ is a *weak simulation relation* for A and B if it is a relation with the following properties:

- for all $a \in \Delta, s_A, t_A \in S_A, s_B \in S_B$, if $R(s_A, s_B)$ and $(s_A, a, t_A) \in \delta_A$, then there is $t_B \in Q_B$ such that $(s_B, a, t_B) \in \delta_B^*$ and $R(t_A, t_B)$,
- for all $s_A, t_A \in S_A, s_B \in S_B$, if $R(s_A, s_B)$ and $(s_A, \varepsilon, t_A) \in \delta_A$, then there is $t_B \in Q_B$ such that $(s_B, \varepsilon, t_B) \in \delta_B^*$ and $R(t_A, t_B)$,
- $R(q_{0,A}, q_{0,B})$, and
- for every $q_A \in F_A$, there is $q_B \in F_B$ such that $R(q_A, q_B)$.

This definition is given as characterisation for the original definition of weak simulation by Milner [96, Def. 6.2 and Prop. 6.3]. In our context, it is easier to keep the two different sets of states separate rather than merging them.

Proposition 4.7. Let $A = (Q_A, \Delta, \delta_A, q_{0,A}, F_A), B = (Q_B, \Delta, \delta_B, q_{0,B}, F_B)$ be two state machines. Let $R_1 \subseteq Q_A \times Q_B$ and $R_2 \subseteq Q_B \times Q_A$ be two weak bisimulations for A and B (respectively B and A). Then, $\mathcal{L}(A) = \mathcal{L}(B)$.

We defined the semantics of global and local types using state machines where subterms were indexed and became states. It is straightforward that without indexing, we obtain the same language. Though, the resulting finite state machine might not be non-merging. This is why we decided to define the standard semantics with indexing. Here, we deviate from this for simplicity as we only care about the languages, simplifying the following proofs.

Lemma 4.8. Let G be a non-deterministic global type, $\mathsf{GAut}(G)$ the state machine with state space

$$Q = \{q_{G'} \mid G' \text{ is syntactic subterm of } G\}$$

where we omit indexing identical subterms and $\text{qOPT}(H(G))$ the state machine built from the HMSC $H(G)$ with states

$$V = \{v^i_{G'} \mid i \in \{1, 2\} \text{ and } G' \text{ is a syntactic subterm of } G\}$$

as defined before. There is one state and two vertices for every syntactic subterm of G. Hence, we index states q and vertices v by these subterms. For the auxiliary vertices in $\text{qOPT}(H(G))$, we use G^j. There are two weak bisimulation relations $R_1 \subseteq Q \times V$ and $R_2 \subseteq V \times Q$ such that

(1) $R_1(q_G, v^1_G)$ and $R_2(v^1_G, q_G)$ as well as
(2) $R_1(q_0, v^2_0)$ and $R_2(v^2_0, q_0)$.

Proof. We prove the claim by structural induction on G.

- Base $G = 0$:
  There is one state in $\mathsf{GAut}(G)$ and two vertices in $\text{qOPT}(H(G))$. It is straightforward to define R_1 and R_2 such that the conditions are satisfied. Both relations are weak bisimulation relations as there is a solely one ε-transition.
- Base $G = t$:
  This is no proper non-deterministic global type but the construction ensures that such states are properly related.
- Step $G = \Sigma_{i \in I} \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i \cdot G_i$:
  The induction hypothesis holds for every $i \in I$ with R^i_1 and R^i_2. We define:

$$R_1 := \cup_{i \in I} R^i_1 \cup \{(q_G, v^1_G), (q_G, v^2_G)\}, \text{ and}$$
$$R_2 := \cup_{i \in I} R^i_2 \cup \{(v^1_G, q_G), (v^2_G, q_G)\} \ \cup \ \cup_{i \in I}\{(v^1_{G^i}, q_{G_i}), (v^2_{G^i}, q_{G_i})\}$$

By this, (1) and (2) are satisfied. For the remaining conditions of a weak bisimulation relation, it suffices to check the added states (and states with new transitions) since the induction hypotheses are sufficient for the rest.
Added states: q_G as well as $v_G^1, v_G^2, v_{G^i}^1, v_{G^i}^2$ for every $i \in I$.
Added transitions: $(q_G, \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i, q_{G_i})$ for every $i \in I$ as well as $(v_G^1, \varepsilon, v_G^2)$, $(v_G^2, \varepsilon, v_{G^i}^1)$, $(v_{G^i}^1, \mathsf{p}_i \to \mathsf{q}_i : \mathsf{m}_i, v_{G^i}^2)$, and $(v_{G^i}^2, \varepsilon, v_{G_i}^1)$.
We check R_1 and hence q_G:

- $x \in \Sigma$ is the only possible transition from q_G to q_{G_i} for every $i \in I$; for v_G^1, we have that $R_1(q_G, v_G^1)$ and $v_G^1 \xrightarrow{\varepsilon} v_G^2 \xrightarrow{\varepsilon} v_{G^i}^1 \xrightarrow{x} v_{G^i}^2 \xrightarrow{\varepsilon} v_{G_i}^1$. This is fine by the induction hypotheses: $R_1^i(q_{G_i}, v_{G_i}^1)$. It happens that v_G^2, with which q_G is also related by R_1, occurs on the previous path, so it is also fine. There are no ε-transitions from q_G.

We check R_2 and hence $v_G^1, v_G^2, v_{G^i}^1, v_{G^i}^2$ for every $i \in I$:

- v_G^1:
  One can only take an ε-transition from v_G^1, i.e. $v_G^1 \xrightarrow{\varepsilon} v_G^2$; by definition we have that $R_2(v_G^2, q_G)$ and $R_2(v_G^1, q_G)$ and the transitive closure allows not to move.
- v_G^2:
  One can only take an ε-transition from v_G^2, i.e. $v_G^2 \xrightarrow{\varepsilon} v_{G^i}^1$ for every $i \in I$; by definition we have that $R_2(v_{G^i}, q_G)$; $R_2(v_G, q_G)$ and the transitive closure allows not to move.
- $v_{G^i}^1$:
  One can only take one non-ε-transition from $v_{G^i}^1$, i.e. $v_{G^i}^1 \xrightarrow{x} v_{G^i}^2$; by definition we have that $R_2(v_{G^i}^1, q_G)$ and $R_2(v_{G^i}^2, q_{G_i})$; we also have that $q_G \xrightarrow{x} q_{G_i}$ and, hence, the conditions are met.
- $v_{G^i}^2$:
  One can only take an ε-transition from $v_{G^i}^2$ to $v_{G_i}^1$; by definition we have that $R_2(v_{G^i}^2, q_{G_i})$, by the induction hypotheses we have that $R_2(v_{G_i}^1, q_{G_i})$ and the transitive closure allows not to move.

- Step $G = \mu t \,.\, G'$:
  The induction hypothesis holds for G' with R_1' and R_2'. Recall that $Q = \{q_{G'} \mid G' \text{ is a syntactic subterm of } G\}$. We define:

$$R_1 := R_1' \cup \{(q_G, v_G^1), (q_G, v_G^2)\}, \text{ and}$$
$$R_2 := R_2' \cup \{(v_G^1, q_G), (v_G^2, q_G)\}.$$

By this, (1) and (2) are already satisfied. For the conditions for a weak bisimulation relation, it suffices to check the added states and states with new transitions.
Added states: q_G as well as v_G^1 and v_G^2.

Added transitions: $(v_G^1, \varepsilon, v_G^2), (v_G^2, \varepsilon, v_{G'}^1), (v_t^2, \varepsilon, v_G^1)$ as well as $(q_G, \varepsilon, q_{G'})$ and (q_t, ε, q_G).
The induction hypotheses apply for all other states and transitions.
We check R_1 and hence q_G:

- One cannot take a non-ε-transition from q_G. The only ε-transition that can be taken is to $q_{G'}$; we only need to check v_G^1 as v_G^2 is on its path and also related; from $v_G^1 \xrightarrow{\varepsilon} v_G^2 \xrightarrow{\varepsilon} v_{G'}^1$ with an ε and by induction hypothesis they are related.

We check R_2 and hence v_G^1, v_G^2, and v_t^2.

- v_G^1:
  Only one ε-transition is possible to v_G^2 and both are related with the same state.
- v_G^2:
  Only one ε-transition is possible: $v_G^2 \xrightarrow{\varepsilon} v_{G'}^1$ and $q_G \xrightarrow{\varepsilon} q_{G'}$ and by definition the last two are related.
- v_t^2:
  Only one ε-transition is possible: $v_t^2 \xrightarrow{\varepsilon} v_G^1$ and $R_2(v_t^2, q_t)$; we have that $q_t \xrightarrow{\varepsilon} q_G$ and $R_2(v_G^1, q_G)$

□

Lemma 4.9. For any non-deterministic global type G, $\mathcal{L}(G) = \mathcal{L}(\text{qOPT}(H(G)))$.

Proof. Recall that $\mathcal{L}(G) = \mathcal{L}(\mathsf{GAut}(G))$. In Lemma 4.8, we show there are two weak simulation relations for $\mathsf{GAut}(G)$ and $\text{qOPT}(H(G))$ while Proposition 4.7 states the well-known fact that, then, both languages are equal. □

Equipped with this, we can now prove the correctness of the encoding of non-deterministic global types from MSTs into HMSCs.

Theorem 4.2. Let G be a non-deterministic global type. Then, the following holds:

(1) $\mathcal{L}(\mathsf{GAut}(G)) \subseteq \mathcal{L}(H(G))$ and
(2) $\mathcal{L}(G) = \mathcal{L}(H(G))$.

Proof. With the first fact of Lemma 4.5, it suffices to show

$$\mathcal{L}(\mathsf{GAut}(G)) \subseteq \mathcal{L}(\text{qOPT}(H(G)))$$

for (1), which follows with Lemma 4.9. For (2), it suffices to show that

$$\mathcal{C}^{\sim}(\mathcal{L}(\mathsf{GAut}(G))) \overset{\text{(A)}}{=} \mathcal{C}^{\sim}(\mathcal{L}(\text{qOPT}(H(G)))) \overset{\text{(B)}}{=} \mathcal{L}(H(G)).$$

Then, (A) follows from Lemma 4.9 while (B) follows from the second fact of Lemma 4.5.

□

4.2 MST Implementability is Decidable

In this section, we show decidability of the implementability problem for sender-driven global types, using results from the domain of message sequence charts. In general, implementability for HMSCs is undecidable but we show that global types, when encoded as HMSCs, belong to a class of HMSCs for which implementability is decidable.

The standard approach for the HMSC implementability problem is different from the classical projection approach to implementability. Given an HMSC, there is a canonical candidate implementation which always implements the HMSC if an implementation exists [5, Thm. 13]. Therefore, approaches centre on checking implementability of HMSC languages and establishing conditions on HMSCs that entail implementability.

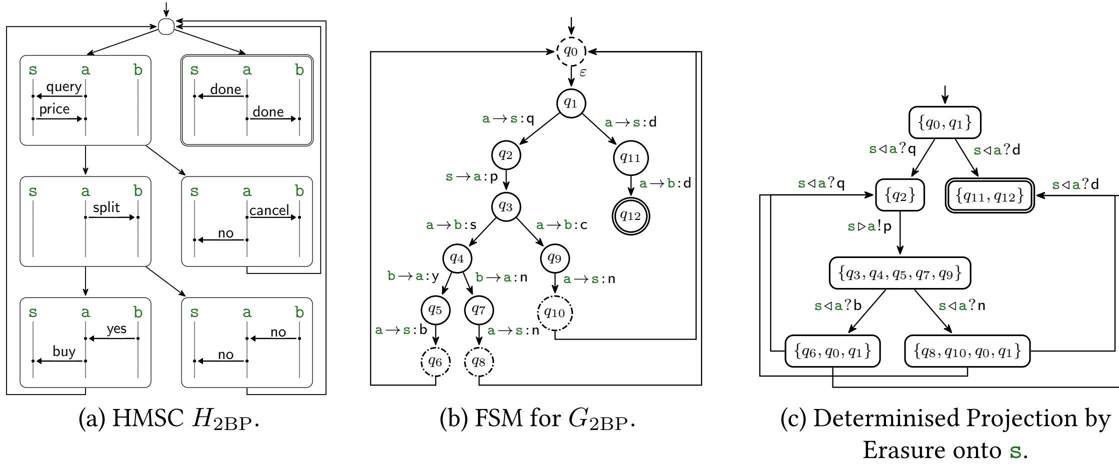
Definition 4.10 (Canonical candidate implementation [5]). Given an HMSC H and a participant p, let $A'_{\mathrm{p}} = (Q', \Gamma_{\mathrm{p}}, \delta', q'_0, F')$ be a state machine with $Q' := \{q_w \mid w \in \mathrm{pref}(\mathcal{L}(H){\Downarrow_{\Gamma_{\mathrm{p}}}})\}$, $F' := \{q_w \mid w \in \mathcal{L}_{\mathrm{fin}}(H){\Downarrow_{\Gamma_{\mathrm{p}}}}\}$, and $\delta'(q_w, x, q_{wx})$ for $x \in \Gamma$. We determinise A'_{p} to obtain the state machine A_{p}. We call $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ the canonical candidate implementation of H.

Intuitively, the intermediate state machine A'_{p} constitutes a tree whose maximal finite paths give $\mathcal{L}(H){\Downarrow_{\Gamma_{\mathrm{p}}}} \cap \Gamma_{\mathrm{p}}^*$. This set can be infinite and, thus, the construction might not be effective. We give an effective construction of a deterministic FSM for the same language which was very briefly hinted at by Alur et al. [6, Proof of Thm. 3].

Definition 4.11 (Projection by erasure). Let $M = (N, p, f, l, (\le_{\mathrm{p}})_{\mathrm{p}\in\mathcal{P}})$ be an MSC and p be a participant. We denote the set of nodes of p with $N_{\mathrm{p}} := \{n \mid p(n) = \mathrm{p}\}$ and define a two-ary next-relation on N_{p}: $\mathrm{next}(n_1, n_2)$ iff $n_1 \lneq n_2$ and there is no n' with $n_1 \lneq n' \lneq n_2$. We define the projection by erasure of M onto p:

$$\begin{aligned} M{\Downarrow_{\mathrm{p}}} &:= (Q_M, \Gamma_{\mathrm{p}}, \delta_M, q_{M,0}, \{q_{M,f}\}) \quad \text{with} \\ Q_M &:= \{q_n \mid n \in N_{\mathrm{p}}\} \uplus \{q_{M,0}\} \uplus \{q_{M,f}\} \quad \text{and} \\ \delta_M &:= \{q_{M,0} \xrightarrow{\varepsilon} q_{n_1} \mid \forall n_2.\, n_1 \le n_2\} \uplus \{q_{n_1} \xrightarrow{l(n_1)} q_{n_2} \mid \mathrm{next}(n_1, n_2)\} \\ &\quad \uplus \{q_{n_2} \xrightarrow{l(n_2)} q_{M,f} \mid \forall n_1.\, n_1 \le n_2\} \end{aligned}$$

where $\uplus$ denotes disjoint union. Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We construct the projection by erasure for every vertex and identify them with the vertex, e.g. Q_v instead of $Q_{\mu(v)}$. We construct an auxiliary FSM $(Q'_H, \Gamma_{\mathrm{p}}, \delta'_H, q'_{H,0}, F'_H)$ with $Q'_H = \biguplus_{v\in V} Q_v$, $\delta'_H = \biguplus_{v\in V} \delta_v \uplus \{q_{v_1,f} \xrightarrow{\varepsilon} q_{v_2,0} \mid (v_1, v_2) \in E\}$, $q'_{H,0} = q_{v^I,0}$, and $F'_H = \biguplus_{v\in V^F} q_{v,f}$. We determinise $(Q'_H, \Gamma_{\mathrm{p}}, \delta'_H, q'_{H,0}, F'_H)$ to obtain $H{\Downarrow_{\mathrm{p}}} := (Q_H, \Gamma_{\mathrm{p}}, \delta_H, q_{H,0}, F_H)$, which is the *determinised projection by erasure of H onto* p. The CSM formed from the projections by erasure $\{\!\{H{\Downarrow_{\mathrm{p}}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ is called *erasure candidate implementation*.



(a) HMSC $H_{2\text{BP}}$. (b) FSM for $G_{2\text{BP}}$. (c) Determinised Projection by Erasure onto s.

Figure 4.1: Two buyer protocol: as HMSC and FSM as well as its determinised projection by erasure onto s.

Example 4.12. Figure 4.1a represents the two buyer protocol from Section 3.1.1 as HMSC. In Fig. 4.1b, we also give the representation as state machine again. Figure 4.1c is the determinised projection by erasure onto seller s. Each state is labelled with a set of states which correspond to states in Fig. 4.1b. Intuitively, these states have a correspondence in the HMSC but it is easier to see the connection with the FSM representation. ◀

Lemma 4.13 (Correctness of determinised projection by erasure)**.** Let H be an HMSC, p be a participant, and $H{\Downarrow}_{\mathsf{p}}$ be its determinised projection by erasure. Then, the following language equality holds: $\mathcal{L}(H{\Downarrow}_{\mathsf{p}}) = \mathcal{L}(H){\Downarrow}_{\Gamma_{\mathsf{p}}}$.

Proof. Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. For every $v \in V$, it is straightforward that the construction of $\mu(v){\Downarrow}_{\mathsf{p}}$ yields $\mathcal{L}(\mu(v)){\Downarrow}_{\Gamma_{\mathsf{p}}} = \mathcal{L}(\mu(v){\Downarrow}_{\mathsf{p}})$ (1). Recall that $\sim$ does not reorder events by the same participant (2).

The following reasoning proves the claim where the first equivalence follows from the construction of the transition relation of $H{\Downarrow}_{\mathsf{p}}$:

$$
\begin{aligned}
& w \in \mathcal{L}(H{\Downarrow}_{\mathsf{p}}) \\
\Leftrightarrow\quad & w = w_1 \ldots, \text{ there is a path } v_1, \ldots \text{ in } H \text{ and } w_i \in \mathcal{L}(\mu(v_i){\Downarrow}_{\mathsf{p}}) \text{ for every } i \\
\overset{(1)}{\Leftrightarrow}\quad & w = w_1 \ldots, \text{ there is a path } v_1, \ldots \text{ in } H \text{ and } w_i \in \mathcal{L}(\mu(v_i)){\Downarrow}_{\Gamma_{\mathsf{p}}} \text{ for every } i \\
\overset{(2)}{\Leftrightarrow}\quad & w \in \mathcal{L}(H){\Downarrow}_{\Gamma_{\mathsf{p}}}
\end{aligned}
$$

This proves the fact for infinite words. It is straightforward to adapt the proof for finite words. □

From this result and the construction of the canonical candidate implementation, it follows that the determinised projection by erasure admits the same finite language.

Corollary 4.14. Let H be an HMSC, $\mathtt{p}$ be a participant, $H{\Downarrow_{\mathtt{p}}}$ be its determinised projection by erasure, and $A_{\mathtt{p}}$ be the canonical candidate implementation. Then, it holds that $\mathcal{L}_{\text{fin}}(H{\Downarrow_{\mathtt{p}}}) = \mathcal{L}_{\text{fin}}(A_{\mathtt{p}})$.

The determinised projection by erasure can be computed effectively and is deterministic. Thus, we use it in place of the canonical candidate implementation. Given a global type, the erasure candidate implementation for its HMSC encoding implements it if it is implementable.

Theorem 4.15. Let G be a global type and $\{\!\{H(G){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be its erasure candidate implementation. If $\mathcal{L}_{\text{fin}}(G)$ is implementable, then $\{\!\{H(G){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free and $\mathcal{L}_{\text{fin}}(\{\!\{H(G){\Downarrow_{\mathtt{p}}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{fin}}(G)$.

Proof. We first use the correctness of the global type encoding (Theorem 4.2) to observe that $\mathcal{L}_{\text{fin}}(G) = \mathcal{L}_{\text{fin}}(H(G))$. Theorem 13 by Alur et al. [5] states that the canonical candidate implementation implements $\mathcal{L}_{\text{fin}}(H(G))$ if it is implementable. Corollary 4.14 and the fact that the FSM for each participant is deterministic by construction allows us to replace every $A_{\mathtt{p}}$ from the canonical candidate implementation with the determinised projection by erasure $H(G){\Downarrow_{\mathtt{p}}}$ for every participant $\mathtt{p}$, proving the claim. □

This result does only apply to finite languages so we need to extend it for infinite words. For this, we require a mild assumption on global types. Intuitively, every run of the protocol should always be able to terminate but does not need to, leaving a possibility for every run to terminate but it can be infinite.

Definition 4.16 (0-Reachable)**.** We say a global type G is 0-reachable if every prefix of a word in its language can be completed to a finite word in its language. Equivalently, it is satisfied if the vertex for the syntactic subterm 0 is reachable from any vertex in $H(G)$.

This assumption constitutes a structural property of a protocol and no fairness condition on runs of the protocol. Basically, this solely rules out global types that have loops without exit. In practice, it is reasonable to assume a mechanism to terminate a protocol for maintenance for instance. In theory, one can think of protocols that are not 0-reachable. They would simply recurse indefinitely and can never terminate. This allows interesting behaviour like two sets of participants that do not interact with each other, as the following example shows.

Example 4.17. Consider the following global type: $\mu t\,.\,\mathtt{p}\!\to\!\mathtt{q}\!:\!\mathtt{m}_1\,.\,\mathtt{r}\!\to\!\mathtt{s}\!:\!\mathtt{m}_2\,.\,t$. It describes an infinite execution with two pairs of participants that independently send and receive messages. This can be implemented in an infinite setting but the loop can never be exited due to the lack of synchronisation, breaking protocol fidelity upon termination. ◀

Under the 0-reachability assumption, we can show that implementations for finite languages generalise for infinite ones.

Lemma 4.18 (Implementation for finite language generalises to infinite language for 0-reachable global types)**.** Let G be a 0-reachable global type that does not necessarily satisfy sender-driven choice and $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be an implementation for $\mathcal{L}_{\text{fin}}(G)$. Then, it holds that $\mathcal{L}_{\text{inf}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{inf}}(G)$. Consequently, $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ implements $\mathcal{L}(G)$.

Proof. By assumption, we know that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deadlock-free and $\mathcal{L}_{\text{fin}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{fin}}(G)$. We prove $\mathcal{L}_{\text{inf}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) = \mathcal{L}_{\text{inf}}(G)$ by showing both inclusions.

First, we show that $\mathcal{L}_{\text{inf}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}) \subseteq \mathcal{L}_{\text{inf}}(G)$. For this direction, let w be a word in $\mathcal{L}_{\text{inf}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. We need to show that there is a run ρ in $\mathsf{GAut}(G)$ such that $w \preceq^{\omega}_{\sim} \text{trace}(\rho)$. Since G is 0-reachable, we know that for every $u \in \text{pref}(w)$, it holds that $u \in \text{pref}(\mathcal{L}_{\text{fin}}(G))$. Thus, there exists a finite run ρ (that does not necessarily end in a final state) and u' such that $u{\cdot}u' \sim \text{trace}(\rho)$. We call ρ a witness run. Intuitively, we need to argue that every such witness run for u can be extended when appending the next event x from w to obtain ux. In general, this does not hold for every choice of witness run. However, because of monotonicity, any run (or rather a prefix of it) for an extension ux can also be used as witness run for u. Thus, we make use of the idea of prophecy variables [1] and assume an oracle which picks the correct witness run for every prefix u. This oracle does not restrict the next possible events in any way. We apply the same idea as in the end of the proof for Lemma 2.10 but for $\mathsf{GAut}(G)$. Consider a tree $\mathcal{T}$ where each node represents a run in $\mathsf{GAut}(G)$ such that their traces are finite prefixes w' of w. The root's label is the empty run. For every node labelled with ρ, the children's labels extend ρ by a single transition. The tree $\mathcal{T}$ is finitely branching by construction of $\mathsf{GAut}(G)$. With König's Lemma, we obtain an infinite path in $\mathcal{T}$ and thus an infinite run ρ in $\mathsf{GAut}(G)$ with $w \preceq^{\omega}_{\sim} \text{trace}(\rho)$. From this, it follows that $w \in \mathcal{L}_{\text{inf}}(G)$.

Second, we show that $\mathcal{L}_{\text{inf}}(G) \subseteq \mathcal{L}_{\text{inf}}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. Let w be a word in $\mathcal{L}_{\text{inf}}(G)$. Eventually, we will apply the same reasoning with König's Lemma to obtain an infinite run in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for w. Inspired by the first statement of Lemma 3.21, we show:

(i) for every prefix $w' \in \text{pref}(w)$, there is a run ρ' in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ with $w' \preceq \text{trace}(\rho')$, and

(ii) for every extension $w'x$ with $w'x \in \text{pref}(w)$, the run ρ' can be extended.

We prove Claim (i) first. We first observe that, since G is 0-reachable, there is an extension w'' of w' with $w'' \in \mathcal{L}(G)$. By construction, we know that there is a run ρ'' in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for w''. For ρ', we can simply take the prefix of ρ'' that matches w'. This proves Claim (i).

Now, let us prove Claim (ii). Similar to the first case, we will use prophecy variables [1] and an oracle to pick the correct witness run that we can extend. Again, because of monotonicity, any run (or rather a prefix of it) for an extension $w'x$ can also be used as witness run for w' and this oracle does not restrict the participants in any way. From this, Claim (ii) follows.

From here, we can reason similarly to the previous case and, in fact, analogously to the end of the proof for Lemma 2.10. From this, it follows that $w \in \mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$. ☐

Corollary 4.19. Let G be a 0-reachable implementable global type. Then, the erasure candidate implementation $\{\!\{H(G)\Downarrow_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ implements G.

Even if a global type is not implementable, the erasure candidate implementation does not remove any behaviour for any participant. Thus, it generates at least the behaviours of the global types.

Proposition 4.20. Let G be some 0-reachable global type. Then, the following inclusion holds: $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{H(G)\Downarrow_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}})$.

This proposition is rather obvious and can be proven analogously to Lemma 3.21 and Lemma 5.5. We therefore omit the proof.

So far, we have shown that, if G is implementable, the erasure candidate implementation for its HMSC encoding $H(G)$ implements G. For HMSCs, implementability is undecidable in general [91]. We show that, because of their syntactic restrictions on choice, global types belong to the class of globally-cooperative HMSCs for which implementability is decidable.

Definition 4.21 (Communication graph [55])**.** Let $M = (N, p, f, l, (\leq_{\mathsf{p}})_{\mathsf{p}\in\mathcal{P}})$ be an MSC. The *communication graph* of M is a directed graph with node p for every participant p that sends or receives a message in M and edges $\mathsf{p} \to \mathsf{q}$ if M contains a message from p to q, i.e. there is $e \in N$ such that $p(e) = \mathsf{p}$ and $p(f(e)) = \mathsf{q}$.

It is important to note that the nodes in the communication graph of M do only represent active participants, i.e. the ones that send or receive in M.

Definition 4.22 (Globally-cooperative HMSCs [55])**.** An HMSC $H = (V, E, v^I, V^T, \mu)$ is called *globally-cooperative* if for every loop, i.e. $v_1, \ldots, v_n$ with $(v_i, v_{i+1}) \in E$ for every $1 \leq i < n$ and $(v_n, v_1) \in E$, the communication graph of $\mu(v_1) \ldots \mu(v_n)$ is weakly connected, i.e. all nodes are connected if every edge is considered undirected.

We can check this directly for a global type G. It is straightforward to define a communication graph for words from Σ^*. We check it on $\mathsf{GAut}(G)$: for each binder state, we check the communication graph for the shortest trace to every corresponding recursion state.

Theorem 4.23 (Thm. 3.7 [91])**.** Let H be a globally-cooperative HMSC. Restricted to its finite language $\mathcal{L}_{\text{fin}}(H)$, implementability is EXPSPACE-complete.

Now, we prove that every implementable global type is globally-cooperative. Recall that we implicitly assume that global types satisfy sender-driven choice, unless explicitly mentioned otherwise.

Lemma 4.24. Let G be an implementable 0-reachable global type. Then, its HMSC encoding $H(G)$ is globally-cooperative.

Proof. We prove our claim by contraposition: assume there is a loop $v_1, \dots, v_n$ such that the communication graph of $\mu(v_1) \dots \mu(v_n)$ is not weakly connected. By construction of $H(G)$, we know that every vertex is reachable so there is a path $u_1 \dots u_m v_1 \dots v_n$ in $H(G)$ for some m and vertices u_1 to u_m such that $u_1 = v^I$. Because G is 0-reachable, this path can be completed to end in a final vertex to obtain

$$u_1 \dots u_m v_1 \dots v_n u_{m+1} \dots u_{m+k}$$

for some k and vertices u_{m+1} to u_{m+k} such that $u_{k+m} \in V^T$. By the syntax of global types and the construction of $H(G)$, there is a participant $\mathtt{p}$ that is the (only) sender in v_1 and u_{m+1}.

Without loss of generality, let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the two sets of (active) participants whose communication graphs of $v_1 \dots v_n$ are weakly connected and their union consists of all active participants. Similar reasoning applies if there are more than two sets.

We want to consider specific linearisations from the language of the BMSC for each subpath. Intuitively, these simply follow the order prescribed by the global type and do not exploit the partial order of BMSCs or the closure of the semantics for global types. For this, we say that w_1 is the *canonical word for path* $u_1, \dots u_m$ if $w_1 \in \{w'_1 \dots w'_m \mid w'_i \in \mathcal{L}(\mu(u_i)) \text{ for } 1 \leq i \leq m\}$. Analogously, let w_2 be the canonical word for $v_1 \dots v_n$ and w_3 be the canonical word for $u_{m+1} \dots u_{m+k}$. Without loss of generality, $\mathcal{S}_1$ contains the sender of the first element in w_2 and w_3 — basically the participant that decides when to exit the loop for the considered loop branch. By its definition and the correctness of $H(G)$, it holds that: $\mathcal{L}(G) = \mathcal{L}(H(G))$. Let $\{\!\{H(G)\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ be the erasure candidate implementation. From Proposition 4.20, we know that

$$\mathcal{L}(H(G)) \subseteq \mathcal{L}(\{\!\{H(G)\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}).$$

Therefore, we know that $w_1 \cdot w_2 \cdot w_3 \in \mathcal{L}(\{\!\{H(G)\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}})$.

From the construction of $H(G)$ and the construction of w_i for $i \in \{1, 2, 3\}$, it also holds that $w_1 \cdot (w_2)^h \cdot w_3 \in \mathcal{L}(H(G)) \subseteq \mathcal{L}(\{\!\{H(G)\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}})$ for any $h > 0$.

By construction of $\mathcal{S}_1$ and $\mathcal{S}_2$, no two participants from both sets communicate with each other in w_2: there are no $\mathtt{r} \in \mathcal{S}_1$ and $\mathtt{s} \in \mathcal{S}_2$ such that $\mathtt{r} \triangleright \mathtt{s} ! \mathsf{m}$ is in w_2 or $\mathtt{s} \triangleright \mathtt{r} ! \mathsf{m}$ is in w_2 (and consequently $\mathtt{r} \triangleleft \mathtt{s} ? \mathsf{m}$ is in w_2 or $\mathtt{s} \triangleleft \mathtt{r} ? \mathsf{m}$ is in w_2) for any m.

From the previous two observations, it follows that

$$w_1 \cdot w_2 \cdot (w_2 \Downarrow_{\Gamma_{\mathcal{S}_1}})^h \cdot w_3 \in \mathcal{L}(\{\!\{H(G)\Downarrow_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}})$$

for any h where $\Gamma_{\mathcal{S}_1} = \bigcup_{\mathtt{r} \in \mathcal{S}_1} \Gamma_{\mathtt{r}}$. Intuitively, this means that the set of participants with the participant to decide when to exit the loop can continue longer in the loop than the participants in $\mathcal{S}_2$.

With $\mathcal{L}(G) = \mathcal{L}(H(G))$, it suffices to show the following to find a contradiction: $w_1 \cdot w_2 \cdot (w_2 \Downarrow_{\Gamma_{\mathcal{S}_1}}) \cdot w_3 \notin \mathcal{L}(H(G))$.

Towards a contradiction, we assume the membership holds. By determinacy of $H(G)$, we need to find a path $v'_1 \ldots v'_{m'}$, that starts at the beginning of the loop, i.e. $v'_1 = v_1$, with canonical word w_4 such that $w_2{\Downarrow_{\Gamma_{\mathcal{S}_1}}} \cdot w_3 \sim w_4$.

We show such a path for w_4 does not exist.

We denote $w_2 \cdot w_3$ with $x := x_1 \ldots x_l$ and $w_2{\Downarrow_{\Gamma_{\mathcal{S}_1}}} \cdot w_3$ with $x' := x'_1 \ldots x'_{l'}$. We know that x' is a subsequence of x. Let $x_1 \ldots x_j = x'_1 \ldots x'_j$ denote the maximal prefix on which both agree. Since $\mathcal{S}_2$ is not empty, we know that j can be at most $|w_2{\Downarrow_{\Gamma_{\mathcal{S}_1}}}|$. (Intuitively, j cannot be so big that it reaches w_3 because there will be mismatches due to $w_2{\Downarrow_{\Gamma_{\mathcal{S}_2}}}$ before.) We also claim that the next event x_{j+1} cannot be a receive event. If it was, there was a matching send event in $x_1 \ldots x_j$ (which is equal to $x'_1 \ldots x'_j$ by construction). Such a matching send event exists by construction of x from a path in $H(G)$. By definition of $\Downarrow_{_}$, the matching receive event must be x'_{j+1} which would contradict the maximality of j. Thus, x_{j+1} must be a send event.

By determinacy of $H(G)$ and $j \leq |w_2{\Downarrow_{\Gamma_{\mathcal{S}_1}}}|$, we know that $x_1 \ldots x_j = x'_1 \ldots x'_j$ share a path $v_1 \ldots v_{n'}$ which is a part of the loop, i.e. $x_1 \ldots x_j \in \mathcal{L}(\mu(v_1) \cdots \mu(v_{n'}))$ with $n' < n$. For $M(\mathtt{p} \to \mathtt{q} : \mathsf{m})$, the BMSC with solely this interaction, we say that $\mathtt{p}$ is its *sender*. The syntax of global types prescribes that choice is deterministic and the sender in a choice is unique. This is preserved for $H(G)$: for every vertex, all its successors have the same sender. Therefore, the path for x' can only diverge — but also needs to diverge — from the loop $v_1 \ldots v_n$ after the common prefix $v_1 \ldots v_{n'}$ with a different send event but with the same sender. Let v_l be the next vertex after $v_1 \ldots v_{n'}$ on the loop $v_1, \ldots, v_n$ for which $\mu(v_l)$ is not M_ε, the BMSC with an empty set of event nodes. Note that x_{j+1} belongs to v_l: $x_{j+1} \in \operatorname{pref}(\mathcal{L}(\mu(v_l)))$.

We do another case analysis whether x_{j+1} belongs to $\mathcal{S}_1$ or not, i.e. if $x_{j+1} \in \Gamma_{\mathcal{S}_1}$.

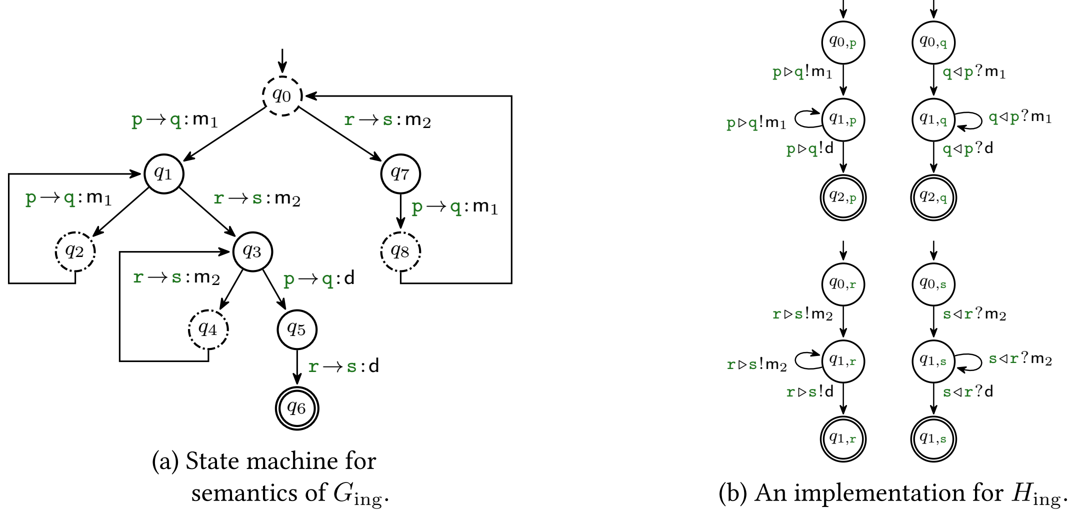
If $x_{j+1} \notin \Gamma_{\mathcal{S}_1}$, there cannot be a path that continues for $x'_{j+1} \in \Gamma_{\mathcal{S}_1}$ as the sender for $\mu(v_l)$ is not in $\mathcal{S}_1$. If $x_{j+1} \in \Gamma_{\mathcal{S}_1}$, the choice of j was not maximal, which yields a contradiction. □

Every implementable (sender-driven) global type is globally-cooperative. Let us show that this is not true for mixed-choice global types and, thus, HMSCs in general.

Example 4.25 (Implementable mixed-choice global type but not globally-cooperative). Let us consider the following mixed-choice global type:

$$G_{\text{ing}} := \mu t_1 \,.\, + \begin{cases} \mathtt{r} \to \mathtt{s} : \mathsf{m}_2 \,.\, \mathtt{p} \to \mathtt{q} : \mathsf{m}_1 \,.\, t_1 \\ \mathtt{p} \to \mathtt{q} : \mathsf{m}_1 \,.\, \mu t_2 \,.\, + \begin{cases} \mathtt{p} \to \mathtt{q} : \mathsf{m}_1 \,.\, t_2 \\ \mathtt{r} \to \mathtt{s} : \mathsf{m}_2 \,.\, \mu t_3 \,.\, + \begin{cases} \mathtt{r} \to \mathtt{s} : \mathsf{m}_2 \,.\, t_3 \\ \mathtt{p} \to \mathtt{q} : \mathsf{done} \,.\, \mathtt{r} \to \mathtt{s} : \mathsf{done} \,.\, 0 \end{cases} \end{cases} \end{cases} .$$

Its FSM is visualised in Fig. 4.2a, where we merged binder states with their only successor for conciseness. Intuitively, this protocol allows two pairs of participants to interact any number of times independently. The global type does not satisfy sender-driven choice and the protocol cannot be represented by a global type that does. One could swap the two interactions in the first loop but the first action of the initial



(a) State machine for semantics of G_{ing}.

(b) An implementation for H_{ing}.

Figure 4.2: An implementable HMSC which is not globally-cooperative with its implementation.

choice would be the same, making it non-deterministic. The protocol is not globally-cooperative but still implementable. Let us explain this in more detail. It comprises three loops. In the first one, $\texttt{r}$ sends a message $\texttt{m}_2$ to $\texttt{s}$ while $\texttt{p}$ sends a message $\texttt{m}_1$ to $\texttt{q}$ so the communication graph is not weakly connected, violating a criterion to be globally-cooperative. Intuitively, the first loop requires that the same number of messages are exchanged between the two pairs of participants but this is impossible due to lack of communication between both. In the second loop, only the interaction between $\texttt{p}$ and $\texttt{q}$ is specified, while, in the third one, it is only the one between $\texttt{r}$ and $\texttt{s}$. Thus, the second and third loop make up for the missing synchronisation in the first loop, allowing each pair to have a number of message exchanges that is independent from the other pair. Thus, the protocol is implementable and the CSM in Fig. 4.2b implements it. ◀

We now have all ingredients for our EXPSPACE decision procedure for MST implementability (with sender-driven choice).

Theorem 4.26. Checking implementability of 0-reachable global types with sender-driven choice is in EXPSPACE.

Proof. Let G be a 0-reachable global type with sender-driven choice. We construct $H(G)$ from G and check if it is globally-cooperative. For this, we apply the coNP-algorithm by Genest et al. [55] which is based on guessing a subgraph and checking its communication graph. If $H(G)$ is not globally-cooperative, we know from Lemma 4.24 that G is not implementable. If $H(G)$ is globally-cooperative, we check implementability of $\mathcal{L}_{\text{fin}}(H(G))$. By Theorem 4.23, this is in EXPSPACE. If $\mathcal{L}_{\text{fin}}(H(G))$ is not implementable, it trivially follows that G is not implementable. If $\mathcal{L}_{\text{fin}}(H(G))$ is implementable, G can be implemented with the erasure candidate implementation with Theorem 4.15 and Lemma 4.18. □

Consequently, the implementability problem for global types with sender-driven choice is decidable.

Corollary 4.27. Let G be a 0-reachable global type with sender-driven choice. It is decidable whether G is implementable and there is an algorithm to obtain its implementation for G.

Lower Bounds for Implementability. For general globally-cooperative HMSCs, i.e. that are not necessarily the encoding of a sender-driven global type, implementability is EXPSPACE-hard [91]. This hardness result does not carry over for the HMSC encoding $H(G)$ of a sender-driven global type G. The construction exploits that HMSCs do not impose any restrictions on choice. Sender-driven global types, however, require every branch to be chosen by a single sender. In Section 5.6, we present a family of global types for which generating an implementation requires exponential time in the size of the input. In the next section, we investigate different MSC techniques that could be used in the MST setting.

On the Synchronous Implementability Problem. We could not find a reference that shows decidability of the implementability problem in a synchronous setting, i.e. without channels. Before giving a proof sketch, let us remark that there are global types that can be implemented synchronously but not asynchronously, e.g. $\mathtt{p} \to \mathtt{q} : \mathsf{l} . \mathtt{r} \to \mathtt{q} : \mathsf{l} . 0 + \mathtt{p} \to \mathtt{q} : \mathsf{r} . \mathtt{r} \to \mathtt{q} : \mathsf{r} . 0$ because $\mathtt{q}$ can force the right choice by $\mathtt{r}$. We sketch how one could prove decidability of the synchronous implementability problem for global types (with sender-driven choice). One defines the synchronous semantics of CSMs and HMSCs as expected. For global types, one uses the independence relation $\mathcal{I}$ (cf. Definition 4.28), which defines reasonable reorderings for synchronous events in a distributed setting, similar to the indistinguishability relation $\sim$. It is straightforward that our HMSC encoding $H(\text{-})$ for global types also works for the synchronous setting (cf. Theorem 4.2). Thus, every implementation for $H(G)$ is also an implementation for G. For the asynchronous setting, we used [5, Thm. 13], which shows that the canonical candidate implementation implements an HMSC if it is implementable. Alur et al. [5, Sec. 8] also considered the synchronous setting. They observe that both Theorem 5 and 8, basis for Theorem 13, stay valid under these modified conditions. Together with our results, the erasure candidate implementation implements a global type if it is implementable. We expect that, also for the synchronous setting, sender-driven choice entails that any implementable global type is globally-cooperative (cf. Lemma 4.24), yielding a decision procedure similar to Theorem 4.26. Closest are works by Jongmans and Yoshida [76] and Glabbeek et al. [60]. Jongmans and Yoshida consider quite restrictive synchronous semantics for global types [76, Ex. 3] that does not allow the natural reorderings in a distributed setting, as enabled by $\mathcal{I}$, e.g. $\mathtt{p} \to \mathtt{q} : \mathsf{m} . \mathtt{r} \to \mathtt{s} : \mathsf{m} . 0$ is considered unimplementable. Glabbeek et al. [60] present

a projection operator that is complete for various notions of lock-freedom, a typical liveness property, and investigate how much fairness is required for those.

4.3 MSC Techniques for MST Verification

In the previous section, we generalised results from the MSC literature to show decidability of the implementability problem for global types from MSTs, yielding an EXPSPACE-algorithm. However, the resulting algorithm suffers from high complexity. This is also true for the original problem of safe realisability of HMSCs. In fact, the problem is undecidable for HMSCs in general. Besides globally-cooperative HMSCs, further restrictions of HMSCs have been studied to obtain algorithms with better complexity. In this section, we elaborate on their applicability to global types. One solely needs to check that the global type (or its HMSC encoding) belongs to the respective class. First, we transfer the algorithms for $\mathcal{I}$-closed HMSCs, which requires an HMSC not to exhibit certain anti-patterns of communication. Second, we present a variant of the implementability problem. It can make unimplementable global types implementable without changing a protocol's structure. Here, we restrict ourselves to MSC results that can directly be applied to the MST setting. For HMSC approaches that introduced the idea of choice to HMSCs, we refer to Chapter 10.

4.3.1 $\mathcal{I}$-closed Global Types

The implementability problem is EXPSPACE-complete for the class of globally-cooperative HMSCs. Membership in EXPSPACE was shown by reducing the problem to implementability of $\mathcal{I}$-closed HMSCs [91, Thm. 3.7]. These require the language of an HMSC to be closed with regard to an independence relation $\mathcal{I}$, where, intuitively, two interactions are independent if there is no participant that is involved in both. Implementability for $\mathcal{I}$-closed HMSCs is PSPACE-complete [91, Thm. 3.6]. As for the EXPSPACE-hardness for globally-cooperative HMSCs, the PSPACE-hardness exploits features that cannot be modelled with global types.

We adapt the definitions from [91] to our setting. These consider atomic BMSCs, which are BMSCs that cannot be split further. With the HMSC encoding for global types, it is straightforward that atomic BMSCs correspond to individual interactions for global types. Thus, we define the independence relation $\mathcal{I}$ on the alphabet Σ.

Definition 4.28 (Independence relation $\mathcal{I}$)**.** We define the independence relation $\mathcal{I}$ on Σ:

$$\mathcal{I} := \{(\mathtt{p}\to\mathtt{q}:_, \mathtt{r}\to\mathtt{s}:_) \mid \{\mathtt{p},\mathtt{q}\} \cap \{\mathtt{r},\mathtt{s}\} = \emptyset)\}.$$

We lift this to words, i.e. $\{(u \cdot x_1 \cdot x_2 \cdot w, u \cdot x_2 \cdot x_1 \cdot w) \mid u, w \in \Sigma^* \text{ and } (x_1, x_2) \in \mathcal{I}\}$, and obtain an equivalence relation $\equiv^{\mathcal{I}}$ as its transitive and reflexive closure. We define its closure for language $L \subseteq \Sigma^*$: $\mathcal{C}^{\equiv^{\mathcal{I}}}(L) := \{u \in \Sigma^* \mid \exists w \in L \text{ with } u \equiv^{\mathcal{I}} w\}$.

Definition 4.29 ($\mathcal{I}$-closed global types)**.** Let G be a global type G. We say G is $\mathcal{I}$-closed if $\mathcal{L}_{\text{fin}}(\mathsf{GAut}(G)) = \mathcal{C}^{\equiv^{\mathcal{I}}}(\mathcal{L}_{\text{fin}}(\mathsf{GAut}(G)))$.

Note that $\mathcal{I}$-closedness is defined on the state machine $\mathsf{GAut}(G)$ of G with alphabet Σ and not on its semantics $\mathcal{L}(G)$ with alphabet Γ.

Example 4.30. The global type $G_{2\text{BP}}$ is $\mathcal{I}$-closed. Buyer $\mathtt{a}$ is involved in every interaction. Thus, for every two consecutive interactions, there is a participant that is involved in both. ◀

We can check on the FSM of a global type if it is $\mathcal{I}$-closed.

Algorithm 4.31 (Checking if G is $\mathcal{I}$-closed)**.** Let G be a global type. We construct the state machine $\mathsf{GAut}(G)$. We need to check every consecutive occurrence of letters from Σ for words from $\mathcal{L}(\mathsf{GAut}(G))$. For binder states, incoming and outgoing transition labels are always ε. This is why we slightly modify the state machine but preserve its language. We remove all variable states and rebend their only incoming transition to the state their only outgoing transition leads to. In addition, we merge binder states with their only successor. For every state q of this modified state machine, we consider the labels $x, y \in \Sigma$ of every combination of incoming and outgoing transition of q. We check if $x \equiv_{\mathcal{I}} y$. If this is false for all x and y, we return *true*. If not, we return *false*.

Let us prove that this algorithm is correct.

Lemma 4.32. A global type G is $\mathcal{I}$-closed iff Algorithm 4.31 returns *true*.

Proof. It is obvious that the language is preserved by the changes to the state machine. (We basically turned an unambiguous state machine into a deterministic one.)

For soundness, we assume that Algorithm 4.31 returns *true* and let w be a word in $\mathcal{C}^{\equiv^{\mathcal{I}}}(\mathcal{L}(\mathsf{GAut}(G)))$. By definition, there is a run with trace w' in $\mathsf{GAut}(G)$ such that $w' \equiv_{\mathcal{I}} w$. The conditions in Algorithm 4.31 ensure that $w = w'$ because no two adjacent elements in w' can be reordered with $\equiv_{\mathcal{I}}$. Therefore, $w \in \mathcal{L}(\mathsf{GAut}(G))$ which proves the claim.

For completeness, we assume that Algorithm 4.31 returns *false* and show that there is $w \in \mathcal{C}^{\equiv^{\mathcal{I}}}(\mathcal{L}_{\text{fin}}(G))$ such that $w \notin \mathcal{L}_{\text{fin}}(\mathsf{GAut}(G))$. Without loss of generality, let q_2 be the state for which an incoming label x and outgoing label y can be reordered, i.e. $x \equiv_{\mathcal{I}} y$, and let q_1 be the state from which the transition with label x originates: $q_1 \xrightarrow{x} q_2 \in \delta_{\mathsf{GAut}(G)}$. We consider a word w' which is the trace of a maximal run that passes q_1 and q_2 and the transitions labelled with x and y. By construction, it holds that $w' \in \mathcal{L}_{\text{fin}}(\mathsf{GAut}(G))$. We swap x and y in w' to obtain w. We denote x with $\mathtt{p} \to \mathtt{q} : \mathsf{m}$ and y with $\mathtt{r} \to \mathtt{s} : \mathsf{m}'$ such that $\{\mathtt{p}, \mathtt{q}\} \cap \{\mathtt{r}, \mathtt{s}\} \neq \emptyset$. From the syntactic restrictions of global types, we know that any transition label from q_1 has sender $\mathtt{p}$ while every transition label from q_2 has sender $\mathtt{r}$. Because of this and determinacy of the state machine, there is no run in $\mathsf{GAut}(G)$ with trace w. Thus, $w \notin \mathcal{L}_{\text{fin}}(\mathsf{GAut}(G))$ which concludes the proof. □

This shows that the presented algorithm can be used to check $\mathcal{I}$-closedness. The algorithm considers every state and all combinations of transitions leading to and from it.

Proposition 4.33. For global type G, checking if G is $\mathcal{I}$-closed is in $O(|G|^2)$.

The tree-like shape of $\mathsf{GAut}(G)$ might suggest that this check can be done in linear time. However, this example shows that recursion can lead to a quadratic number of checks.

Example 4.34. Let us consider the following global type for some $n \in \mathbb{N}$:

$$\mu t \,.\, + \begin{cases} \mathtt{p} \to \mathtt{q}_0 : \mathsf{m}_0 \,.\, \mathtt{q}_0 \to \mathtt{r}_0 : \mathsf{m}_0 \,.\, \mathtt{r}_0 \to \mathtt{s}_0 : \mathsf{m}_0 \,.\, 0 \\ \mathtt{p} \to \mathtt{q}_1 : \mathsf{m}_1 \,.\, \mathtt{q}_1 \to \mathtt{r}_1 : \mathsf{m}_1 \,.\, \mathtt{r}_1 \to \mathtt{s}_1 : \mathsf{m}_1 \,.\, t \\ \quad \vdots \\ \mathtt{p} \to \mathtt{q}_n : \mathsf{m}_n \,.\, \mathtt{q}_n \to \mathtt{r}_n : \mathsf{m}_n \,.\, \mathtt{r}_n \to \mathtt{s}_n : \mathsf{m}_n \,.\, t \end{cases} .$$

It is obvious that $(\mathtt{p} \to \mathtt{q}_i : \mathsf{m}_i, \mathtt{q}_i \to \mathtt{r}_i : \mathsf{m}_i) \notin \mathcal{I}$ and $(\mathtt{q}_i \to \mathtt{r}_i : \mathsf{m}_i, \mathtt{r}_i \to \mathtt{s}_i : \mathsf{m}_i) \notin \mathcal{I}$ for every i. Because of the recursion, we need to check if $(\mathtt{r}_i \to \mathtt{s}_i : \mathsf{m}_i, \mathtt{p} \to \mathtt{q}_j : \mathsf{m}_j)$ is in $\mathcal{I}$ for every $0 \neq i \neq j$. This might lead to a quadratic number of checks. ◀

If a global type G is $\mathcal{I}$-closed, we can apply the results for its $\mathcal{I}$-closed HMSC encoding $H(G)$, for which checking implementability is in PSPACE. With Corollary 4.19, the determinised projection by erasure implements G.

Corollary 4.35. Checking implementability of 0-reachable, $\mathcal{I}$-closed global types with sender-driven choice is in PSPACE.

Example 4.36. Not every implementable global type is $\mathcal{I}$-closed – for instance, the following one: $\mathtt{p} \to \mathtt{q} : \mathsf{m} \,.\, \mathtt{r} \to \mathtt{s} : \mathsf{m} \,.\, 0$. ◀

4.3.2 Payload Implementability

A deadlock-free CSM implements a global type if their languages are precisely the same. In the HMSC literature, a variant of the implementability problem has been studied. Intuitively, it allows to add fresh data to the payload of an existing message and protocol fidelity allows to omit the additional payload data [55]. This allows to add synchronisation messages to existing interactions and can make unimplementable global types implementable while preserving the structure of the protocol. It can also be used if a global type is rejected by a projection operator or the run time of the previous algorithms is not acceptable.

Definition 4.37 (Payload implementability)**.** Let L be a language with message alphabet $\mathcal{V}_1$. We say that L is *payload-implementable* if there is a message alphabet $\mathcal{V}_2$ for a deadlock-free CSM $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that, for every $\mathtt{p} \in \mathcal{P}$, $A_\mathtt{p}$ is an FSM over $\{\mathtt{p}\triangleright\mathtt{q}!\mathsf{m}, \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \mid \mathtt{q} \in \mathcal{P} \setminus \{\mathtt{p}\},\ \mathsf{m} \in \mathcal{V}_1 \times \mathcal{V}_2\}$ such that its language is the same when projecting onto the message alphabet $\mathcal{V}_1$, i.e. $\mathcal{C}^\sim(L) = \mathcal{L}(\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}})\Downarrow_{\mathcal{V}_1}$, where $(\mathtt{p}\triangleright\mathtt{q}!(\mathsf{m}_1,\mathsf{m}_2))\Downarrow_{\mathcal{V}_1} := \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}_1$ and $(\mathtt{q}\triangleleft\mathtt{p}?(\mathsf{m}_1,\mathsf{m}_2))\Downarrow_{\mathcal{V}_1} := \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m}_1$ and is lifted to words and languages as expected.

First, payload implementability can be used if a global type is not implementable in the standard way. Second, it can be used if the algorithmic complexity for standard implementability is not tolerable.

Genest et al. [55] showed that the finite language $\mathcal{L}_{\text{fin}}(H)$ of a *local* HMSC H is always payload-implementable with a deadlock-free CSM of linear size.

Definition 4.38 (Local HMSCs [55])**.** Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We call H *local* if $\mu(v^I)$ has a unique minimal event and there is a function $\text{root}\colon V \to \mathcal{P}$ such that for every $(v, u) \in E$, it holds that $\mu(u)$ has a unique minimal event e and e belongs to $\text{root}(v)$, i.e. for $\mu(u) = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$, we have that $p(e) = \text{root}(v)$ and $e \leq e'$ for every $e' \in N$.

Proposition 4.39 (Prop. 21 [55])**.** For any local HMSC H, the language $\mathcal{L}_{\text{fin}}(H)$ is payload-implementable.

With Lemma 4.18, we can use the implementation of a local $H(G)$ for a 0-reachable global type G.

Corollary 4.40. Let G be a 0-reachable global type for which $H(G)$ is local. Then, G can be implemented with a CSM of linear size.

The algorithm to construct a deadlock-free CSM [55, Sec. 5.2] suggests that the BMSCs for such HMSCs need to be maximal – in the sense that any vertex with a single successor is collapsed with its successor. If this was not the case, the result would claim that the language of the following global type is payload-implementable:

$$\mu t \,.\, + \begin{cases} \mathtt{p} \to \mathtt{q} : \mathtt{m}_1 \,.\, \mathtt{r} \to \mathtt{s} : \mathtt{m}_2 \,.\, t \\ \mathtt{p} \to \mathtt{q} : \mathtt{m}_3 \,.\, 0 \end{cases} .$$

However, it is easy to see that it is not payload-implementable since there is no interaction between $\mathtt{p}$, which decides whether to stay in the loop or not, and $\mathtt{r}$. Thus, we cannot simply check whether $H(G)$ is local. In fact, it would always be. Instead, we first need to minimise it and then check if it is local. If we collapse the two consecutive vertices with independent pairs of participants in this example, the HMSC is not local. The representation of the HMSC matters which shows that local as property is rather a syntactic than a semantic notion.

Algorithm 4.41 (Checking if $H(G)$ is local – directly on $\mathsf{GAut}(G)$)**.** Let G be a global type. We consider the finite trace w of every longest branch-free, loop-free and non-initial run in the state machine $\mathsf{GAut}(G)$. We implicitly split the synchronous interactions into asynchronous events: $w = w_1 \ldots w_n \in \Gamma^*$. We need to check if there is $u \sim w$ with $u = u_1 \ldots u_n$ such that $u_1 \neq w_1$. For this, we can construct an MSC for w [52, Sec. 3.1] and check if there is a single minimal event. This works because MSCs are closed under $\sim$ (Lemma 2.21). If the MSC of every trace w' has a single minimal event, we return *true*. If not, we return *false*.

It is straightforward that this mimics the corresponding check for the HMSC $H(G)$ and, with similar modifications as for Algorithm 4.31, the check can be done in $O(|G|)$.

Proposition 4.42. For a global type G, Algorithm 4.41 returns *true* iff $H(G)$ is local.

Ben-Abdallah and Leue [15] introduced local-choice HMSCs, which are as expressive as local HMSCs. Their condition also uses a root-function and minimal events but quantifies over paths. Every local HMSC is a local-choice HMSC and every local-choice HMSC can be translated to a local HMSC that accepts the same language with a quadratic blow-up [55]. It is straightforward to adapt the Algorithm 4.41 to check if a global type is local-choice. If this is the case, we translate the protocol and use the implementation for the translated protocol.

4.4 Implementability with Intra-participant Reordering

In this section, we introduce a generalisation of the implementability problem that relaxes the total event order for each participant and prove that this generalisation is undecidable in general.

4.4.1 A Case for More Reordering

From the perspective of a single participant, each word in its language consists of a sequence of send and receive events. Choice in global types happens by sending (and not by receiving). Hence, we argue that a participant should be able to receive messages from different senders in any order between sending two messages. In practice, receiving a message can induce a task with non-trivial computation that our model does not account for. Therefore, such a reordering for a sequence of receive events can have outsized performance benefits. In addition, there are global types that can be implemented with regard to this generalised relation even if no (standard) implementation exists.

Example 4.43 (Example for intra-participant reordering)**.** Let us consider a global type where a central coordinator p distributes independent tasks to different participants in rounds:

$$G_{\mathrm{TC}} := \mu t\,.\, + \begin{cases} \mathsf{p} \to \mathsf{q}_1 : \mathsf{task}\,.\, \ldots\, \mathsf{p} \to \mathsf{q}_n : \mathsf{task}\,.\, \mathsf{q}_1 \to \mathsf{p} : \mathsf{result}\,.\, \ldots\, \mathsf{q}_n \to \mathsf{p} : \mathsf{result}\,.\, t \\ \mathsf{p} \to \mathsf{q}_1 : \mathsf{done}\,.\, \ldots\, \mathsf{p} \to \mathsf{q}_n : \mathsf{done}\,.\, 0 \end{cases} .$$

Since all tasks in each round are independent, p can benefit from receiving the results in the order they arrive instead of busy-waiting. ◀

We generalise the indistinguishability relation $\sim$ accordingly.

Definition 4.44 (Intra-participant indistinguishability relation $\approx$)**.** We define the *intra-participant indistinguishability relation* $\approx \subseteq \Sigma^* \times \Sigma^*$ as smallest equivalence relation such that

(a) If $w \sim u$, then $w \approx u$.

(b) If $q \neq r$, then $w \cdot p \triangleleft q ? m \cdot p \triangleleft r ? m' \cdot u \;\approx\; w \cdot p \triangleleft r ? m' \cdot p \triangleleft q ? m \cdot u$.

Define $u \preceq_{\approx} v$ if there is $w \in \Sigma^*$ such that $u \cdot w \approx v$. We observe that $u \approx v$ iff $u \preceq_{\approx} v$ and $v \preceq_{\approx} u$. Using this, we extend $\approx$ to infinite words and languages as for $\sim$.

Definition 4.45 (Implementability w.r.t. $\approx$)**.** A language $L \subseteq \Gamma^\infty$ is *implementable with regard to* $\approx$ if there exists a deadlock-free CSM $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ such that $\mathcal{C}^{\approx}(L) = \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_p\}\!\}_{p \in \mathcal{P}}))$.[1] We say that $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ $\approx$-implements L. As for $\sim$, we say that $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ implements G if $L = \mathcal{L}(G)$.

In this section, we emphasise the indistinguishability relation which we consider, e.g. $\approx$-implementable. We deliberately choose not to require $\mathcal{C}^{\approx}(\mathcal{L}(G)) = \mathcal{L}(\{\!\{A_p\}\!\}_{p \in \mathcal{P}})$, like $\sim$-implementability does, because this, in turn, requires the CSM to be closed under $\approx$. In general, this is not possible with finitely many states. In particular, if there is a loop without any send events for a participant, the labels in the loop can yield an infinite closure if we require that $\mathcal{C}^{\approx}(\mathcal{L}(G))\Downarrow_{\Gamma_p} = \mathcal{L}(A_p)$.

Example 4.46. We consider a variant of G_{TC} from Example 4.43 with $n = 2$ where q_1 and q_2 send a log-message to r after receiving the task and before sending the result back:

$$G_{\text{TCLog}} := \mu t.\; + \begin{cases} p \to q_1 : \mathsf{task}.\, p \to q_2 : \mathsf{task}.\, q_1 \to r : \mathsf{log}.\, q_2 \to r : \mathsf{log}.\, q_1 \to p : \mathsf{result}.\, q_2 \to p : \mathsf{result}.\, t \\ p \to q_1 : \mathsf{done}.\, p \to q_2 : \mathsf{done}.\, 0 \end{cases} .$$

There is no FSM for r that precisely accepts $\mathcal{C}^{\approx}(\mathcal{L}(G_{\text{TCLog}}))\Downarrow_{\Gamma_r}$. If we rely on the fact that q_1 and q_2 send the same number of log-messages to r, we can use an FSM A_r with a single state (both initial and final) with two transitions: one for the log-message from q_1 and q_2 each, that lead back to the only state. For this, it holds that $\mathcal{C}^{\approx}(\mathcal{L}(G_{\text{TCLog}}))\Downarrow_{\Gamma_r} \subseteq \mathcal{L}(A_r)$. If we cannot rely on this, the FSM would need to keep track of the difference, which can be unbounded, making the language not recognisable by an FSM. ◀

This is why we chose a more permissive definition by requiring that the $\approx$-closure of both are the same. It is trivial that any $\sim$-implementation for a global type does also $\approx$-implement it.

Proposition 4.47. Let G be a global type and $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ be a CSM. If $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ $\sim$-implements G, then $\{\!\{A_p\}\!\}_{p \in \mathcal{P}}$ also $\approx$-implements G.

[1] The original definition also included the additional condition $L \subseteq \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_p\}\!\}_{p \in \mathcal{P}}))$, which, however, is entailed by the present condition because $L \subseteq \mathcal{C}^{\approx}(L)$.

For instance, the erasure candidate implementation is a $\sim$-implementation as well as a $\approx$-implementation for the task coordination protocol G_{TC} from Example 4.43. Still, $\approx$-implementability gives more freedom and allows to consider all possible combinations of arrivals of results for the coordinator $\mathtt{p}$. In addition, $\approx$-implementability renders some global types implementable that would not be otherwise. For instance, those with a participant that would need to receive different sequences, related by $\approx$ though, in different branches it cannot distinguish (yet).

Example 4.48 ($\approx$-implementable but not $\sim$-implementable)**.** Let us consider the following global type:

$$+\begin{cases} \mathtt{p}\to\mathtt{q}:\mathtt{t}\,.\,\mathtt{p}\to\mathtt{r}:\mathtt{m}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{m}\,.\,0 \\ \mathtt{p}\to\mathtt{q}:\mathtt{b}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{m}\,.\,\mathtt{p}\to\mathtt{r}:\mathtt{m}\,.\,0 \end{cases} .$$

This cannot be $\sim$-implemented because $\mathtt{r}$ would need to know about the choice to receive the messages from $\mathtt{p}$ and $\mathtt{q}$ in the correct order. However, it is $\approx$-implementable. The FSMs for $\mathtt{p}$ and $\mathtt{q}$ can be obtained by determinising the projection by erasure. For $\mathtt{r}$, we can have an FSM that only accepts $\mathtt{r}\triangleleft\mathtt{p}?\mathtt{m}\cdot\mathtt{r}\triangleleft\mathtt{q}?\mathtt{m}$ but also an FSM which accepts $\mathtt{r}\triangleleft\mathtt{q}?\mathtt{m}\cdot\mathtt{r}\triangleleft\mathtt{p}?\mathtt{m}$ in addition. Note that $\mathtt{r}$ does not learn the choice in the second FSM even if it branches. Hence, it would not be implementable if it sent different messages in both branches later on. However, it could still learn by receiving and, afterwards, send different messages. ◀

4.4.2 Undecidability

Unfortunately, checking implementability with regard to $\approx$ for global types (even with directed choice) is undecidable. Intuitively, the reordering allows participants to drift arbitrarily far apart as the execution progresses which makes it hard to learn which choices were made.

We reduce the *Post Correspondence Problem* (PCP) [105] to the problem of checking implementability with regard to $\approx$. An instance of PCP over an alphabet Δ with $|\Delta| > 1$ is given by two finite lists $(u_1, u_2, \dots, u_n)$ and $(v_1, v_2, \dots, v_n)$ of finite words over Δ, also called tile sets. A solution to the instance is a sequence of indices $(i_j)_{1\leq j\leq k}$ with $k \geq 1$ and $1 \leq i_j \leq n$ for all $1 \leq j \leq k$, such that $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$. To be precise, we present a reduction from the modified PCP (MPCP) [112, Sec. 5.2], which is also undecidable. It simply requires that a match starts with a specific pair – in our case we choose the pair with index 1. It is possible to directly reduce from PCP but the reduction from MPCP is more concise. Intuitively, we require the solution to start with the first pair so there exists no trivial solution and, using MPCP, we can choose a single pair, which is more concise than all possible ones.

Theorem 4.49. Checking implementability with regard to $\approx$ for 0-reachable global types with directed choice is undecidable.

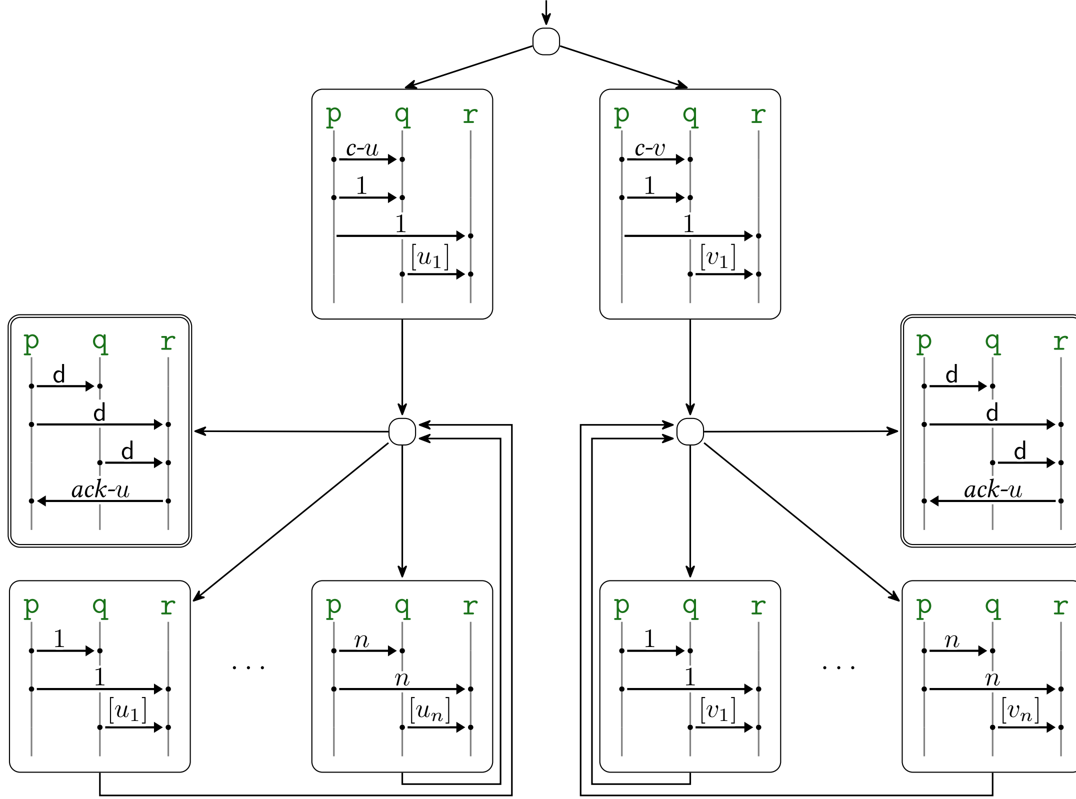


Figure 4.3: HMSC encoding $H(G_{\text{MPCP}})$ of the MPCP encoding.

Proof. Let $\{(u_1, u_2, \ldots, u_n), (v_1, v_2, \ldots, v_n)\}$ be an instance of MPCP over alphabet Δ where 1 is the special index which each solution needs to start with. We use u and v as identifiers for both sets. For a word $w = a_1 a_2 \cdots a_m \in \Delta^*$, a message labelled $[w]$ denotes a sequence of individual message interactions with message $a_1, a_2, \ldots, a_m$, each of size 1. We define a parametric global type for which $x \in \{u, v\}$:

$$G(x, X) \quad := \quad \mathtt{p}\to\mathtt{q}:c\text{-}x.\,\mathtt{p}\to\mathtt{q}:1.\,\mathtt{p}\to\mathtt{r}:1.\,\mathtt{q}\to\mathtt{r}:[x_1].\,\mu t.\,+\begin{cases}\mathtt{p}\to\mathtt{q}:1.\,\mathtt{p}\to\mathtt{r}:1.\,\mathtt{q}\to\mathtt{r}:[x_1].\,t\\ \ldots\\ \mathtt{p}\to\mathtt{q}:n.\,\mathtt{p}\to\mathtt{r}:n.\,\mathtt{q}\to\mathtt{r}:[x_n].\,t\\ \mathtt{p}\to\mathtt{q}:\mathsf{d}.\,\mathtt{p}\to\mathtt{r}:\mathsf{d}.\,\mathtt{q}\to\mathtt{r}:\mathsf{d}.\,X\end{cases}$$

where c-x indicates *choosing* tile set x. Using this, we obtain our encoding:

$$G_{\text{MPCP}} \quad := \quad +\begin{cases}G(u, \mathtt{r}\to\mathtt{p}:\textit{ack-u}.\,0)\\ G(v, \mathtt{r}\to\mathtt{p}:\textit{ack-v}.\,0)\end{cases}.$$

Figure 4.3 illustrates its HMSC encoding $H(G_{\text{MPCP}})$.

It suffices to show the following equivalences:

$$\begin{aligned} & G_{\text{MPCP}} \text{ is } \approx\text{-implementable} \\ \Leftrightarrow_1 \quad & \mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Gamma_{\mathtt{r}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Gamma_{\mathtt{r}}} = \emptyset \\ \Leftrightarrow_2 \quad & \text{MPCP instance has no solution} \end{aligned}$$

We prove $\Rightarrow_1$ by contraposition. Let $w \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Gamma_{\mathtt{r}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Gamma_{\mathtt{r}}}$. For $x \in \{u, v\}$, let $w_x \in \mathcal{C}^{\approx}(\mathcal{L}(G(x,0)))$ such that $w_x\Downarrow_{\Gamma_{\mathtt{r}}} = w$. By construction of G_{MPCP}, we know that $w_x \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}x \cdot \mathtt{p}\triangleleft\mathtt{r}?\mathit{ack}\text{-}x \in \mathcal{C}^{\approx}(\mathcal{L}(G_{\text{MPCP}}))$.

Suppose that a CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ $\approx$-implements G_{MPCP}. Then, it holds that

$$w_x \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}x \cdot \mathtt{p}\triangleleft\mathtt{r}?\mathit{ack}\text{-}x \in \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$$

by Definition 4.45. We also know that $w_x \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}y \cdot \mathtt{p}\triangleleft\mathtt{r}?\mathit{ack}\text{-}y \notin \mathcal{C}^{\approx}(\mathcal{L}(G_{\text{MPCP}}))$ for $x \neq y$ where $x, y \in \{u, v\}$. By the choice of w_u and w_v, it holds that $w_u\Downarrow_{\Gamma_{\mathtt{r}}} = w = w_v\Downarrow_{\Gamma_{\mathtt{r}}}$. Therefore, $\mathtt{r}$ needs to be in the same state of $A_{\mathtt{r}}$ after processing $w_u\Downarrow_{\Gamma_{\mathtt{r}}}$ or $w_v\Downarrow_{\Gamma_{\mathtt{r}}}$. Thus, there are three possibilities what to send to $\mathtt{p}$: $\mathtt{r}$ can either send both $\mathit{ack}\text{-}u$ and $\mathit{ack}\text{-}v$, only one of them, or none of them. Thus, either one of the following is true:

a) (sending both) $w_x \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}y \in \text{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$ for $x \neq y$ with $x, y \in \{u, v\}$, or
b) (sending u without loss of generality) $w_v \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}u \notin \text{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$, or
c) (sending none) $w_x \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}x \notin \text{pref}(\mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})))$ for $x \in \{u, v\}$.

All cases lead to deadlocks in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$. For a) and b): if $c\text{-}v$ was chosen in the beginning, $\mathtt{p}$ cannot receive the sent message as it disagrees with its choice from the beginning $c\text{-}x$. In all other cases, $\mathtt{p}$ waits for a message while no message will ever be sent. The possibility of deadlocks in $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ contradicts our assumption that it $\approx$-implements G_{MPCP} (and there cannot be any CSM that $\approx$-implements G_{MPCP}).

We prove $\Leftarrow_1$ next. The language $\mathcal{C}^{\approx}(\mathcal{L}(G_{\text{MPCP}}))$ is obviously non-empty. Therefore, let $w' \in \mathcal{C}^{\approx}(\mathcal{L}(G_{\text{MPCP}}))$. We split w' to obtain:

$$w' = w \cdot \mathtt{r}\triangleright\mathtt{p}!\mathit{ack}\text{-}x \cdot \mathtt{p}\triangleleft\mathtt{r}?\mathit{ack}\text{-}x \text{ for some } w \text{ and } x \in \{u, v\} \; .$$

By construction of G_{MPCP}, we know that

$$w \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0))) \;\cup\; \mathcal{C}^{\approx}(\mathcal{L}(G(v,0))).$$

By assumption, the intersection of both sets is empty. Thus, exactly one of the following holds:

$$w\Downarrow_{\Gamma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0)))\Downarrow_{\Gamma_{\mathtt{r}}} \quad \text{or} \quad w\Downarrow_{\Gamma_{\mathtt{r}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(v,0)))\Downarrow_{\Gamma_{\mathtt{r}}}.$$

We give a $\approx$-implementation for G_{MPCP}. It is straightforward to construct FSMs for both p and q. They are involved in the initial decision and $\approx$ does not affect their projected languages. Thus, determinising the projection by erasure yields the FSMs $A_{\mathtt{p}}$ and $A_{\mathtt{q}}$. We construct an FSM $A_{\mathtt{r}}$ for r with control state $\mathtt{i} \in \{1, \dots, n\}$, $\mathtt{j} \in \{1, \dots, \max(|u_i| \mid i \in \{1, \dots, n\})\}$, $\mathtt{d} \in \{0, 1, 2\}$, and $\mathtt{x} \in \{u, v\}$, where $|w|$ denotes the length of a word. The FSM is constructed in a way such that

$$w{\Downarrow_{\Gamma_{\mathtt{r}}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(u, 0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \quad \text{if and only if} \quad \mathtt{d} \text{ is } 2 \text{ and } \mathtt{x} \text{ is } u \qquad \text{as well as}$$
$$w{\Downarrow_{\Gamma_{\mathtt{r}}}} \in \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \quad \text{if and only if} \quad \mathtt{d} \text{ is } 2 \text{ and } \mathtt{x} \text{ is } v.$$

Let us first explain that this characterisation suffices to show that $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p} \in \mathcal{P}}$ $\approx$-implements G_{MPCP}. The control state d counts the number of received d-messages. Thus, there will be no more messages to r in any channel once d is 2 by construction of G_{MPCP}. Once in a state for which d is 2, r sends message *ack-u* to p if x is u and message *ack-v* if x is v. With the earlier characterisation, this message *ack-x* matches the message *c-x* sent from p to q in the beginning and, thus, p will be able to receive it and conclude the execution.

Now, we will explain how to construct the FSM $A_{\mathtt{r}}$. Intuitively, r keeps track of a tile number, which it tries to match against, and stores this in i. It is initially set to 0 to indicate no tile has been chosen yet. The index j denotes the position of the letter it needs to match in tile $u_{\mathtt{i}}$ next and, thus, is initialised to 1. The variable d indicates the number of d-messages received so far, so initially d is 0. With this, r knows when it needs to send *ack-x*. The FSM for r tries to match the received messages against the tiles of u, so x is initialised to u. If this matching fails at some point, x is set to v as it learned that v was chosen initially by p.

In any of the following cases, if a received message is a d-message, d is solely increased by 1:

- If x is u and i is 0, r receives a message z from p and sets i to z (technically the integer represented by z).
- If x is u and i is not 0, r receives a message z from q.
  - If z is the same as $u_{\mathtt{i}}[\mathtt{j}]$, we increment j by 1 and check if $\mathtt{j} > |u_{\mathtt{i}}|$ and, if so, set i to 0 and j to 1.
  - If not, we set x to v.
- Once x is v, r can simply receive all remaining messages in any order.

The described FSM can be used for r because it reliably checks whether a presented sequence of indices and words belongs to tile set u or v. It can do so because $\mathcal{C}^{\approx}(\mathcal{L}(G(u, 0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v, 0))){\Downarrow_{\Gamma_{\mathtt{r}}}} = \emptyset$ by assumption.

Note that this construction of $A_{\mathtt{r}}$ accepts words that are not in $\mathcal{C}^{\sim}(G_{\text{MPCP}}){\Downarrow_{\Gamma_{\mathtt{r}}}}$. This is fine because we can rely on $\mathtt{p}$ and $\mathtt{q}$ sending genuine messages. There is no need to but we could enforce not to accept such words by checking against both tile sets simultaneously and only stop checking once there is a mismatch. By our assumption, there will always be one valid tile set at the end.

Next, we prove $\Rightarrow_2$ by contraposition. Suppose the MPCP instance has a solution. Let $i_1, \ldots, i_k$ be a non-empty sequence of indices such that $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$ and $i_1 = 1$. It is easy to see that

$$w_x := \mathtt{r}\triangleleft\mathtt{p}?i_1 \cdot \mathtt{r}\triangleleft\mathtt{q}?[x_{i_1}] \cdots \mathtt{r}\triangleleft\mathtt{p}?i_k \cdot \mathtt{r}\triangleleft\mathtt{q}?[x_{i_k}] \cdot \mathtt{r}\triangleleft\mathtt{p}?\mathsf{d} \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathsf{d} \in \mathcal{L}(G(x,0)){\Downarrow_{\Gamma_{\mathtt{r}}}} \text{ for } x \in \{u, v\}.$$

By definition of $\approx$, we can re-arrange the previous sequences such that

$$\mathtt{r}\triangleleft\mathtt{p}?i_1 \cdots \mathtt{r}\triangleleft\mathtt{p}?i_k \cdot \mathtt{r}\triangleleft\mathtt{q}?[x_{i_1}] \cdots \mathtt{r}\triangleleft\mathtt{q}?[x_{i_k}] \cdot \mathtt{r}\triangleleft\mathtt{p}?\mathsf{d} \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathsf{d} \in \mathcal{C}^{\approx}(\mathcal{L}(G(x,0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \text{ for } x \in \{u, v\}.$$

Because $i_1, \ldots, i_k$ is a solution to the instance of MPCP, it holds that

$$\mathtt{r}\triangleleft\mathtt{q}?[u_{i_1}] \cdots \mathtt{r}\triangleleft\mathtt{q}?[u_{i_k}] = \mathtt{r}\triangleleft\mathtt{q}?[v_{i_1}] \cdots \mathtt{r}\triangleleft\mathtt{q}?[v_{i_k}] \quad \text{and, thus,}$$

$$\mathtt{r}\triangleleft\mathtt{p}?i_1 \cdots \mathtt{r}\triangleleft\mathtt{p}?i_k \cdot \mathtt{r}\triangleleft\mathtt{q}?[u_{i_1}] \cdots \mathtt{r}\triangleleft\mathtt{q}?[u_{i_k}] \cdot \mathtt{r}\triangleleft\mathtt{p}?\mathsf{d} \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathsf{d} \text{ is in } \mathcal{C}^{\approx}(\mathcal{L}(G(v,0))){\Downarrow_{\Gamma_{\mathtt{r}}}}.$$

This shows that $\mathcal{C}^{\approx}(\mathcal{L}(G(u,0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \cap \mathcal{C}^{\approx}(\mathcal{L}(G(v,0))){\Downarrow_{\Gamma_{\mathtt{r}}}} \neq \emptyset$.

Lastly, we prove $\Leftarrow_2$. We know that the MPCP instance has no solution. Thus, there cannot be a non-empty sequence of indices $i_1, i_2, \ldots, i_k$ such that $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$ and $i_1 = 1$. We consider any possible words $w_u \in \mathcal{C}^{\approx}(\mathcal{L}(G(u,0))){\Downarrow_{\Gamma_{\mathtt{r}}}}$ and $w_v \in \mathcal{C}^{\approx}(\mathcal{L}(G(v,0))){\Downarrow_{\Gamma_{\mathtt{r}}}}$ and try to match them to find an element in the intersection of both sets. To be precise, we consider the sequence of receive events $w_x{\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?_}}$ with sender $\mathtt{p}$ and the sequence of receive events $w_x{\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?_}}$ with sender $\mathtt{q}$ for $x \in \{u, v\}$. The intra-participant indistinguishability relation $\approx$ allows to reorder events of both senders. Thus, the intersection is non-empty if and only if we can find words w_u and w_v such that $w_u{\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?_}} = w_v{\Downarrow_{\mathtt{r}\triangleleft\mathtt{p}?_}}$ and $w_u{\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?_}} = w_v{\Downarrow_{\mathtt{r}\triangleleft\mathtt{q}?_}}$. However, $G(x,0)$ for $x \in \{u, v\}$ is constructed in a way that this is only possible if the MPCP instance has a solution. Therefore, the intersection is empty which proves our claim. □

This result carries over to HMSCs.

Definition 4.50. An HMSC H is said to be *implementable with regard to* $\approx$ if there exists a deadlock-free CSM $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that the following holds: (i) $\mathcal{L}(H) \subseteq \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$ and (ii) $\mathcal{C}^{\approx}(\mathcal{L}(H)) = \mathcal{C}^{\approx}(\mathcal{L}(\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}))$.

Corollary 4.51. Checking implementability w.r.t. $\approx$ for HMSCs is undecidable.

It is obvious that a final vertex is reachable from every vertex in $H(G_{\text{MPCP}})$, making G_{MPCP} 0-reachable. Also, because of the done-messages, if G_{MPCP} is implementable, its implementation is sink-final for every participant. The protocol also satisfies a number of channel restrictions. Our results in Chapter 7 show that it is existentially 1-bounded, 1-synchronisable and half-duplex.

The MPCP encoding only works since receive events can be reordered unboundedly in an execution. If we amended the definition of $\approx$ to give each receive event a budget that depletes with every reordering, this encoding would not be possible. Alternatively, one could require every active participant in a loop to send at least once. This also prevents such unbounded reorderings. For such restrictions on the considered indistinguishability relation, the corresponding implementability problem likely becomes decidable. We leave a detailed analysis for future work.

Chapter 5

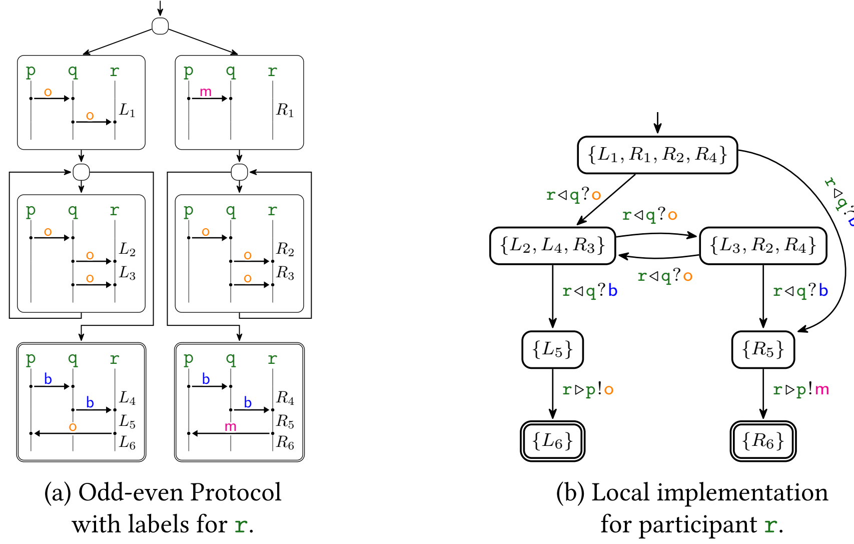
Direct and Complete Projection for Multiparty Session Types

In the previous chapter, we developed a decision procedure for the implementability problem of sender-driven global types. This is of theoretical interest but does not lend itself to an efficient implementation, e.g. in a prototype tool. In this chapter, we develop the first direct, sound and complete MST projection operator. We do so for a slightly more general class of protocols: we still assume sender-driven choice but do not require that every partial execution of a protocol can always be extended to a finite completed execution. Our projection operator separates constructing a candidate implementation from checking implementability. It uses textbook automata-theoretic constructions and generates annotations for the resulting automata, which are subsequently used to check implementability of the global type. We prove soundness and completeness of our PSPACE approach. We also present a family of global types for which generating an implementation requires exponential time in the size of the input. Our approach can also be used to solve the soft implementability problem, which employs a slightly different notion of deadlocks. With a prototype implementation, we demonstrate the practicality of our approach. Last, we survey properties of global types and their implementations from the literature and show that our notion of implementability entails most of them or can easily be checked.

5.1 Constructing Implementations

The synthesis step in our projection operator uses textbook automata-theoretic constructions. From a given global type, we derive a finite state machine, and use it to define a homomorphism automaton for each participant. We then determinise this homomorphism automaton via subset construction to obtain a local candidate implementation for each participant. If the global type is implementable, this construction always yields an implementation.

Let us first explain the idea of our construction with an example.



(a) Odd-even Protocol with labels for r.

(b) Local implementation for participant r.

Figure 5.1: Odd-even revisited: as HMSC with labels for r and the subset construction for r.

Example 5.1 (Odd-even revisited)**.** Let us revisit the odd-even example from Example 3.43. Figure 5.1a illustrates its HMSC encoding, with labels for r. Recall that no classical projection operator can generate the local implementation for participant r. Our approach applies a projection by erasure and determinises the result, obtaining the FSM in Fig. 5.1b for r. Intuitively, such a deterministic local candidate implementation can follow multiple runs in the global type simultaneously. We use the labels $L_1, \ldots, L_6, R_1, \ldots, R_6$ in Figs. 3.11 and 3.12c to indicate where r might be in the global type. For readability, we only use the labels for r's trajectory but our construction keeps track of all possible (global) points in the protocol. ◀

Formally, the construction is carried out in two steps. First, analogous to the projection by erasure for HMSCs, for each participant p, we define an intermediate state machine $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}}$ that is a homomorphism of $\mathsf{GAut}(G)$. We call $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}}$ the *projection by erasure* for p. Note that the projection by erasure for HMSCs already applies the subset construction while the one for global types does not.

Definition 5.2 (Projection by erasure)**.** Let G be some global type with its state machine $\mathsf{GAut}(G) = (Q_G, \Sigma, \delta_G, q_{0,G}, F_G)$. For each participant $\mathtt{p} \in \mathcal{P}$, we define the state machine $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}} = (Q_G, \Gamma_{\mathtt{p}} \uplus \{\varepsilon\}, \delta_{\downarrow}, q_{0,G}, F_G)$ with $\delta_{\downarrow} := \{q \xrightarrow{a\Downarrow_{\Gamma_{\mathtt{p}}}} q' \mid q \xrightarrow{a} q' \in \delta_G\}$. It holds that $a\Downarrow_{\Gamma_{\mathtt{p}}} \in \Gamma_{\mathtt{p}} \uplus \{\varepsilon\}$.

Then, we determinise $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}}$ via a standard subset construction to obtain a deterministic local state machine for p.

Definition 5.3 (Subset construction)**.** Let G be a global type and $\mathtt{p}$ be a participant. Then, the *subset construction* for $\mathtt{p}$ is defined as

$$\mathscr{C}(G,\mathtt{p}) = \big(Q_{\mathtt{p}}, \Gamma_{\mathtt{p}}, \delta_{\mathtt{p}}, s_{0,\mathtt{p}}, F_{\mathtt{p}}\big) \text{ where}$$

- $\delta(s,a) := \{q' \in Q_G \mid \exists q \in s, q \xrightarrow{a}\xrightarrow{\varepsilon}{}^{*} q' \in \delta_{\downarrow}\}$ for every $s \subseteq Q_G$ and $a \in \Gamma_{\mathtt{p}}$,
- $s_{0,\mathtt{p}} := \{q \in Q_G \mid q_{0,G} \xrightarrow{\varepsilon}{}^{*} q \in \delta_{\downarrow}\}$,
- $Q_{\mathtt{p}} := \mathrm{lfp}^{\subseteq}_{\{s_{0,\mathtt{p}}\}} \lambda Q.\, Q \cup \{\delta(s,a) \mid s \in Q \wedge a \in \Gamma_{\mathtt{p}}\} \setminus \{\emptyset\}$,
- $\delta_{\mathtt{p}} := \delta|_{Q_{\mathtt{p}} \times \Gamma_{\mathtt{p}}}$, and
- $F_{\mathtt{p}} := \{s \in Q_{\mathtt{p}} \mid s \cap F_G \neq \emptyset\}$.

In the above definition, the least fixed point operator lfp is solely used to restrict the state space to the reachable ones. This ensures that our construction does only contain reachable states and, thus, our validity conditions are only checked for these.

Note that the construction ensures that $Q_{\mathtt{p}}$ only contains subsets of Q_G whose states are reachable via the same traces, i.e. we typically have $|Q_{\mathtt{p}}| \ll 2^{|Q_G|}$.

The following characterisation is immediate from the subset construction; the proof can be found in Appendix A.1.

Lemma 5.4. *Let G be a global type, $\mathtt{r}$ be a participant, and $\mathscr{C}(G,\mathtt{r})$ be its subset construction. If w is a trace of $\mathsf{GAut}(G)$, $w{\Downarrow_{\Gamma_{\mathtt{r}}}}$ is a trace of $\mathscr{C}(G,\mathtt{r})$. If u is a trace of $\mathscr{C}(G,\mathtt{r})$, there is a trace w of $\mathsf{GAut}(G)$ such that $w{\Downarrow_{\Gamma_{\mathtt{r}}}} = u$. Then, it holds that $\mathcal{L}(G){\Downarrow_{\Gamma_{\mathtt{r}}}} = \mathcal{L}(\mathscr{C}(G,\mathtt{r}))$.*

With this lemma, we show the CSM $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ preserves all behaviours of G. This result is similar to Lemma 3.21 for our generalised classical projection operator.

Lemma 5.5. *For all global types G, it holds that $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$.*

Again, we defer the proof to Appendix A.1. We briefly sketch the proof here though. Given that $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deterministic, to prove language inclusion it suffices to prove the inclusion of the respective prefix sets:

$$\mathrm{pref}(\mathcal{L}(G)) \subseteq \mathrm{pref}(\mathcal{L}\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$$

Let w be a word in $\mathcal{L}(G)$. If w is finite, membership in $\mathcal{L}(\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$ is immediate from the claim above. If w is infinite, we show that w has an infinite run in $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ using König's Lemma. We construct an infinite graph $\mathcal{G}_w(V,E)$ with $V := \{v_\rho \mid \mathrm{trace}(\rho) \leq w\}$ and $E := \{(v_{\rho_1}, v_{\rho_2}) \mid \exists\, x \in \Gamma.\ \mathrm{trace}(\rho_2) = \mathrm{trace}(\rho_1) \cdot x\}$. Because $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deterministic, $\mathcal{G}_w$ is a tree rooted at v_ε, the vertex corresponding to the empty run. By König's Lemma, every infinite tree contains either a vertex of infinite degree or an infinite path. Because $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ consists of a finite number of FSM, the last configuration of any run has a finite number of next configurations, and $\mathcal{G}_w$ is finitely branching. Therefore, there must exist an infinite path in $\mathcal{G}_w$ representing an infinite run for w, and thus $w \in \mathcal{L}(\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$.

The proof of the inclusion of prefix sets proceeds by structural induction and primarily relies on Lemma 5.4 and the fact that all prefixes in $\mathcal{L}(G)$ respect the order of send before receive events.

5.2 Checking Implementability

We now turn our attention to checking implementability of a CSM produced by the subset construction. We present two conditions: *send* and *receive validity*. For both, we present a pair of examples that showcases key differences between implementable and unimplementable global types. From these, we distil our conditions that precisely characterise the implementability of global types.

In general, a global type G is not implementable when the agreement on a global run of $\mathsf{GAut}(G)$ among all participants cannot be conveyed via sending and receiving the specified messages alone. When this happens, participants can take locally permitted transitions that commit to incompatible global runs, resulting in a trace that is not specified by G. Consequently, our conditions need to ensure that when a participant $\mathtt{p}$ takes a transition in $\mathscr{C}(G, \mathtt{p})$, it only commits to global runs that are consistent with the local views of all other participants. We discuss the relevant conditions imposed on send and receive transitions separately.

5.2.1 Send Validity

Example 5.6. Consider the following global types, also depicted in Figs. 5.2a and 5.2c:

$$G_s := + \begin{cases} \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathtt{o}\,.\,\mathtt{r}\!\to\!\mathtt{q}\!:\!\mathtt{o}\,.\,0 \\ \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathtt{m}\,.\,\mathtt{r}\!\to\!\mathtt{q}\!:\!\mathtt{m}\,.\,0 \end{cases} \qquad G'_s := + \begin{cases} \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathtt{o}\,.\,\mathtt{r}\!\to\!\mathtt{q}\!:\!\mathtt{b}\,.\,0 \\ \mathtt{p}\!\to\!\mathtt{q}\!:\!\mathtt{m}\,.\,\mathtt{r}\!\to\!\mathtt{q}\!:\!\mathtt{b}\,.\,0 \end{cases}$$

In both examples, $\mathtt{p}$ chooses a branch by sending either $\mathtt{o}$ or $\mathtt{m}$ to $\mathtt{q}$, which also determines the expected behaviour of participant $\mathtt{r}$. For both G_s and G'_s, participant $\mathtt{r}$ cannot learn the branch chosen by $\mathtt{p}$. Thus, for G_s, participant $\mathtt{r}$ cannot know whether to send $\mathtt{o}$ or $\mathtt{m}$ to $\mathtt{q}$, leading inevitably to deadlocking runs. In contrast, G'_s is implementable because the expected behaviour of $\mathtt{r}$ is independent of the choice by $\mathtt{p}$.

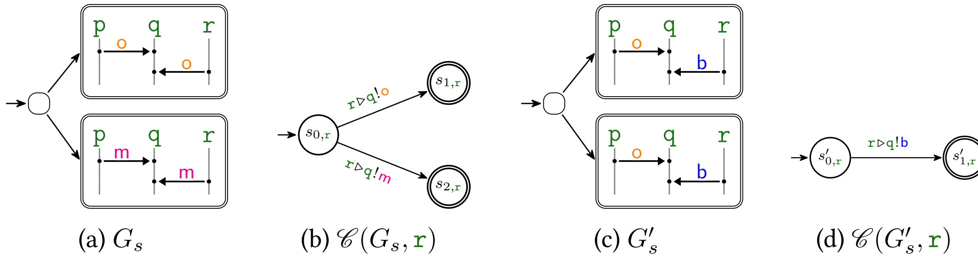


Figure 5.2: HMSCs for G_s and G'_s and subset constructions onto $\mathtt{r}$ for Example 5.6.

Let us see how we can distinguish both scenarios in the subset construction.

We first consider $\{\!\{\mathscr{C}(G_s, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for which we depict the FSM for $\mathtt{r}$ in Figure 5.2b. It has an execution with the trace $\mathtt{p}\triangleright\mathtt{q}!\mathtt{o}\cdot\mathtt{q}\triangleleft\mathtt{p}?\mathtt{o}\cdot\mathtt{r}\triangleright\mathtt{q}!\mathtt{m}$. This trace is possible because the initial state of $\mathscr{C}(G_s, \mathtt{r})$, $s_{0,\mathtt{r}}$, contains two states of $\mathsf{GAut}(G_s)\!\downarrow_{\mathtt{r}}$, each of which has a single outgoing send transition labelled with $\mathtt{r}\triangleright\mathtt{q}!\mathtt{o}$ and $\mathtt{r}\triangleright\mathtt{q}!\mathtt{m}$ respectively. Both of these transitions are always enabled in $s_{0,\mathtt{r}}$, meaning that $\mathtt{r}$ can send $\mathtt{r}\triangleright\mathtt{q}!\mathtt{m}$ even when $\mathtt{p}$ has chosen the top branch and $\mathtt{q}$ expects to receive $\mathtt{o}$ instead of $\mathtt{m}$ from $\mathtt{r}$. However, the run will deadlock because, after receiving $\mathtt{o}$ from $\mathtt{p}$, the machine $\mathscr{C}(G_s, \mathtt{q})$ is in a state that can only receive $\mathtt{o}$ from $\mathtt{r}$. Intuitively, the violation arises because for the two traces where $\mathtt{p}$ chooses to send $\mathtt{o}$ or $\mathtt{m}$ to $\mathtt{q}$, participant $\mathtt{r}$'s local view does not contain enough information to tell the two scenarios apart.

In contrast, while the state $s'_{0,\mathtt{r}}$ in $\mathscr{C}(G'_s, \mathtt{r})$ (Fig. 5.2d) also contains two states of $\mathsf{GAut}(G'_s)\!\downarrow_{\mathtt{r}}$, each with a single outgoing send transition, both these transitions are labelled with $\mathtt{r}\triangleright\mathtt{q}!\mathtt{b}$. These two transitions collapse to a single one in $\mathscr{C}(G'_s, \mathtt{r})$. This transition is consistent with both possible local views that $\mathtt{p}$ and $\mathtt{q}$ might hold on the global run. ◀

Intuitively, to prevent the emergence of inconsistent local views from send transitions of $\mathscr{C}(G, \mathtt{p})$, we must enforce that for every state $s \in Q_{\mathtt{p}}$ with an outgoing send transition labelled x, a transition labelled x must be enabled in all states of $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}}$ represented by s. We use the following auxiliary definition to formalise this intuition subsequently.

Definition 5.7 (Transition origin). Let $s \xrightarrow{x} s' \in \delta_{\mathtt{p}}$ be a transition in $\mathscr{C}(G, \mathtt{p})$ and $\delta_{\downarrow}$ be the transition relation of $\mathsf{GAut}(G)\!\downarrow_{\mathtt{p}}$. We define the set of *transition origins* $\text{tr-orig}(s \xrightarrow{x} s')$ as follows:

$$\text{tr-orig}(s \xrightarrow{x} s') := \{G \in s \mid \exists G' \in s'.\, G \xrightarrow{x}{}^{*} G' \in \delta_{\downarrow}\} \ .$$

Our condition on send transitions is then stated below.

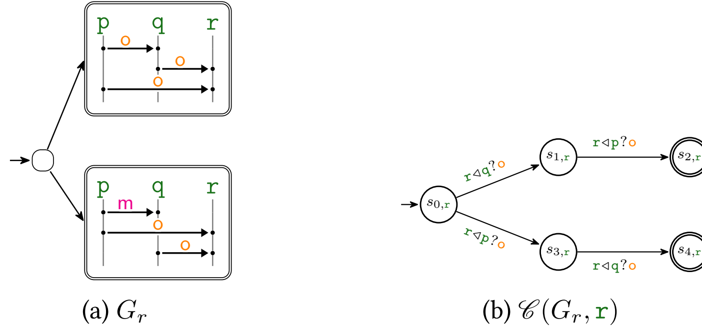
Definition 5.8 (Send Validity). $\mathscr{C}(G, \mathtt{p})$ satisfies *Send Validity* iff every send transition $s \xrightarrow{x} s' \in \delta_{\mathtt{p}}$ is enabled in all states contained in s:

$$\forall s \xrightarrow{x} s' \in \delta_{\mathtt{p}}.\, x \in \Gamma_{\mathtt{p},!} \implies \text{tr-orig}(s \xrightarrow{x} s') = s \ .$$

5.2.2 Receive Validity

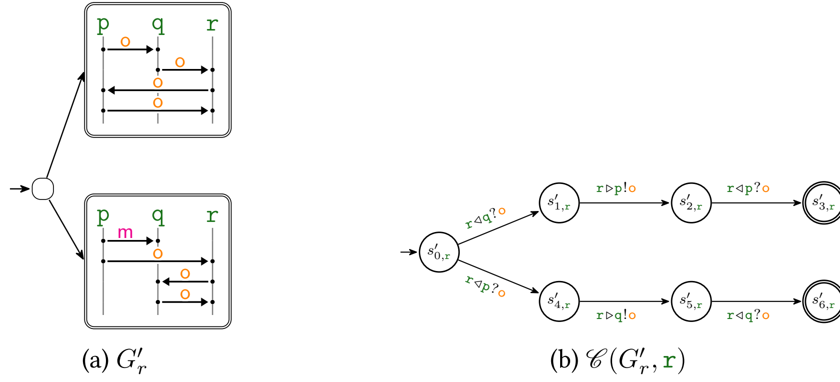
Example 5.9. Consider the following global types, also depicted in Figs. 5.3a and 5.4a:

$$G_r := + \begin{cases} \mathtt{p}\to\mathtt{q}:\mathtt{o}.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}.\,\mathtt{p}\to\mathtt{r}:\mathtt{o}.\,0 \\ \mathtt{p}\to\mathtt{q}:\mathtt{m}.\,\mathtt{p}\to\mathtt{r}:\mathtt{o}.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}.\,0 \end{cases} \qquad G'_r := + \begin{cases} \mathtt{p}\to\mathtt{q}:\mathtt{o}.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}.\,\mathtt{r}\to\mathtt{p}:\mathtt{o}.\,\mathtt{p}\to\mathtt{r}:\mathtt{o}.\,0 \\ \mathtt{p}\to\mathtt{q}:\mathtt{m}.\,\mathtt{p}\to\mathtt{r}:\mathtt{o}.\,\mathtt{r}\to\mathtt{q}:\mathtt{o}.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}.\,0 \end{cases}$$



(a) G_r (b) $\mathscr{C}(G_r, \mathtt{r})$

Figure 5.3: HMSC for G_r and its subset construction onto r.



(a) G'_r (b) $\mathscr{C}(G'_r, \mathtt{r})$

Figure 5.4: HMSC for G'_r and its subset construction onto r.

As before, p chooses a branch by sending either o or m to q, determining the expected behaviour of participant r. The global type G_r is not implementable because r cannot learn which branch was chosen by p. For any local implementation of r to be able to execute both branches, it must be able to receive o from p and q in any order. Because the two send events p▷r!o and q▷r!o are independent of each other, they may be reordered. Consequently, any implementation of G_r would have to permit executions that are consistent with global behaviours not described by G_r, such as p→q:m · q→r:o · p→r:o. Contrast this with G'_r, which is implementable. In the top branch of G'_r, participant p can only send to r after it has received from r, which prevents the reordering of the send events p▷r!o and q▷r!o. The bottom branch is symmetric. Hence, r learns the choice of p based on which message it receives first.

Let us again see how we can detect this with the subset construction. We first consider $\{\!\{\mathscr{C}(G_r, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, for which Figure 5.3b depicts the FSM for r. It recognises the following trace, which is not in the global type language $\mathcal{L}(G_r)$:

$$\mathtt{p}\triangleright\mathtt{q}!\mathtt{o} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathtt{o} \cdot \mathtt{q}\triangleright\mathtt{r}!\mathtt{o} \cdot \mathtt{p}\triangleright\mathtt{r}!\mathtt{o} \cdot \mathtt{r}\triangleleft\mathtt{p}?\mathtt{o} \cdot \mathtt{r}\triangleleft\mathtt{q}?\mathtt{o} \ .$$

The issue lies with r which cannot distinguish between the two branches in G_r. The initial state $s_{0,\mathtt{r}}$ of $\mathscr{C}(G_r, \mathtt{r})$ has two states of $\mathsf{GAut}(G_r)$ corresponding to the subterms $G_t := \mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,\mathtt{p}\to\mathtt{r}:\mathtt{o}\,.\,0$ and $G_b := \mathtt{p}\to\mathtt{r}:\mathtt{o}\,.\,\mathtt{q}\to\mathtt{r}:\mathtt{o}\,.\,0$. Here, G_t and G_b are the top and bottom branch of G_r respectively. This means that there are outgoing transitions

in $s_{0,\mathtt{r}}$ labelled with $\mathtt{r}\triangleleft\mathtt{p}?\mathtt{o}$ and $\mathtt{r}\triangleleft\mathtt{q}?\mathtt{o}$. If $\mathtt{r}$ takes the transition labelled with $\mathtt{r}\triangleleft\mathtt{p}?\mathtt{o}$, it commits to the bottom branch G_b. However, observe that the message $\mathtt{o}$ from $\mathtt{p}$ can also be available at this time point if the other participants follow the top branch G_t. This is because $\mathtt{p}$ can send $\mathtt{o}$ to $\mathtt{r}$ without waiting for $\mathtt{r}$ to first receive from $\mathtt{q}$. In this scenario, the participants disagree on which global run of $\mathsf{GAut}(G_r)$ to follow, resulting in the violating trace above.

Contrast this with G'_r and its subset construction $\{\!\{\mathscr{C}(G'_r, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ for which the FSM of $\mathtt{r}$ is depicted in Fig. 5.4b. Here, $s'_{0,\mathtt{r}}$ again has outgoing transitions labelled with $\mathtt{r}\triangleleft\mathtt{p}?\mathtt{o}$ and $\mathtt{r}\triangleleft\mathtt{q}?\mathtt{o}$. However, if $\mathtt{r}$ takes the transition labelled $\mathtt{r}\triangleleft\mathtt{p}?\mathtt{o}$, committing to the bottom branch, no disagreement occurs. This is because if the other participants are following the top branch, then $\mathtt{p}$ is blocked from sending to $\mathtt{r}$ until after it has received confirmation that $\mathtt{r}$ has received its first message from $\mathtt{q}$.

◀

For a receive transition $s \xrightarrow{x} s_1$ in $\mathscr{C}(G, \mathtt{p})$ to be safe, we must enforce that the receive event x cannot also be available due to reordered sent messages in the continuation $G_2 \in s_2$ of another outgoing receive transition $s \xrightarrow{y} s_2$. We define the set of *transition destinations* to capture all subterms that ought to be checked.

Definition 5.10 (Transition destination)**.** Let $s \xrightarrow{x} s' \in \delta_{\mathtt{p}}$ be a transition in $\mathscr{C}(G, \mathtt{p})$ and $\delta_{\downarrow}$ be the transition relation of $\mathsf{GAut}(G){\downarrow_{\mathtt{p}}}$. We define the set of *transition destinations* $\text{tr-dest}(s \xrightarrow{x} s')$ as follows:

$$\text{tr-dest}(s \xrightarrow{x} s') := \{G' \in s' \mid \exists G \in s \,.\, G \xrightarrow{x}{}^{*} G' \in \delta_{\downarrow}\} \ .$$

Available Messages. Given a receive transition $s \xrightarrow{x} s'$, $\text{tr-dest}(\text{-})$ tells us which subsequent subterms can be reached. These are precisely the subterms for which we need to check if later messages can appear in the message channels prior to receiving the message of x. In Section 3.2.2, we defined the notion of *available messages* for this. There, we considered a syntactic version $\text{avail}(\mathcal{B}, T, G)$ with a set of blocked participants $\mathcal{B}$, a set of recursion variables T and a subterm G. We also defined $\text{msgs}^{\mathcal{B}}_{(G\ldots)}$ from a language-theoretic perspective. We proved both notions to be equivalent (Lemma 3.34). In our theoretical development, we use $\text{msgs}^{\mathcal{B}}_{(G\ldots)}$. In our prototype tool, we simply compute the available messages directly on the state machine for G. Recall that, intuitively, $\text{msgs}^{\mathcal{B}}_{(G\ldots)}$ consists of all receive events $\mathtt{r}\triangleleft\mathtt{q}?\mathsf{m}$ that can occur on the traces of G such that m will be the first message added to channel $(\mathtt{q}, \mathtt{r})$ before any of the participants in $\mathcal{B}$ takes a step. If $\mathcal{B}$ is a singleton set, we omit set notation and write $\text{msgs}^{\mathtt{p}}_{(G\ldots)}$ for $\text{msgs}^{\{\mathtt{p}\}}_{(G\ldots)}$. The set of available messages captures the possible states of all channels before a given receive transition is taken.

Definition 5.11 (Receive Validity). $\mathscr{C}(G, \mathrm{p})$ satisfies *Receive Validity* iff no receive transition is enabled in an alternative continuation that originates from the same source state:

$$\forall s \xrightarrow{\mathrm{p} \triangleleft \mathrm{q}_1 ? \mathrm{m}_1} s_1,\ s \xrightarrow{\mathrm{p} \triangleleft \mathrm{q}_2 ? \mathrm{m}_2} s_2 \in \delta_{\mathrm{p}}.$$
$$\mathrm{q}_1 \neq \mathrm{q}_2 \implies \forall\, G_2 \in \text{tr-dest}(s \xrightarrow{\mathrm{p} \triangleleft \mathrm{q}_2 ? \mathrm{m}_2} s_2).\ \mathrm{p} \triangleleft \mathrm{q}_1 ? \mathrm{m}_1 \notin \mathrm{msgs}^{\mathrm{p}}_{(G_2 \ldots)} .$$

5.2.3 Subset Projection

We are now ready to define our projection operator.

Definition 5.12 (Subset projection of G). The *subset projection* $\mathscr{P}(G, \mathrm{p})$ of G onto p is $\mathscr{C}(G, \mathrm{p})$ if it satisfies Send Validity and Receive Validity. We lift this operation to a partial function from global types to CSMs in the expected way.

We conclude our discussion with an observation about the syntactic structure of the subset projection: Send Validity implies that no state has both outgoing send and receive transitions — also known as a mixed-choice state. Note that, here, mixed-choice is a condition on an FSM over Γ_{p} for some participant p and refers to the mixed choice of sending or receiving.

Corollary 5.13 (No mixed-choice states). If $\mathscr{P}(G, \mathrm{p})$ satisfies Send Validity, then for all $s \xrightarrow{x_1} s_1, s \xrightarrow{x_2} s_2 \in \delta_{\mathrm{p}}$, $x_1 \in \Gamma_!$ iff $x_2 \in \Gamma_!$.

Together with our completeness result discussed later, this implies that every implementable global type admits an implementation without mixed-choice states.

5.3 Soundness

In this section, we prove the soundness of our subset projection, stated as follows.

Theorem 5.14. Let G be a global type and $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}}$ be the subset projection. Then, $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}}$ implements G.

Recall that implementability is defined as protocol fidelity and deadlock freedom. Protocol fidelity consists of two language inclusions.

The first inclusion, $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}})$, enforces that the subset projection generates at least all behaviours of the global type. We showed in Lemma 5.5 that this holds for the subset construction alone (without Send and Receive Validity).

The second inclusion, $\mathcal{L}(\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}}) \subseteq \mathcal{L}(G)$, enforces that no new behaviours are introduced. The proof of this direction relies on a stronger inductive invariant that we show for all traces of the subset projection. As discussed in Section 5.2, violations of implementability occur when participants commit to global runs that are inconsistent with the local views of other participants. Our inductive invariant states the exact opposite: that all local views are consistent with one another. First, we formalise the local view of a participant.

Definition 5.15 (Possible run sets)**.** Let G be a global type and $\mathsf{GAut}(G)$ be the corresponding state machine. Let $\mathtt{p}$ be a participant and $w \in \Gamma^*$ be a word. We define the set of possible runs $\mathrm{R}^G_{\mathtt{p}}(w)$ as all maximal runs of $\mathsf{GAut}(G)$ that are consistent with $\mathtt{p}$'s local view of w:

$$\mathrm{R}^G_{\mathtt{p}}(w) := \{\rho \text{ is a maximal run of } \mathsf{GAut}(G) \mid w{\Downarrow_{\Gamma_{procA}}} \leq \mathrm{trace}(\rho){\Downarrow_{\Gamma_{procA}}}\} \ .$$

While Definition 5.15 captures the set of maximal runs that are consistent with the local view of a single participant, we would like to refer to the set of runs that is consistent with the local view of all participants. We formalise this as the intersection of the possible run sets for all participants, which we denote as

$$I(w) := \bigcap_{\mathtt{p} \in \mathcal{P}} \mathrm{R}^G_{\mathtt{p}}(w) \ .$$

With these definitions in hand, we can now formulate our inductive invariant:

Lemma 5.16. Let G be a global type and $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ be the subset projection. Let w be a trace of $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$. It holds that $I(w)$ is non-empty.

Prior to discussing details of how to prove this, let us briefly explain that in $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ every sent message could eventually be received. Namely, it satisfies feasible eventual reception (cf. Definition 2.1).

With the previous lemma, there is a run in the global type's state machine that all local views are compliant with. It is straightforward that all participants can catch up to the participant that progressed furthest in this run and then, by definition of global types, all channels are empty.

Corollary 5.17 (Feasible eventual reception)**.** The subset projection $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ for a global type G satisfies feasible eventual reception if it is defined.

The reasoning for the sufficiency of Lemma 5.16 is included in the proof of Theorem 5.14, found in Appendix A.2. In the rest of this section, we focus our efforts on how to show this inductive invariant, namely that the intersection of all participants' possible run sets is non-empty.

We begin with the observation that the empty trace ε is consistent with all runs. As a result, $I(\varepsilon) = \bigcap_{\mathtt{p} \in \mathcal{P}} \mathrm{R}^G_{\mathtt{p}}(\varepsilon)$ contains all maximal runs in $\mathsf{GAut}(G)$. By definition, state machines for global types include at least one run, and the base case is trivially discharged. Intuitively, $I(w)$ shrinks as more events are appended to w, but we show that at no point does it shrink to $\emptyset$. We consider the cases where a send or receive event is appended to the trace separately, and show that the intersection set shrinks in a principled way that preserves non-emptiness. In fact, when a trace is extended with a receive event, Receive Validity guarantees that the intersection set does not shrink at all.

Lemma 5.18. Let G be a global type and $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ such that $x \in \Gamma_?$. Then, $I(w) = I(wx)$.

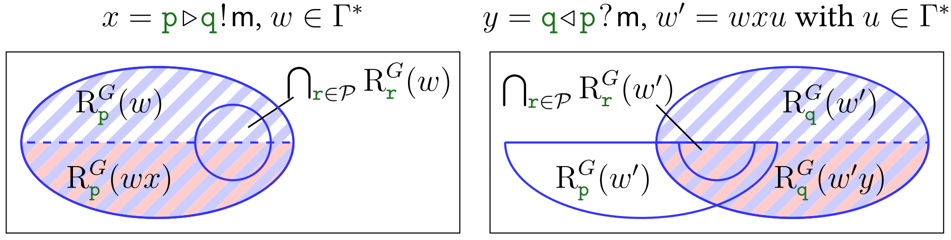


Figure 5.5: Evolution of $\mathrm{R}^G_{-}(\text{-})$ sets
when p sends a message m and q receives it.

To prove this equality, we further refine our characterisation of intersection sets. In particular, we show that in the receive case, the intersection between the sender and receiver's possible run sets stays the same, i.e.

$$\mathrm{R}^G_{\mathsf{p}}(w) \cap \mathrm{R}^G_{\mathsf{q}}(w) = \mathrm{R}^G_{\mathsf{p}}(wx) \cap \mathrm{R}^G_{\mathsf{q}}(wx) \ \ .$$

Note that it is not the case that the receiver only follows a subset of the sender's possible runs. In other words, $\mathrm{R}^G_{\mathsf{q}}(w) \subseteq \mathrm{R}^G_{\mathsf{p}}(w)$ is not inductive. The equality above simply states that a receive action can only eliminate runs that have already been eliminated by its sender. Fig. 5.5 depicts this relation.

Given that the intersection set strictly shrinks, the burden of eliminating runs must then fall upon send events. We show that send transitions shrink the possible run set of the sender in a way that is *prefix-preserving*. To make this more precise, we introduce the following definition on runs.

Definition 5.19 (Unique splitting of a possible run)**.** Let G be a global type, p a participant, and $w \in \Gamma^*$ a word. Let ρ be a possible run in $\mathrm{R}^G_{\mathsf{p}}(w)$. We define the longest prefix of ρ matching w:

$$\alpha' := \max\{\rho' \mid \rho' \leq \rho \ \wedge\ \mathrm{trace}(\rho'){\Downarrow}_{\Gamma_{procA}} \leq w{\Downarrow}_{\Gamma_{procA}}\} \ \ .$$

If $\alpha' \neq \rho$, we can split ρ into $\rho = \alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ where $\alpha' = \alpha \cdot G'$, G'' denotes the state following G', and β denotes the suffix of ρ following $\alpha \cdot G' \cdot G''$. We call $\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ the unique splitting of ρ for p matching w. We omit the participant p when obvious from context. This splitting is always unique because the maximal prefix of any $\rho \in \mathrm{R}^G_{\mathsf{p}}(w)$ matching w is unique.

When participant p fires a send transition $\mathsf{p}{\triangleright}\mathsf{q}!\mathsf{m}$, any run $\rho = \alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ in p's possible run with $l{\Downarrow}_{\Gamma_{procA}} \neq \mathsf{p}{\triangleright}\mathsf{q}!\mathsf{m}$ is eliminated. While the resulting possible run set could no longer contain runs that end with $G'' \cdot \beta$, Send Validity guarantees that it must contain runs that begin with $\alpha \cdot G'$. This is formalised by the following lemma.

Lemma 5.20. Let G be a global type and $\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ such that $x \in \Gamma_! \cap \Gamma_{procA}$ for some $\mathsf{p} \in \mathcal{P}$. Let ρ be a run in $I(w)$, and $\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ be the unique splitting of ρ for p with respect to w. Then, there exists a run ρ' in $I(wx)$ such that $\alpha \cdot G' \leq \rho'$.

This concludes our discussion of the send and receive cases in the inductive step to show the non-emptiness of the intersection of all participants' possible run sets. The full proofs and additional definitions can be found in Appendix A.2.

5.4 Completeness

In this section, we prove completeness of our approach. While soundness states that if a global type's subset projection is defined, it then implements the global type, completeness considers the reverse direction. For this, we prove the necessity of our Send and Receive Validity conditions, stated on our subset construction.

Theorem 5.21 (Completeness)**.** If G is implementable, then the subset projection $\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ is defined.

We sketch the proof and refer to Appendix A.3 for the full proof.

From the assumption that G is implementable, we know there exists a witness CSM that implements G. While the soundness proof picks our subset projection as the existential witness for showing implementability – thereby allowing us to reason directly about a particular implementation – completeness only guarantees the existence of some witness CSM. We cannot assume without loss of generality that this witness CSM is our subset construction; instead, we use the fact that it implements G to show that Send and Receive Validity hold on our subset construction.

We proceed via proof by contradiction: we assume the negation of Send and Receive Validity for the subset construction, and show a contradiction to the fact that this witness CSM implements G. In particular, we contradict protocol fidelity, stating that the witness CSM generates precisely the language $\mathcal{L}(G)$. To do so, we exploit a simulation argument: we first show that the negation of Send and Receive Validity forces the subset construction to recognise a trace that is not a prefix of any word in $\mathcal{L}(G)$. Then, we show that this trace must also be recognised by the witness CSM, under the assumption that the witness CSM implements G.

To highlight the constructive nature of our proof, we convert our proof obligation to a witness construction obligation. To contradict protocol fidelity, it suffices to construct a witness trace v_0 satisfying two properties, where $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ is our witness CSM:

(a) v_0 is a trace of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$, and
(b) the run intersection set of v_0 is empty: $I(v_0) = \bigcap_{\mathsf{p}\in\mathcal{P}} \mathrm{R}^G_\mathsf{p}(v_0) = \emptyset$.

We first establish the sufficiency of conditions (a) and (b). Because $\{\!\{B_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ is deadlock-free by assumption, every prefix extends to a maximal trace. Thus, to prove the inequality of the two languages $\mathcal{L}(\{\!\{B_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}})$ and $\mathcal{L}(G)$, it suffices to prove the inequality of their respective prefix sets. In turn, it suffices to show the existence of a prefix of a word in one language that is not a prefix of any word in the other. We choose to construct a prefix in the CSM language that is not a prefix in $\mathcal{L}(G)$. We again leverage the definition of intersection sets (Definition 5.15) to weaken the property of language non-membership to the property of having an empty intersection set as follows. By the semantics of $\mathcal{L}(G)$, for any $w \in \mathcal{L}(G)$, there exists $w' \in \mathcal{L}(\mathsf{GAut}(G))$ with $w \sim w'$. For any $w' \in \mathcal{L}(\mathsf{GAut}(G))$, it trivially holds that w' has a non-empty intersection set. Because intersection sets are invariant under the indistinguishability relation $\sim$, w must also have a non-empty intersection set. Since intersection sets are monotonically decreasing, if the intersection set of w is non-empty, then for any $v \leq w$, the intersection set of v is also non-empty. Modus tollens of the chain of reasoning above tells us that in order to show a word is not a prefix in $\mathcal{L}(G)$, it suffices to show that its intersection set is empty.

Having established the sufficiency of properties (a) and (b) for our witness construction, we present the steps to construct v_0 from the negation of Send and Receive Validity respectively. We start by constructing a trace in $\{\!\{\mathscr{C}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ that satisfies (b), and then show that $\{\!\{B_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ also recognises the trace, thereby satisfying (a). In both cases, let p be the participant and s be the state for which the respective validity condition is violated.

Send Validity (Definition 5.8). Let $s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!\mathsf{m}} s' \in \delta_{\mathsf{p}}$ be a transition such that

$$\text{tr-orig}(s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!\mathsf{m}} s') \neq s \quad .$$

First, we find a trace u of $\{\!\{\mathscr{C}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ that satisfies: (1) participant p is in state s in the CSM configuration reached via u, and (2) the run of $\mathsf{GAut}(G)$ on u visits a state in $s \setminus \text{tr-orig}(s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!\mathsf{m}} s')$. We obtain such a witness u from the trace of a run prefix of $\mathsf{GAut}(G)$ that ends in some state in $s \setminus \text{tr-orig}(s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!\mathsf{m}} s')$. Any prefix thus obtained satisfies (1) by definition of $\mathscr{C}(G,\mathsf{p})$, and satisfies (2) by construction. Due to the fact that send transitions are always enabled in a CSM, $u \cdot \mathsf{p}\triangleright\mathsf{q}!\mathsf{m}$ must also be a trace of $\{\!\{\mathscr{C}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$, thus satisfying property (a) by a simulation argument. We then argue that $u \cdot \mathsf{p}\triangleright\mathsf{q}!\mathsf{m}$ satisfies property (b), stating that $I(u \cdot \mathsf{p}\triangleright\mathsf{q}!\mathsf{m})$ is empty: the negation of Send Validity gives that there exist no run extensions from our candidate state in $s \setminus \text{tr-orig}(s \xrightarrow{\mathsf{p}\triangleright\mathsf{q}!\mathsf{m}} s')$ with the immediate next event $\mathsf{p} \to \mathsf{q} : \mathsf{m}$, and therefore there exists no maximal run in $\mathsf{GAut}(G)$ consistent with $u \cdot \mathsf{p}\triangleright\mathsf{q}!\mathsf{m}$.

Receive Validity (Definition 5.11). Let $s \xrightarrow{\mathsf{p}\triangleleft\mathsf{q}_1?\mathsf{m}_1} s_1$ and $s \xrightarrow{\mathsf{p}\triangleleft\mathsf{q}_2?\mathsf{m}_2} s_2 \in \delta_{\mathsf{p}}$ be two transitions, and let $G_2 \in \text{tr-dest}(s \xrightarrow{\mathsf{p}\triangleleft\mathsf{q}_2?\mathsf{m}_2} s_2)$ such that

$$\mathsf{q}_1 \neq \mathsf{q}_2 \text{ and } \mathsf{p}\triangleleft\mathsf{q}_1?\mathsf{m}_1 \in \text{msgs}^{\mathsf{p}}_{(G_2\ldots)} \quad .$$

Constructing the witness v_0 pivots on finding a trace u of $\{\!\{\mathscr{C}(G,\mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ such that both $u \cdot \mathrm{p}\triangleleft\mathrm{q}_1?\mathsf{m}_1$ and $u \cdot \mathrm{p}\triangleleft\mathrm{q}_2?\mathsf{m}_2$ are traces of $\{\!\{\mathscr{C}(G,\mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$. Equivalently, we show there exists a reachable configuration of $\{\!\{\mathscr{C}(G,\mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ in which p can receive either message from distinct senders q_1 and q_2. Formally, the local state of p has two outgoing states labelled with $\mathrm{p}\triangleleft\mathrm{q}_1?\mathsf{m}_1$ and $\mathrm{p}\triangleleft\mathrm{q}_2?\mathsf{m}_2$, and the channels $(\mathrm{q}_1,\mathrm{p})$ and $(\mathrm{q}_2,\mathrm{p})$ have m_1 and m_2 at their respective heads. We construct such a trace u by considering a run in $\mathsf{GAut}(G)$ that contains two transitions labelled with $\mathrm{q}_1\to\mathrm{p}:\mathsf{m}_1$ and $\mathrm{q}_2\to\mathrm{p}:\mathsf{m}_2$. Such a run must exist due to the negation of Receive Validity. We start with the split trace of this run, and argue that, from the definition of msgs^{-}_{-} and the indistinguishability relation $\sim$, we can perform iterative reorderings using $\sim$ to move the send event $\mathrm{q}_1\triangleright\mathrm{p}!\mathsf{m}_1$ to the position before the receive event $\mathrm{p}\triangleleft\mathrm{q}_2?\mathsf{m}_2$. Then, (a) for $u \cdot \mathrm{p}\triangleleft\mathrm{q}_1?\mathsf{m}_1$ holds by a simulation argument. We then separately show that (b) holds for $\mathrm{p}\triangleleft\mathrm{q}_1?\mathsf{m}_1$ using similar reasoning as the send case to complete the proof that $u \cdot \mathrm{p}\triangleleft\mathrm{q}_1?\mathsf{m}_1$ suffices as a witness for v_0.

It is worth noting that the construction of the witness prefix v_0 in the proof immediately yields an algorithm for computing counterexample traces to implementability.

5.5 CSMs vs. Local Types

We consider CSMs as implementation model for global types. In contrast, most MST frameworks use local types as introduced in Section 3.1.

Local types are an expression-based formalism but we also showed how to interpret them as state machines. They are not equi-expressive though. The syntactic restrictions of local types have interesting consequences on the expressive power of possible implementations (cf. Proposition 3.5). We divide these observations about state machines built from local types in three parts:

(1) There are no mixed-choice states, i.e. there is no state with both outgoing send and receive transitions.
(2) They are sink-final. (Every final state has no outgoing transitions and every state without outgoing transitions is final.)
(3) Every such state machine has a tree-like structure and recursion happens at leaves to ancestors.

For (1), we will show that this is not restrictive when it comes to implementability. In fact, every (sender-driven) global type can be implemented without mixed-choice states. (2) has influences on the semantics of deadlocks. In the implementability problem under consideration, a deadlock is a non-final CSM configuration that got stuck, i.e. there are no further CSM transitions. We introduced the soft implementability problem in Section 2.7, which employs a different notion of deadlock. There, a stuck configuration is a soft deadlock if any local state machine has an outgoing transition even if the

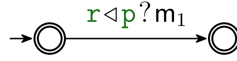


Figure 5.6: Subset projection of global type that is not sink-final.

configuration is final. We proved our subset projection operator complete for the implementability problem. The soft implementability problem can also be solved with our subset projection operator, using a simple postprocessing step. In Section 8.2.2, we will show that the restrictions in (3) do not change the expressivity of sink-final state machines without mixed-choice states.

5.5.1 On Mixed-choice States

Our completeness result (Theorem 5.21) basically shows the necessity of Send Validity for implementability. Corollary 5.13 shows that Send Validity precludes states with both send and receive outgoing transitions. Together, this implies that an implementable (sender-driven) global type can always be implemented without mixed-choice states. Note that the syntactic restrictions on (sender-driven) global types do not inherently prevent mixed-choice states from arising in a participant's subset construction, as evidenced by $\mathtt{r}$ in the following type:

$$\mathtt{p}\to\mathtt{q}:\mathsf{l}.\mathtt{q}\to\mathtt{r}:\mathsf{m}.0 + \mathtt{p}\to\mathtt{q}:\mathsf{r}.\mathtt{r}\to\mathtt{q}:\mathsf{m}.0 \ .$$

Its subset construction onto $\mathtt{r}$ exposes a mixed-choice state and exemplifies that the syntactic restrictions on global types do not prevent this. Our completeness result thus implies that this global type is not implementable. Most MST frameworks [39, 67] implicitly force *no mixed-choice (states)* through syntactic restrictions on local types. We are the first to prove that mixed-choice states are indeed not necessary for completeness. This is interesting because mixed choice is known to be crucial for the expressive power of the synchronous π-calculus compared to its asynchronous variant [102].

5.5.2 Sink States and Deadlocks

When using local types, final configurations are always sink-state configurations, i.e. where each participant is in a sink state. For our setting, this is not the case and has repercussions on the semantics and meaning of deadlocks: our state machines can have final states with outgoing transitions and such states can be part of final configurations. If there is no next transition for such a configuration, it is a soft deadlock. The soft implementability problem (Definition 2.33) employs this notion for deadlocks: a soft implementation needs to be free from soft deadlocks and generate the same protocol. The state machines for global types are all sink-final, i.e. every state is final if and only if it is a sink state. Despite, there are global types that are implementable but not softly implementable.

Example 5.22. Consider the following directed global type:

$$G := \begin{cases} \mathtt{p} \rightarrow \mathtt{q} : \mathtt{m}_1 \,.\, \mathtt{p} \rightarrow \mathtt{r} : \mathtt{m}_1 \,.\, 0 \\ \mathtt{p} \rightarrow \mathtt{q} : \mathtt{m}_2 \,.\, 0 \end{cases} .$$

It is implementable by its subset projection. We give the subset projection for $\mathtt{r}$ in Fig. 5.6. It has a final state with an outgoing receive transition. Note that the global type is directed, i.e. sender and receiver are the same for both branches. However, the conditions on choice do only impose restrictions for the branches and not what happens subsequently. Here, $\mathtt{r}$ is not involved in the bottom branch, making its initial state final. ◀

With our completeness result, we showed that Send Validity is a necessary condition for implementability. It requires that every send transition is possible from every subterm in the respective state. A state is only final if 0 is in this state. No send transition is possible from 0, which is why there cannot be final states with outgoing send transitions in a subset projection.

Intuitively, if one aims for soft deadlock freedom, no state with outgoing transitions needs to be final. For soft deadlocks, only final sink-state configurations matter. One way to achieve this is to require that the implementation is sink-final. However, while this is a sufficient condition, its necessity is not obvious. A soft deadlock requires a reachable final non-sink-state configuration that is stuck. At first, it seems possible that such a final configuration never gets stuck in a CSM that is not sink-final: intuitively, it might never make use of the fact that this final configuration is final but it always continues from this configuration anyway. We show that this is not the case if one of two conditions hold: every local state machine of the CSM is deterministic or it ensures that every sent message will be received eventually, which we called feasible eventual reception.

In addition, we show that the subset projection is sink-final if and only if a global type is implementable by any sink-final CSM satisfying one of both conditions. Together, this shows that, given a global type, we can construct its subset projection and check if it is sink-final if we aim for a deterministic soft implementation. Note that CSM from local types always satisfy the condition on determinacy. It is also reasonable to only consider deterministic CSMs for type-checking, as we do in Chapter 6. In the presence of non-determinism, the type-checking would need to check along all possibilities anyway.

For our result, it is fine to assume implementability of G as it is a prerequisite for soft implementability.

Theorem 5.23. Let G be an implementable global type. Then, the following statements are equivalent:

(a) G can be implemented by a sink-final CSM that satisfies feasible eventual reception or every of its state machines is deterministic.

(b) The subset construction $\mathscr{C}(G, \mathtt{p})$ is sink-final for every participant $\mathtt{p}$.

(c) All reachable final configurations of $\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ are sink-state configurations.

(d) G can be softly implemented by a CSM that satisfies feasible eventual reception or every of its state machines is deterministic.

Proof. We can assume Send and Receive Validity hold for the subset construction, entailing that the subset projection is defined. This is justified by the fact that G is implementable and the completeness theorem (Theorem 5.21). As argued before, because of Send Validity, any final state in the subset projection cannot have outgoing send transitions.

We prove four implications, yielding a cycle and equivalence of all statements.

Proof that (a) implies (b):

Let $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a sink-final CSM that satisfies feasible eventual reception or every of its state machines is deterministic, and implements G. Towards a contradiction, assume that the subset construction is not sink-final. Without loss of generality, let $\mathscr{C}(G, \mathtt{p})$ be the subset construction with at least one final non-sink state and let s be one of the states that is final and has outgoing transitions.

By the fact that s is final, there is $0 \in s$. By the fact that s has outgoing transitions, there is $G' \in s$ with $G' \xrightarrow{x} G''$ for $x\Downarrow_{\Gamma_{\mathtt{p}}} \in \Gamma_{\mathtt{p}}$ and some G''. Because of Send Validity, we have $x = \mathtt{q}\!\to\!\mathtt{p}\!:\!\mathsf{m}$ for some participant $\mathtt{q}$ and message m.

In the subset construction, two subterms G_1 and G_2 do only occur in the same state $\vec{s}_{\mathtt{p}}$ of $\mathscr{C}(G, \mathtt{p})$ if there are two runs $G \xrightarrow{w}^{*} G_1$ and $G \xrightarrow{w'}^{*} G_2$ with $w \in \Gamma^*$, $w' \in \Gamma^*$, and $w\Downarrow_{\Gamma_{\mathtt{p}}} = w'\Downarrow_{\Gamma_{\mathtt{p}}}$. Here, we use choose $G_1 = 0$ and $G_2 = G'$. Thus, we have $w \in \mathcal{L}(G)$.

Let $(\vec{s}, \xi)$ be the configuration of the subset construction $\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ that is reached after processing w. Because of protocol fidelity, determinacy of the subset construction and $w \in \mathcal{L}(G)$, it holds that $(\vec{s}, \xi)$ is final. Recall there is a transition from G' to G'' labelled with $\mathtt{q}\!\to\!\mathtt{p}\!:\!\mathsf{m}$, so we can extend w' to obtain $w'' := w' \cdot \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}$ for which $w\Downarrow_{\Gamma_{\mathtt{p}}} = w''\Downarrow_{\Gamma_{\mathtt{p}}}$. Let $(\vec{s}'', \xi'')$ be the configuration of $\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ after processing w''. By construction of the traces, the channels are empty after processing w''. Hence, we have $\xi''(\mathtt{q}, \mathtt{p}) = \mathsf{m}$ with the additional send event in w''. Because of $w\Downarrow_{\Gamma_{\mathtt{p}}} = w''\Downarrow_{\Gamma_{\mathtt{p}}}$, it holds that $\vec{s}_{\mathtt{p}} = \vec{s}''_{\mathtt{p}}$. Let us consider the two configurations of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ that are reached with w and w''. By Lemma 2.5, they will have the same channel contents as the subset construction respectively. Let $(\vec{t}, \xi)$ and $(\vec{t}'', \xi'')$ be the configurations of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ after processing w and w'' respectively. By $w\Downarrow_{\Gamma_{\mathtt{p}}} = w''\Downarrow_{\Gamma_{\mathtt{p}}}$, it is possible and we assume that $\vec{t}_{\mathtt{p}} = \vec{t}''_{\mathtt{p}}$. Note that we do not assume determinacy of $\{\!\{A_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ but we can assume that both runs end in the same state for $\mathtt{p}$ since all ways of non-determinism need to be accounted for. We show that $\vec{t}_{\mathtt{p}}$ is not sink-final: it is final but has at least one outgoing transition.

To start, we show that $\vec{t}_{\mathtt{p}}$ is final. It suffices to show that $(\vec{t}, \xi)$ is a final configuration. By the fact that $(\vec{s}, \xi)$ is final, ξ has only empty channels. Towards a contradiction, assume that $\vec{t}_{\mathtt{r}}$ is not final for some participant $\mathtt{r}$. Then, w is not in $\mathcal{L}(\{\!\{A'_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}})$ if there is no other run for w but, if there were, $\{\!\{A'_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ still deadlocks in the configuration $(\vec{t}, \xi)$, contradicting deadlock freedom. Hence, $(\vec{t}, \xi)$ is final.

It remains to show that $\vec{t}_{\mathrm{p}}$ has an outgoing transition. We do a case analysis on the side condition for $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$.

First, we assume that $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ satisfies feasible eventual reception. We use the second configuration $(\vec{t}'', \xi'')$ where p is in the same state. We know that m is the first message in $\xi''(\mathrm{q}, \mathrm{p})$. It was sent and must be received. Thus, $\vec{t}''_{\mathrm{p}}$ has at least one outgoing transition.

Second, assume that A_{r} is deterministic for every participant r. Again, we use the second configuration $(\vec{t}'', \xi'')$ where p is in the same state. By the semantics of global types, there is an extension w''' of w'' with $w''' \in \mathcal{L}(G)$ that contains the receive event $\mathrm{p}\triangleleft\mathrm{q}?\mathrm{m}$ for the enqueued message m. If w''' is finite, it is straightforward that p needs to be able to receive the message m from q to satisfy protocol fidelity, ensuring an outgoing transition. If w'' is infinite, towards a contradiction, assume that $\vec{t}_{\mathrm{p}}$ has no outgoing transition. The semantics require that every prefix $u \in \mathrm{pref}(w''')$ is in $\mathrm{pref}(\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}})$. However, if $\vec{t}_{\mathrm{p}}$ could not receive m, because of determinacy of A_{p}, there is a prefix of w''' that is not. This contradicts the assumption that $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ implements G.

Proof that (b) implies (c):
By assumption, all final configurations are sink-state configurations. Hence, all reachable final configurations are sink-state configurations.

Proof that (c) implies (d):
We claim that $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ is such a CSM. By soundness (Theorem 5.14), we know that $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ satisfies protocol fidelity and is deadlock-free. Hence, it suffices to show $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ is free from soft deadlocks. Since $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ is deadlock-free, the only way there could be soft deadlocks is that there is a reachable stuck final configuration that is no sink-state configuration. This is impossible because every reachable final configuration is a sink-state configuration by assumption. It remains to show that $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}}$ satisfies one of both side conditions. In fact, it satisfies both as argued earlier.

Proof that (d) implies (a):
By definition, every soft implementation is also an implementation for G because soft deadlock freedom implies deadlock freedom. We prove the following: a softly implementable global type can be implemented by a sink-final CSM, i.e. for which all state machines are sink-final. This is sufficient for our claim. Let $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ be a soft implementation for G that is not sink-final for some participant p. This means that A_{p} has final states that have outgoing transitions. We show that these states do not need to be final for any such p. Hence, we can turn $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ to a sink-final CSM $\{\!\{A'_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ by inductively applying this to all participants whose state machines are not sink-final. Observe that whether a state is final or not does only matter for soft deadlock freedom and not protocol fidelity. From soft deadlock freedom, we know that there is no stuck reachable final non-sink-state configuration. In other words, all the stuck configurations are final sink-state configurations. Thus, any non-sink final state does not need to be final. This proves the claim that the existence of $\{\!\{A_{\mathrm{p}}\}\!\}_{\mathrm{p}\in\mathcal{P}}$ always implies

the existence of a sink-final CSM $\{\!\{A'_{\mathtt{p}}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ which is our witness for implementability. The side conditions for both statements are the same and not affected by our construction so they simply carry over. □

This result also shows that our method will always find a sink-final implementation if it exists. Hence, if it provides one that is not, the global type cannot be implemented with a sink-final implementation. If this is undesirable, the protocol ought to be redesigned.

5.6 Complexity

In this section, we establish a PSPACE upper bound for the MST implementability problem and show there is a family of implementable directed global types that requires projections of exponential size.

Theorem 5.24. Checking implementability for global types with sender-driven choice is in PSPACE.

Proof. The decision procedure enumerates the subsets of $\mathsf{GAut}(G)\downarrow_{\mathtt{p}}$ for each participant $\mathtt{p}$. This can be done in polynomial space and exponential time. For each $\mathtt{p}$ and $s \subseteq Q_G$, it then (i) checks membership of s in $Q_{\mathtt{p}}$ of $\mathscr{C}(G, \mathtt{p})$, and (ii) if $s \in Q_{\mathtt{p}}$, checks whether all outgoing transitions of s in $\mathscr{C}(G, \mathtt{p})$ satisfy Send and Receive Validity. Check (i) can be reduced to the intersection non-emptiness problem for nondeterministic finite state machines, which is in PSPACE [129]. It is easy to see that check (ii) can be done in polynomial time. In particular, the computation of available messages for Receive Validity only requires a single unfolding of every loop in G.

Note that the synthesis problem has the same complexity. The subset construction to determinise $\mathsf{GAut}(G)\downarrow_{\mathtt{p}}$ can be done using a PSPACE transducer. While the output can be of exponential size, it is written on an extra tape that is not counted towards memory usage. However, this means we need to perform the validity checks as described above instead of using the computed deterministic state machines.

□

Lemma 5.25. Constructing an implementation for a global type with directed choice may require exponential time.

Proof. We construct a family of directed global types G_k that is implementable and requires a projection of exponential size for $\mathtt{q}$. Hence, constructing the projections requires exponential time in the size of the input.

$$G_k := \mu t\,.\,+\begin{cases} \mathtt{p}\!\to\!\mathtt{r}\!:\!s\,.\,+\begin{cases}\mathtt{p}\!\to\!\mathtt{q}\!:\!a\,.\,t\\ \mathtt{p}\!\to\!\mathtt{q}\!:\!b\,.\,t\end{cases}\\ \mathtt{p}\!\to\!\mathtt{r}\!:\!l\,.\,+\begin{cases}\mathtt{p}\!\to\!\mathtt{q}\!:\!a\,.\,G_{a,k-1}\\ \mathtt{p}\!\to\!\mathtt{q}\!:\!b\,.\,G_{b,k-1}\end{cases}\end{cases}\qquad\text{where}$$

$$G_{a,i} := +\begin{cases} \mathtt{p}\to\mathtt{q}{:}a\,.\,G_{a,i-1} \\ \mathtt{p}\to\mathtt{q}{:}b\,.\,G_{a,i-1} \end{cases} \text{for } i>0 \qquad G_{b,i} := +\begin{cases} \mathtt{p}\to\mathtt{q}{:}a\,.\,G_{b,i-1} \\ \mathtt{p}\to\mathtt{q}{:}b\,.\,G_{b,i-1} \end{cases} \text{for } i>0$$

$$G_{a,0} := \mathtt{p}\to\mathtt{q}{:}d\,.\,\mathtt{q}\to\mathtt{p}{:}a\,.\,0 \qquad G_{b,0} := \mathtt{p}\to\mathtt{q}{:}d\,.\,\mathtt{q}\to\mathtt{p}{:}b\,.\,0$$

Intuitively, p sends a sequence of letters from $\{a, b\}$ to q, followed by d to indicate that the sequence is over. Then, q needs to send the kth last letter back to p. Hence, q needs to remember the last k letters at all times, yielding at least 2^k different states for its projection. This is a variation of a language that is well-known to be recognisable only by a DFA with exponentially many states, compared to a minimal NFA: all words over $\{a, b\}$ where a is the kth last letter. Note that this result hinges on the fact that multiple occurences of the same subterm do not contribute to the size of the global type. □

In the above example, we require q to remember the last k messages it received at all times. Since words of any size greater than k are accepted, any local implementation for q will accept precisely the words from the projection. We call this local language preserving.

Definition 5.26. Let G be an implementable global type and $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be a CSM that implements G. We say $\{\!\{A_\mathtt{p}\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is a *local language preserving* implementation for G if $\mathcal{L}(\mathscr{P}(G, \mathtt{p})) = \mathcal{L}(G)\Downarrow_{\Gamma_\mathtt{p}}$ for every $\mathtt{p} \in \mathcal{P}$.

Our subset projection yields an implementation where local languages for praticipants are preserved (Lemma 5.4). Together with our completeness result, this leads to the following observation.

Corollary 5.27. Let G be an implementable global type. Then, the subset projection $\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is a local language preserving implementation for G and can be computed in PSPACE.

5.7 Evaluation

We consider the following three aspects in the evaluation of our approach: (E1) difficulty of implementation (E2) completeness, (E3) comparison to state of the art for protocols with sender-driven choice, and (E4) confirming the exponential lower bound for the output.

For this, we implemented our subset projection in a prototype tool, which is publicly available [106, 114]. It takes a global type as input and computes the subset projection for each participant. It was straightforward to implement the core functionality in approximately 700 lines of Python3 code closely following the formalisation (E1).

Source	Name	Impl.	Subset Proj. (complete)		Size	$\lvert\mathcal{P}\rvert$	Size Proj's	Gen. Proj. (incomplete)	
[111]	Instrument Contr. Prot. A	✓	✓	0.3 ms	22	3	48	✓	0.1 ms
	Instrument Contr. Prot. B	✓	✓	0.3 ms	18	3	37	✓	< 0.1 ms
	OAuth2	✓	✓	0.1 ms	10	3	22	✓	< 0.1 ms
[108]	Multi Party Game	✓	✓	0.4 ms	20	3	43	✓	0.1 ms
[67]	Streaming	✓	✓	0.2 ms	13	4	28	✓	< 0.1 ms
[29]	Non-Compatible Merge	✓	✓	0.1 ms	10	3	22	✓	< 0.1 ms
[113]	Spring-Hibernate	✓	✓	0.7 ms	48	6	100	✓	0.6 ms
Section 3.2.6	Group Present	✓	✓	0.5 ms	37	4	77	✓	0.5 ms
	Late Learning	✓	✓	0.3 ms	18	4	34	✓	0.1 ms
	Load Balancer ($n = 10$)	✓	✓	3.0 ms	37	12	88	✓	1.9 ms
	Logging ($n = 10$)	✓	✓	56.4 ms	82	13	304	✓	7.7 ms
Section 3.1.1	2 Buyer Protocol	✓	✓	0.4 ms	22	3	48	✓	0.1 ms
Section 3.4	2B-Prot. Omit No	✓	✓	0.3 ms	20	3	48	(×)	< 0.1 ms
	2B-Prot. Subscription	✓	✓	0.6 ms	27	3	68	(×)	0.3 ms
	2B-Prot. Inner Recursion	✓	✓	0.3 ms	18	3	39	✓	< 0.1 ms
	Odd-even	✓	✓	0.4 ms	34	3	65	(×)	0.1 ms
Section 5.2	G_r – Receive Val. Violated	×	×	0.1 ms	12	3	-	(×)	< 0.1 ms
	G'_r – Receive Val. Satisfied	✓	✓	0.2 ms	16	3	34	✓	< 0.1 ms
	G_s – Send Val. Violated	×	×	< 0.1 ms	8	3	-	(×)	< 0.1 ms
	G'_s – Send Val. Satisfied	✓	✓	< 0.1 ms	6	3	13	✓	< 0.1 ms
Section 5.6	State Space Expl. ($n = 5$)	✓	✓	13.5 ms	52	3	508	×	2.0 ms

Table 5.1: Projecting Global Types. For every protocol, we report whether it is implementable ✓ or not ×, the time to compute our subset projection and our generalised projection as well as the outcome as ✓ for "implementable", × for "not implementable" and (×) for "not known". We also give the size of the protocol (number of states and transitions), the number of participants, the combined size of all subset projections (number of states and transitions).

We consider the communication protocols from earlier (cf. Table 3.1) as well as new examples from this thesis (cf. 1st column of Table 5.1). Our experiments were run on a computer with an Intel Core i5-1335U CPU and used at most 100MB of memory. The results are summarised in Table 5.1. The reported size is the number of states and transitions of the respective state machine, which allows not to account for multiple occurrences of the same subterm. As expected, our tool can project every implementable protocol we have considered (E2).

Regarding the comparison against the state of the art for protocols with sender-driven choice (E3), we directly compared our subset projection to our incomplete generalised projection (Section 3.2), and found that the run times are in the same order of magnitude in general (typically a few milliseconds). However, as expected, the generalised projection operator fails to project four implementable protocols (including Example 3.43). We further note that most of the run times reported by Scalas and

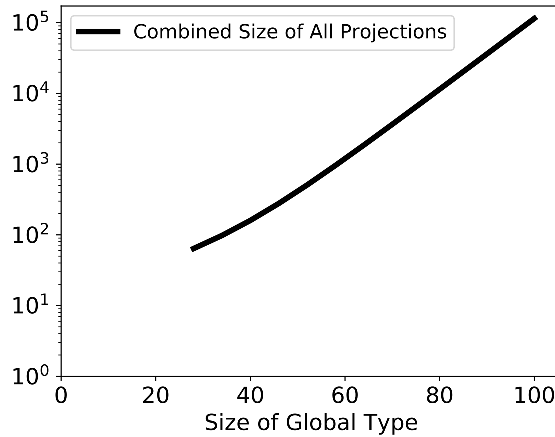


Figure 5.7: Projecting G_k (cf. Section 5.6) for $0 < n < 14$ with our prototype tool; x-axis: size of G_k; y-axis: combined size of all subset projections in log-scale.

Yoshida [109] on their model checking based tool are around 1 second and are thus two to three orders of magnitude slower and they do not check protocol fidelity.

To confirm the exponential lower bound for implementations of global types (E4), we project the family of protocols in Section 5.6. We found that the subset projection for one participant grows exponentially in the size of the global type (cf. Fig. 5.7). This experiment used at most 500MB of memory.

5.8 Properties Entailed by Implementability

We defined implementability for a protocol as the question whether there exists a deadlock-free CSM that generates the same language as the protocol specifies. Various other properties of implementations and protocols have been proposed in the literature. For systems with two participants, an *unspecified reception* [34, Def. 12] occurs if a receiver cannot receive the first message in its (only) channel. Intuitively, this yields a deadlock for a binary system or an execution in which the other participant will send messages indefinitely. For multi-party systems with sender-driven choice, this is not necessarily the case and our receive validity condition ensures that a message in a participant's channel can appear but will not confuse the receiver regarding which choices have been made. Thus, we could lift the property to multiparty systems with a universal quantification over channels. Our subset projection would prevent this lifted version of unspecified receptions because of deadlock freedom and the fact that any non-deadlock configuration can be extended in a way that the unspecified reception can eventually be received. Similarly, our subset projection ensures the *absence of orphan messages* [45, Sec. 2][16, Sec. 3]. *Orphan messages* have been defined using *final-state* configurations, i.e. where all states are final: there is a reachable final-state configuration whose channels are not empty. This is reasonable if every final state is

also a sink state. This is not the case for our implementation model though. We refer to Section 5.5.2 for details and how our approach can be used as solution for the soft implementatability problem. In short, if our subset projection provides a solution to the soft implementability problem, sink-state configurations and final-state configurations coincide and our approach guarantees absence of orphan messages for these kind of configurations. For the implementability problem, our soundness proof ensures the absence of orphan messages in sink-state configurations and ensures that messages in final-state configurations can be received eventually.

Deadlock freedom is sometimes also studied as *progress* – in the sense that a system should never get stuck. The standard notion of *progress* [39, Sec. 1] asks that every sent message is eventually received and every participant waiting for a message eventually receives one. We call the property that a sent message is eventually received feasible eventual reception. We proved our subset projection sound for finite as well as infinite CSM runs. For finite runs, both properties trivially hold. For infinite runs, our subset projection ensures that both is possible but one would require fairness assumptions to ensure that it will actually happen as is common for liveness properties. First, we can impose a (strong) fairness assumption – as Castagna et al. [29]. Second, we can require that every loop branch contains at least all participants that occur in interactions of any path with which the protocol can finish.

In our subset projection, it is also guaranteed that each transition of a local implementation can be fired in some execution of the subset projection. This is called *executable* by Cécé and Finkel [34, Def. 12] while it is the property *live* in work by Scalas and Yoshida [109, Fig. 5(3)] and called *liveness* by Barbanera et al. [11, Def. 2.9].

A CSM has the *stable property* if any reachable configuration has a transition sequence to a configuration with empty channels. With our proof technique, we showed every run of a CSM has a common path in the protocol that complies with all participants' local observations of the run. There is a furthest point in this path and it is possible that all participants catch up to this point, having empty channels.

Scalas and Yoshida [109] also consider two properties that are rather protocol-specific than implementation-specific, i.e. protocol fidelity and deadlock freedom trivially ensure that every implementation satisfies these properties if the protocol does. First, a global type is *terminating* [109, Fig. 5(2)] if every CSM run is finite. It is trivial that this is only true if there are no (used) recursion variable binders (μt). Second, a global type is *never-terminating* [109, Fig. 5(3)] if every CSM run is infinite. Consequently, this is only the case if the global type has no term 0.

Chapter 6

A Type System Using Communicating State Machines

In the previous chapter, we presented how to go from global types to communicating state machines as implementation model. Usually, MST frameworks use local types instead. However, with our sound and complete approach, we showed that CSMs are well-suited for projection. After projection, local types are usually used as local specifications in a type system. This raises the question if CSMs are also a good fit for type-checking. In this chapter, we answer this question positively and show that CSMs can be seamlessly integrated into a type system. To show this, we present a session type system that uses CSMs instead of local types.

The presented type system nicely complements the use of CSMs as implementation model for our sound and complete subset projection. One can take global types as global protocol specifications and type-check processes against them, with CSMs as intermediate interface for local specifications. This clean separation of concerns makes our type system applicable to any type of protocol specification that can be projected to CSMs. Our results show that CSMs are useful and advantageous for both projection and type-checking. CSMs are strictly more expressive than local types. Thus, the use of CSMs as intermediate interface improves generality without loosing efficiency.

For the design of our type system, we follow [109] as a particularly streamlined instance of a session type system. There are two main differences. Scalas et al. [109] do not consider global types, so they do not check against a global protocol specification, but they still use local types. In our type system, we use CSMs which are extracted from global types.

6.1 Payload Types and Delegation

So far, we treated message payloads as uninterpreted names from a fixed finite set. In practice, each message would be a label and a payload type. The label can indicate the branch that was taken while the payload type is interpreted as type of the data transmitted. For instance, consider the following global type:

$$G := \mathtt{p} \to \mathtt{q} : l(\mathtt{str}) + \mathtt{p} \to \mathtt{q} : r(\mathtt{int})$$

Here, l and r are labels so q knows which branch was taken. For the right branch, `int` is interpreted as type of the payload. Thus, the payload should be of type integer, e.g. the number 2. If there is no payload, we simply omit it and only write the label.

A global type specifies the intended behaviour for one session. With a type system, it is interesting to consider systems with multiple sessions that possibly follow protocols given by different global types. For session s, the endpoint of participant p is denoted by $s[\mathsf{p}]$.

Let us give a rather informal example of a process that follows G. More precisely, let s be the session that follows G. Then, the process $P_{\mathsf{p}} \parallel P_{\mathsf{q}}$, where $\parallel$ is parallel composition, would comply with the above global type:

$$P_{\mathsf{p}} := s[\mathsf{p}][\mathsf{q}]!l\langle\text{“foo”}\rangle \,.\, 0 \oplus s[\mathsf{p}][\mathsf{q}]!r\langle 2\rangle \,.\, 0$$
$$P_{\mathsf{q}} := s[\mathsf{q}][\mathsf{p}]?l(x) \,.\, 0 \;\&\; s[\mathsf{q}][\mathsf{p}]?r(y) \,.\, 0$$

We use internal choice $\oplus$ for P_{p} because it sends first: $s[\mathsf{p}][\mathsf{q}]!r\langle 2\rangle$ indicates that endpoint $s[\mathsf{p}]$ sends message $r\langle 2\rangle$ to $s[\mathsf{q}]$. For P_{q}, we use external choice $\&$ as it cannot actually choose but receives the sent message. It stores the payload in a variable which could be used subsequently, e.g. $s[\mathsf{q}][\mathsf{p}]?l(x)$ stores the message in x. Both processes terminate after their only action.

If all payload types are base types like `str`, `int`, `Bool`, etc., type-checking is rather simple. It gets more complicated and interesting if one allows to send channel endpoints. This amounts to sending an endpoint of a session, e.g. $s[\mathsf{p}]$. Upon receiving, the receiver shall subsequently comply with what is specified. This is why sending a channel endpoint is called *delegation*. Usually, local types are used to specify channel endpoint types. However, we do not use local types so how do we specify such behaviour? We use the states from the subset projections of a global type because this specifies the behaviour of a participant in our setting. We assume that they are distinct across all considered state machines. Our implementation model uses finite state machines, giving us a finite set of labels. Note that one cannot only send the initial state. Each state corresponds to a position in the local behaviour. Sending non-initial states corresponds to sending subexpressions of local types.

Example 6.1 (Global Types with Delegation)**.** We consider a protocol where (one) buyer a buys a book from a seller s who delegates the payment to a payment service p. The one buyer protocol is specified as follows:

$$G_1 := \mathsf{a}\to\mathsf{s}:\mathsf{query}(\mathtt{str})\,.\,\mathsf{s}\to\mathsf{a}:\mathsf{price}(\mathtt{int})\,.\,+\begin{cases}\mathsf{a}\to\mathsf{s}:\mathsf{buy}\,.\,\mathsf{a}\to\mathsf{s}:\mathsf{card}(\mathtt{str})\,.\,G' \\ \mathsf{a}\to\mathsf{s}:\mathsf{no}\,.\,0\end{cases} \quad \text{where}$$

$$G' := +\begin{cases}\mathsf{s}\to\mathsf{a}:\mathsf{valid}\,.\,\mathsf{s}\to\mathsf{a}:\mathsf{confirm}\,.\,0 \\ \mathsf{s}\to\mathsf{a}:\mathsf{invalid}\,.\,\mathsf{s}\to\mathsf{a}:\mathsf{cancel}\,.\,0\end{cases}$$

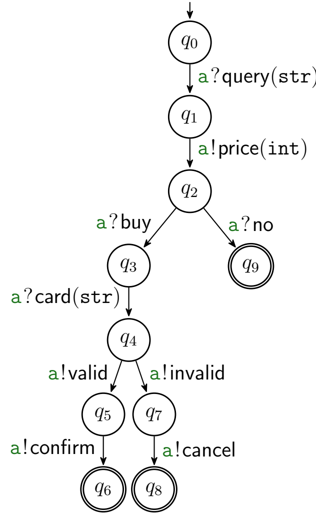


Figure 6.1: Projection of the one buyer protocol onto seller s.

We project G_1 onto s to obtain the state machine in Fig. 6.1 with its set of states $\{q_1, \ldots, q_9\}$. We define the second global type, which specifies the interaction between the seller s and the payment service p, including delegation.

$$G_2 := +\begin{cases} \mathsf{s} \to \mathsf{p} : \mathsf{price}(\mathtt{int}) \,.\, \mathsf{s} \to \mathsf{p} : \mathsf{deleg}(q_3) \,.\, + \begin{cases} \mathsf{p} \to \mathsf{s} : \mathsf{valid}(q_5) \,.\, 0 \\ \mathsf{p} \to \mathsf{s} : \mathsf{invalid}(q_7) \,.\, 0 \end{cases} \\ \mathsf{s} \to \mathsf{p} : \mathsf{no} \,.\, 0 \end{cases}$$

First, the seller delegates checking the card details to the payment service by sending q_3. The payment service then takes care of the payment but we do not specify this here. Afterwards, the payment service delegates control back to the seller: depending on the outcome of the credit card check, they send q_5 or q_7. Starting from there, the seller will either confirm or cancel. This choice is not up to the seller but determined by the label, valid or invalid, sent by the payment service earlier.

We use the states of the projection onto seller s as syntactic marker for the behaviour that is expected from the receiver of that channel endpoint. Usually, this is achieved using local types. In general, local types are less expressive than state machines. However, with Lemma 8.37, we will show that their expressivity coincides for sink-final state machines, i.e. the ones where final states have no outgoing transitions. Hence, any sink-final state machine can be turned into a local type. This, however, corresponds to the initial state and, with delegation, we can also send non-initial states, e.g. q_3. Despite, it appears feasible to first construct a state machine that represents the same behaviour as starting from a non-initial state and, then, the same techniques for constructing a local type apply. For instance, the following local type specifies the behaviour of q_3:

$$\mathsf{s} \triangleleft \mathsf{a} ? \mathsf{card}(\mathtt{str}) \,.\, \oplus \begin{cases} \mathsf{s} \triangleright \mathsf{a} ! \mathsf{valid} \,.\, \mathsf{s} \triangleright \mathsf{a} ! \mathsf{confirm} \,.\, 0 \\ \mathsf{s} \triangleright \mathsf{a} ! \mathsf{invalid} \,.\, \mathsf{s} \triangleright \mathsf{a} ! \mathsf{cancel} \,.\, 0 \end{cases}$$

We have seen an example for delegation with global types. There, in order to talk about states from the subset projections of G_1 in G_2, we had projected G_1 already before defining G_2. We consider it reasonable that one obtains the local behaviours prior to using them in other global types. Therefore, we assume a strict partial order $<$, i.e. it is irreflexive, antisymmetric, and transitive, for the global types under consideration. With its acyclicity, this relation provides means to decide the order in which the global types can be projected: for every G_1 and G_2 in a system, if $G_1 < G_2$, then G_2 can use states from the projection of G_1 so we project G_1 first. We expect that this condition could be worked around with more sophisticated techniques but leave this for future work. Technically, we solely need the existence of $<$ to prove that well-typed processes do not leave messages in channels behind, so-called orphan messages. Interestingly, Scalas et al. [109], allow delegation using local types and do not impose such restrictions. However, they also do not prove the absence of orphan messages. It is unclear whether their type system can be extended to prove the absence of orphan messages without such restrictions.

6.2 Process Calculus

We first define processes and runtime configurations.

Definition 6.2. Processes, runtime configurations and process definitions are defined by the following grammar:

$$
\begin{aligned}
c &::= x \mid s[\mathtt{p}] \\
P &::= 0 \mid P_1 \parallel P_2 \mid (\nu s{:}G)\, P \mid \bigoplus_{i \in I} c[\mathtt{q}_i]!l_i\langle c_i\rangle \,.\, P_i \mid \mathop{\&}_{i \in I} c[\mathtt{q}_i]?l_i(y_i) \,.\, P_i \mid \mathsf{Q}[\vec{c}] \\
R &::= 0 \mid R_1 \parallel R_2 \mid (\nu s{:}G)\, R \mid \bigoplus_{i \in I} c[\mathtt{q}_i]!l_i\langle c_i\rangle \,.\, P_i \mid \mathop{\&}_{i \in I} c[\mathtt{q}_i]?l_i(y_i) \,.\, P_i \mid \mathsf{Q}[\vec{c}] \\
&\quad\;\; \mid s \blacktriangleright \sigma \mid \mathtt{err} \\
\Delta &::= \big(\mathsf{Q}[\vec{x}] = \bigoplus_{i \in I} c[\mathtt{q}_i]!l_i\langle c_i\rangle \,.\, P_i\big);\ \Delta \mid \big(\mathsf{Q}[\vec{x}] = \mathop{\&}_{i \in I} c[\mathtt{q}_i]?l_i(y_i) \,.\, P_i\big);\ \Delta \mid \varepsilon
\end{aligned}
$$

The term c can either be a variable x or a session endpoint of shape $s[\mathtt{p}]$ (which will have type q for some state q). Let us explain the constructors for processes and runtime configurations in more detail. The term 0 denotes termination while $\parallel$ is the parallel operator. With $(\nu s{:}G)$, we restrict a new session s for which the global type G specifies the intended session behaviour. (G is ignored by our reduction semantics and solely used for type-checking.) As for local types, we have internal ($\oplus$) and external ($\&$) choice and assume that $|I| > 0$. For runtime configurations, the use of processes P_i for continuations ensures that we only specify queue contents for active session restrictions. A session restriction is *active* if it is not *guarded* by internal or external actions. $\mathsf{Q}[\vec{c}]$ specifies the use of a process definition with identifier $\mathsf{Q} \in \mathcal{Q}$ and parameters $\vec{c}$ for which Δ provides the definitions. We only consider guarded process definitions (and c

and c_i can be in $\vec{x}$). This allows us to properly distinguish between processes and runtime configurations later: in the reduction rules, we will only add queues for active session restrictions. We assume that Δ has one single definition for every process identifier in $\mathcal{Q}$. Thus, we define $\Delta(\mathsf{Q}, \vec{c})$ as unfolding of the process definition when its variables $\vec{x}$ are substituted by $\vec{c}$. For runtime configurations, $s \blacktriangleright \sigma$ denotes that the queues of session s are currently σ. The function $\sigma \colon \mathcal{P} \times \mathcal{P} \to \mathit{Msg}^*$ where $\mathit{Msg} \ni m ::= l\langle v \rangle$ specifies the queue content where l is from a finite set of labels. If all queues are empty, we exploit notation and use ε, i.e. $\varepsilon(\mathsf{p}, \mathsf{q}) := \varepsilon$. We will use the term `err` to specify if something went wrong.

For global and local types, we defined recursion using μt, which binds a recursion variable t that can be used subsequently. Note that, in our process calculus, recursion can be achieved using process definitions $\mathsf{Q}[\vec{x}]$.

Example 6.3. We give a process P that uses the global types from Example 6.1 and assume base types `Bool`, `str` and `int` as well as a construct for if-then-else for illustrative purposes:

$$
\begin{aligned}
P &:= (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ P_{\mathsf{a}} \parallel P_{\mathsf{s}} \parallel P_{\mathsf{p}} \quad \text{where} \\
P_{\mathsf{a}} &:= s_1[\mathsf{a}][\mathsf{s}]!\mathsf{query}\langle\text{“Alice in Wonderland”}\rangle \,.\, s_1[\mathsf{a}][\mathsf{s}]?\mathsf{price}(p)\,. \\
&\qquad \text{if } p > 10 \\
&\qquad\quad \text{then } s_1[\mathsf{a}][\mathsf{s}]!\mathsf{no}\,.\,0 \\
&\qquad\quad \text{else } s_1[\mathsf{a}][\mathsf{s}]!\mathsf{buy}\,.\, s_1[\mathsf{a}][\mathsf{s}]!\mathsf{card}\langle\text{“1234..., 08/2024, 532”}\rangle\,. \\
&\qquad\qquad\qquad\qquad\qquad \&\begin{cases} s_1[\mathsf{a}][\mathsf{s}]?\mathsf{confirm}\,.\,0 \\ s_1[\mathsf{a}][\mathsf{s}]?\mathsf{cancel}\,.\,0 \end{cases} \\
P_{\mathsf{s}} &:= s_1[\mathsf{s}][\mathsf{a}]?\mathsf{query}(b)\,.\, s_1[\mathsf{s}][\mathsf{a}]!\mathsf{price}\langle \text{prices}[b]\rangle\,. \\
&\qquad\qquad\qquad\qquad \&\begin{cases} s_1[\mathsf{s}][\mathsf{a}]?\mathsf{no}\,.\, s_2[\mathsf{s}][\mathsf{p}]!\mathsf{no}\,.\,0 \\ s_1[\mathsf{s}][\mathsf{a}]?\mathsf{buy}\,.\, s_2[\mathsf{s}][\mathsf{p}]!\mathsf{price}\langle \text{prices}[b]\rangle\,.\, s_2[\mathsf{s}][\mathsf{p}]!\mathsf{deleg}\langle s_1[\mathsf{s}]\rangle\,.\, P'_{\mathsf{s}} \end{cases} \\
P'_{\mathsf{s}} &:= \&\begin{cases} s_2[\mathsf{s}][\mathsf{p}]?\mathsf{valid}(y_1)\,.\, y_1[\mathsf{a}]!\mathsf{confirm}\,.\,0 \\ s_2[\mathsf{s}][\mathsf{p}]?\mathsf{invalid}(y_2)\,.\, y_2[\mathsf{a}]!\mathsf{cancel}\,.\,0 \end{cases} \\
P_{\mathsf{p}} &:= \&\begin{cases} s_2[\mathsf{p}][\mathsf{s}]?\mathsf{no}\,.\,0 \\ s_2[\mathsf{p}][\mathsf{s}]!\mathsf{price}(p)\,.\, s_2[\mathsf{p}][\mathsf{s}]?\mathsf{deleg}\langle y\rangle\,.\, y[\mathsf{a}]?\mathsf{card}(z)\,. \\ \qquad\qquad \text{if is-valid}(z) \text{ then } s_2[\mathsf{p}][\mathsf{s}]!\mathsf{valid}\langle y\rangle\,.\,0 \text{ else } s_2[\mathsf{p}][\mathsf{s}]!\mathsf{invalid}\langle y\rangle\,.\,0 \end{cases}
\end{aligned}
$$

in which $\text{prices}[b]$ denotes a lookup for the price and $\text{is-valid} \colon \texttt{str} \to \texttt{Bool}$ is a function that checks if credit card details are valid. Note the use of variable y in P_{p} for the delegation. In fact, it does not know the endpoint, or local type, it receives but needs to trust that it can perform the respective actions on it. A type system can ensure this. ◀

Definition 6.4. We define a function $\lceil \text{-} \rceil$ to convert a process into a runtime configuration by adding channel types for active sessions:

$$\lceil P_1 \parallel P_2 \rceil := \lceil P_1 \rceil \parallel \lceil P_2 \rceil$$
$$\lceil (\nu s{:}G)\, P \rceil := (\nu s{:}G)\, (P \parallel s \blacktriangleright \varepsilon)$$
$$\lceil P \rceil := P \text{ otherwise}$$

We define structural (pre)congruence. Intuitively, this shows which kind of transformations do not change the meaning of a process or runtime configuration. For instance, parallel composition of P with 0 is basically the same as P itself.

Definition 6.5. For processes, the rules for structural congruence $\equiv$ are the following:

- $P_1 \parallel P_2 \equiv P_2 \parallel P_1$
- $(P_1 \parallel P_2) \parallel P_3 \equiv P_1 \parallel (P_2 \parallel P_3)$
- $P \parallel 0 \equiv P$
- $(\nu s{:}G)\, (\nu s'{:}G')\, P \equiv (\nu s'{:}G')\, (\nu s{:}G)\, P$
- $(\nu s{:}G)\, (P_1 \parallel P_2) \equiv P_1 \parallel (\nu s{:}G)\, P_2$, if s is not free in P_1

We define structural precongruence $\sqsubseteq$ for processes as the smallest precongruence relation that includes $\equiv$ and $(\nu s{:}G)\, 0 \sqsubseteq 0$. For runtime configurations, the rules for structural congruence $\equiv$ are the ones above. We define structural precongruence $\sqsubseteq$ for runtime configurations as the smallest precongruence relation that includes $\equiv$ and $(\nu s{:}G)\, s \blacktriangleright \varepsilon \sqsubseteq 0$.

We only define one direction for the rules $(\nu s{:}G)\, 0 \sqsubseteq 0$ and $(\nu s{:}G)\, s \blacktriangleright \varepsilon \sqsubseteq 0$. This is solely required to prove that structural congruence preserves typability for both processes and runtime configurations (cf. Lemmas 6.18 and 6.19). Intuitively, the other direction would require to impose conditions on the global type G. This treatment is not restrictive in terms of reductions: applying these rules from right to left will not change the possibility for reductions.

Last, we define the reduction rules for our process calculus.

Definition 6.6. For our reduction rules, we first define a reduction context:

$$\mathbb{C} ::= \mathbb{C} \parallel R \;\mid\; R \parallel \mathbb{C} \;\mid\; (\nu s{:}G)\, \mathbb{C} \;\mid\; [\,]$$

Now, we can define the reduction rules:

$$\frac{\Delta(\mathsf{Q},\vec{c}) \parallel R \to R'}{\mathsf{Q}[\vec{c}] \parallel R \to R'}\ \text{RR-Q} \qquad \frac{R \to R'}{\mathbb{C}[R] \to \mathbb{C}[R']}\ \text{RR-CTX}$$

$$\frac{k \in I}{\bigoplus_{i\in I} s[\mathsf{p}][\mathsf{q}_i]!l_i\langle v_i\rangle\,.\,P_i \parallel s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k) \mapsto \vec{m}] \to \lceil P_k \rceil \parallel s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k) \mapsto \vec{m}\cdot l_k\langle v_k\rangle]}\ \text{RR-OUT}$$

$$\frac{k \in I}{\&_{i\in I}\, s[\mathsf{p}][\mathsf{q}_i]?l_i(y_i)\,.\,P_i \parallel s \blacktriangleright \sigma[(\mathsf{q}_k,\mathsf{p}) \mapsto l_k\langle v_k\rangle\cdot\vec{m}] \to \lceil P_k[v_k/y_k] \rceil \parallel s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k) \mapsto \vec{m}]}\ \text{RR-IN}$$

$$\frac{R_1 \sqsubseteq R_1' \qquad R_1' \to R_2' \qquad R_2' \sqsubseteq R_2}{R_1 \to R_2}\ \text{RR-}\sqsubseteq$$

$$\frac{\forall i \in I.\, \sigma(\mathsf{q}_i,\mathsf{p}) = l\langle_\rangle\cdot\vec{m} \text{ and } l_i \neq l}{\&_{i\in I}\, s[\mathsf{p}][\mathsf{q}_i]?l_i(y_i)\,.\,P_i \parallel s \blacktriangleright \sigma \to \texttt{err}}\ \text{RR-ERR1} \qquad \frac{\sigma(\mathsf{p},\mathsf{q}) \neq \varepsilon \text{ for some } \mathsf{p},\mathsf{q}}{(\nu s{:}G)\ s \blacktriangleright \sigma \to \texttt{err}}\ \text{RR-ERR2}$$

The rule RR-Q allows to unfold a process definition while RR-CTX allows to descend for reductions using contexts. Both rules RR-OUT and RR-IN specify how a message is output to a queue or received as input from a queue. RR-$\sqsubseteq$ allows to consider structurally precongruent runtime configurations for reductions. RR-ERR1 yields an error if the next action is to receive but all possible incoming messages do not match any specified label. Last, RR-ERR2 yields an error if a session is over but there are non-empty queues for this session.

Example 6.7. We give a reduction for the process specified in Example 6.3. First, we apply the function $\lceil\text{-}\rceil$ to turn the process into a runtime configuration, yielding

$$R := (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ P_{\mathsf{a}} \parallel P_{\mathsf{s}} \parallel P_{\mathsf{p}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon$$

There is only one possible reduction step: the message $\mathsf{query}\langle$“Alice in Wonderland”$\rangle$ is send by $s_1[\mathsf{a}]$ to $s_1[\mathsf{s}]$. Then, we obtain the following runtime configuration:

$$R' := (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ P_{\mathsf{a}}' \parallel P_{\mathsf{s}} \parallel P_{\mathsf{p}} \\ \parallel s_1 \blacktriangleright \varepsilon[(\mathsf{a},\mathsf{s}) \mapsto \mathsf{query}\langle\text{“Alice in Wonderland”}\rangle] \parallel s_2 \blacktriangleright \varepsilon$$

where

$$\begin{aligned} P_{\mathsf{a}}' := {} & s_1[\mathsf{a}][\mathsf{s}]?\mathsf{price}(p)\,. \\ & \quad \text{if } p > 10 \\ & \quad\ \text{then } s_1[\mathsf{a}][\mathsf{s}]!\mathsf{no}\,.\,0 \\ & \quad\ \text{else } s_1[\mathsf{a}][\mathsf{s}]!\mathsf{buy}\,.\,s_1[\mathsf{a}][\mathsf{s}]!\mathsf{card}\langle\text{“1234..., 08/2024, 111”}\rangle\,.\,\&\begin{cases} s_1[\mathsf{a}][\mathsf{s}]?\mathsf{confirm}\,.\,0 \\ s_1[\mathsf{a}][\mathsf{s}]?\mathsf{cancel}\,.\,0 \end{cases} \end{aligned}$$

Despite dealing with runtime configurations, we can specify processes because the queues are specified at top level. ◀

6.3 Type System for Processes and Runtime Configurations

Typing for base types is well-understood. Thus, we focus on the more difficult case of delegation, following work by Scalas et al. [109]. Integration of base types is mostly orthogonal and would distract from the main concerns here so we briefly remark differences in the treatment of base types after presenting our type system.

For processes, we have two typing contexts: Θ and Λ. We consider states as syntactic markers for local specifications so we use L as type for such payloads. So far, we only considered a fixed set of participants $\mathcal{P}$. In a system with multiple global types, we write $\mathcal{P}_G$ to denote the subset of participants of G and Chan_G for the respective channels. We might also use the session s instead of the respective global type.

Prior to giving definitions for our type system, let us remark that we make use of the *Barendregt Variable Convention* [12], which assumes that the names of bound variables is always distinct from the ones of free variables. This allows us not to explicitly rename variables and simplifies the formalisation both for writing and reading.

Definition 6.8. The process definition typing context Θ is a function from process identifiers to types for its parameters: $\Theta \colon \mathcal{Q} \to \vec{L}$. A *syntactic typing context* is defined by the following grammar:

$$\Lambda ::= \Lambda, s[\mathrm{p}]\!:\!L \ \mid\ \Lambda, x\!:\!L \ \mid\ \emptyset$$

A syntactic typing context is a *typing context* if every element has at most one type. Here, we do only consider typing contexts. We consider typing contexts to be equivalent up to reordering and, thus, we may also treat them as mappings. We use notation $\{\Lambda_i\}_{i \in I}$ to denote that we split Λ into $|I|$ typing contexts.

Equipped with these typing contexts, we can give the typing rules for processes. The first two rules solely deal with the process definition typing context, which provides the types for process definitions. The other rules deal with the different constructs of our process calculus and how to type them. Most importantly, our type system ensures that all information in the typing context is used exactly once.

Definition 6.9. We define $\text{end}(q)$ to hold when q is final and does not have outgoing receive transitions. The typing rules for processes are the following:

$$\frac{}{\vdash \varepsilon : \Theta}\;\text{PT-DEF-}\varepsilon \qquad \frac{\Theta \mid \vec{x}:\vec{L} \vdash P \qquad \Theta(\mathsf{Q}) = \vec{L}}{\vdash (\mathsf{Q}[\vec{x}] = P); \Delta : \Theta}\;\text{PT-DEF} \qquad \frac{\Theta(\mathsf{Q}) = \vec{L}}{\Theta \mid \vec{c}:\vec{L} \vdash \mathsf{Q}[\vec{c}]}\;\text{PT-Q}$$

$$\frac{}{\Theta \mid \emptyset \vdash 0}\;\text{PT-0} \qquad \frac{\Theta \mid \Lambda \vdash P \qquad \text{end}(q)}{\Theta \mid c{:}q, \Lambda \vdash P}\;\text{PT-END} \qquad \frac{\Theta \mid \Lambda_1 \vdash P_1 \qquad \Theta \mid \Lambda_2 \vdash P_2}{\Theta \mid \Lambda_1, \Lambda_2 \vdash P_1 \parallel P_2}\;\text{PT-}\parallel$$

$$\frac{\delta(q) \supseteq \{(\mathsf{p} \triangleright \mathsf{q}_i ! l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \Lambda, c{:}q_i, \{c_j : L_j\}_{j \in I \setminus \{i\}} \vdash P_i}{\Theta \mid \Lambda, c{:}q, \{c_i : L_i\}_{i \in I} \vdash \bigoplus_{i \in I} c[\mathsf{q}_i]!l_i\langle c_i \rangle \,.\, P_i}\;\text{PT-}\oplus$$

$$\frac{\delta(q) = \{(\mathsf{p} \triangleleft \mathsf{q}_i ? l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \Lambda, c{:}q_i, y_i : L_i \vdash P_i}{\Theta \mid \Lambda, c{:}q \vdash \&_{i \in I}\, c[\mathsf{q}_i]?l_i(y_i) \,.\, P_i}\;\text{PT-}\&$$

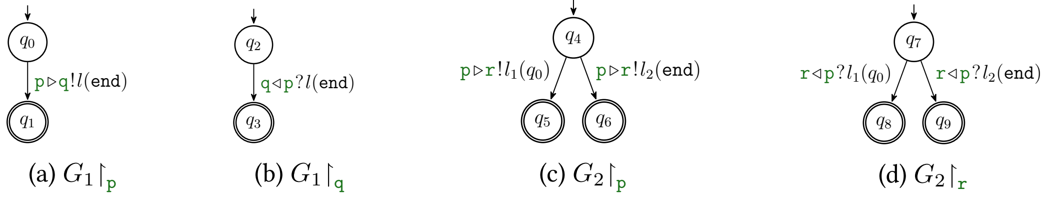
$$\frac{\Lambda_s = \{s[\mathsf{p}] : \text{init}(\mathscr{P}(G, \mathsf{p}))\}_{\mathsf{p} \in \mathcal{P}_G} \qquad \Theta \mid \Lambda, \Lambda_s \vdash P}{\Theta \mid \Lambda \vdash (\nu s{:}G)\, P}\;\text{PT-}\nu$$

where $\text{init}(\text{-})$ denotes the initial state of a state machine.

The rules PT-DEF-ε and PT-DEF ensure that the process definition typing context provides the right types for parameters. This is then used to type process definitions in a process, using PT-Q. PT-0 types 0 with an empty second typing context while PT-END allows to remove type bindings $c{:}q$ where q is a final state without outgoing receive transitions. The rules PT-$\oplus$ and PT-$\&$ can be used to type internal and external choice. The rule PT-$\parallel$ allows to split the typing contexts and type the respective processes independently. Last, PT-ν adds type bindings for a session s and requires that the remaining process is typed using this.

Our type system is *linear*, i.e. it requires that every type binding is considered and they can only be dropped if they correspond to final states without outgoing receive transitions. This ensures that all the actions specified by the global type are actually taken and the participants of a session cannot stop earlier.

It might seem that p in PT-$\oplus$ and PT-$\&$ is unbound but since we assume q to be distinct across all state machines under consideration, it is clear from context. Let us explain PT-$\oplus$ in more detail. To type $\Theta \mid \Lambda, c{:}q, \{c_i : L_i\}_{i \in I} \vdash \bigoplus_{i \in I} c[\mathsf{q}_i]!l_i\langle c_i \rangle \,.\, P_i$, we require all send actions to be possible from q, i.e. $\delta(q) \supseteq \{(\mathsf{p} \triangleright \mathsf{q}_i ! l_i(L_i), q_i) \mid i \in I\}$. We do not require all of them to be possible though, in contrast to the receive actions in PT-$\&$. In addition, for every $i \in I$, we require $\Theta \mid \Lambda, c{:}q_i, \{c_j : L_j\}_{j \in I \setminus \{i\}} \vdash P_i$, which gives c the new binding q_i and removes the type binding for the payload $c_i : L_i$. Intuitively, it is transferred when sending a message. Thus, in its counterpart PT-$\&$, the payload's type will be used to type the continuation after receiving it, i.e. $y_i : L_i$.



(a) $G_1\restriction_{\mathtt{p}}$ (b) $G_1\restriction_{\mathtt{q}}$ (c) $G_2\restriction_{\mathtt{p}}$ (d) $G_2\restriction_{\mathtt{r}}$

Figure 6.2: Projections of two global types onto its participants.

Note that the assumption that we only consider typing contexts, and not syntactic typing contexts, ensures that Λ in the conclusion of PT-ν does not contain any s for instance. The same will hold for the typing rules for runtime configurations.

Remark 6.10 ($\mathrm{end}(\text{-})$ and final non-sink states)**.** We use $\mathrm{end}(q)$ to check if q is a final state without outgoing receive transitions. (Recall that send validity entails that there cannot be final states with outgoing send transitions in a subset projection.) Following standard MST frameworks, we would simply require q to be final. However, they consider sink-final implementations only. We elaborated in Section 5.5.2 that our subset projection is sink-final for G if there exists a sink-final implementation for G, and hence provides a solution to the soft implementability problem. Thus, this does not impose a limitation for our method. The condition $\mathrm{end}(q)$ amounts to checking if a state is final if all local implementations are sink-final.

Example 6.11. Let us type delegation with an example. We consider

$$G_1 := \mathtt{p} \to \mathtt{q} : l(\mathtt{end}) \,.\, 0$$

for which we could model the payload of $l(\mathtt{end})$ with an arbitrary state *end* such that $\mathrm{end}(\mathit{end})$. For readability, we omit its treatment in the typing derivations. We project G_1 onto its participants and obtain the FSMs in Fig. 6.2a and Fig. 6.2b. Using the states of these, we define delegation in the second global type:

$$G_2 := +\begin{cases} \mathtt{p} \to \mathtt{r} : l_1(q_0) \,.\, 0 \\ \mathtt{p} \to \mathtt{r} : l_2(\mathtt{end}) \,.\, 0 \end{cases}$$

Its projections are given in Fig. 6.2c and Fig. 6.2d. Let us define a process that uses both global types:

$$P := (\nu s_1 : G_1)(\nu s_2 : G_2)\; P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \text{ where}$$

$$P_{\mathtt{p}} := \oplus\begin{cases} s_1[\mathtt{p}][\mathtt{r}]!l_1\langle s_1[\mathtt{p}]\rangle \,.\, 0 \\ s_1[\mathtt{p}][\mathtt{r}]!l_2\langle \mathit{end}\rangle \,.\, s_1[\mathtt{p}][\mathtt{q}]!l\langle \mathit{end}\rangle \,.\, 0 \end{cases}$$

$$P_{\mathtt{q}} := s_1[\mathtt{q}][\mathtt{p}]?l(x) \,.\, 0$$

$$P_{\mathtt{r}} := \&\begin{cases} s_2[\mathtt{r}][\mathtt{p}]?l_1(x) \,.\, x[\mathtt{q}]!l\langle \mathit{end}\rangle \,.\, 0 \\ s_2[\mathtt{r}][\mathtt{p}]?l_2(x) \,.\, 0 \end{cases}$$

In this example, the process definition typing context is always empty so we omit it. Because of space constraints, we give the typing derivation in pieces. We use numbers (0) to (5) to refer to the typing derivation for the respective branch. Still, it should be read from bottom to top, starting from (0). We start with the initial part until typing we arrive at typing the parallel composition.

$$\dfrac{\dfrac{(3)}{s_1[\mathtt{p}]{:}q_0, s_2[\mathtt{p}]{:}q_4 \vdash P_{\mathtt{p}}} \qquad \dfrac{(4)}{s_1[\mathtt{q}]{:}q_2 \vdash P_{\mathtt{q}}} \qquad \dfrac{(5)}{s_2[\mathtt{r}]{:}q_7 \vdash P_{\mathtt{r}}}}{(2):\quad \Lambda_{s_1}, \Lambda_{s_2} \vdash P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}}}\ \text{PT-}\parallel\ (\textsc{twice})$$

$$\dfrac{\Lambda_{s_2} = s_2[\mathtt{p}]{:}q_4, s_2[\mathtt{r}]{:}q_7 \qquad \dfrac{(2)}{\Lambda_{s_1}, \Lambda_{s_2} \vdash P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}}}}{(1):\quad \Lambda_{s_1} \vdash (\nu s_2{:}G_2)\ P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}}}\ \text{PT-}\nu$$

$$\dfrac{\Lambda_{s_1} = s_1[\mathtt{p}]{:}q_0, s_1[\mathtt{q}]{:}q_2 \qquad \dfrac{(1)}{\Lambda_{s_1} \vdash (\nu s_2{:}G_2)\ P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}}}}{(0):\quad \emptyset \vdash (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}}}\ \text{PT-}\nu$$

We apply the rule for restrictions PT-ν and then the one for parallel composition PT-‖. Let us give the typing derivations for the individual branches.

$$\text{PT-}\textsc{end}\ \dfrac{\text{PT-}\oplus\ \dfrac{\delta(q_0) = \{(\mathtt{p}\triangleright\mathtt{q}!l(\mathtt{end}), q_1)\} \qquad \text{PT-}\textsc{end}\ \dfrac{\text{PT-0}\ \dfrac{}{\emptyset \vdash 0} \qquad \mathrm{end}(q_1)}{s_1[\mathtt{p}]{:}q_1 \vdash 0}}{s_1[\mathtt{p}]{:}q_0 \vdash s_1[\mathtt{p}][\mathtt{q}]!l\langle \mathit{end}\rangle\,.\,0}}{s_1[\mathtt{p}]{:}q_0, s_2[\mathtt{p}]{:}q_6 \vdash s_1[\mathtt{p}][\mathtt{q}]!l\langle \mathit{end}\rangle\,.\,0}$$

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 0}\ \text{PT-0} \qquad \mathrm{end}(q_5)}{s_2[\mathtt{p}]{:}q_5 \vdash 0}\ \text{PT-}\textsc{end} \qquad \delta(q_4) = \{(\mathtt{p}\triangleright\mathtt{r}!l_1(q_0), q_5), (\mathtt{p}\triangleright\mathtt{r}!l_2(\mathtt{end}), q_6),\}}{(3):\quad s_1[\mathtt{p}]{:}q_0, s_2[\mathtt{p}]{:}q_4 \vdash P_{\mathtt{p}}}\ \text{PT-}\oplus$$

$$\dfrac{\delta(q_2) = \{(\mathtt{q}\triangleleft\mathtt{p}?l(\mathtt{end}), q_3)\} \qquad \dfrac{\text{PT-0}\ \dfrac{}{\emptyset \vdash 0} \qquad \mathrm{end}(q_3)}{s_1[\mathtt{q}]{:}q_3 \vdash 0}\ \text{PT-}\textsc{end}}{(4):\quad s_1[\mathtt{q}]{:}q_2 \vdash P_{\mathtt{q}}}\ \text{PT-}\&$$

$$
\text{PT-END}\;\dfrac{\text{PT-}\oplus\;\dfrac{\delta(q_0) = \{(\mathtt{p}\triangleright\mathtt{q}!l(\mathtt{end}), q_1)\} \qquad \text{PT-END}\;\dfrac{\text{PT-0}\;\dfrac{}{\emptyset \vdash 0} \qquad \mathrm{end}(q_1)}{x\!:\!q_1 \vdash 0}}{x\!:\!q_0 \vdash x[\mathtt{q}]!l\langle \mathit{end}\rangle\,.\,0} \qquad \mathrm{end}(q_8)}{s_2[\mathtt{r}]\!:\!q_8, x\!:\!q_0 \vdash x[\mathtt{q}]!l\langle \mathit{end}\rangle\,.\,0}
$$

$$
\dfrac{\text{PT-END}\;\dfrac{\dfrac{}{\emptyset \vdash 0}\;\text{PT-0} \qquad \mathrm{end}(q_9)}{s_2[\mathtt{r}]\!:\!q_9 \vdash 0} \qquad \delta(q_7) = \{(\mathtt{r}\triangleleft\mathtt{p}?l_1(q_0), q_8), (\mathtt{r}\triangleleft\mathtt{p}?l_2(\mathtt{end}), q_9)\}}{(5):\quad s_2[\mathtt{r}]\!:\!q_7 \vdash P_{\mathtt{r}}}\;\text{PT-\&}
$$

◀

During runtime, we have queues for each session. For these, we define queue types and use them in queue typing contexts.

Definition 6.12. Queue types are defined by the following grammar:

$$\gamma ::= l(L) \cdot \gamma \;\mid\; \varepsilon$$

A *syntactic queue typing context* is defined by the following grammar:

$$\Omega ::= \Omega, s[\mathtt{p}][\mathtt{q}] :: \gamma \;\mid\; \emptyset$$

A syntactic queue typing context is a *queue typing context* if every element has at most one type. Here, we do only consider queue typing contexts. We consider queue typing contexts to be equivalent up to reordering and, thus, we may also treat them as mappings.

While the (second) typing context specifies states for each participant, the queue typing context specifies the content of the queues. This is all we need to define reductions following the respective communicating state machine.

Definition 6.13. We define the reductions for typing contexts as follows:

$$
\dfrac{q \xrightarrow{\mathtt{p}\triangleright\mathtt{q}!l(L)} q'}{s[\mathtt{p}]\!:\!q, \Lambda \mid s[\mathtt{p}][\mathtt{q}] :: \gamma, \Omega \to s[\mathtt{p}]\!:\!q', \Lambda \mid s[\mathtt{p}][\mathtt{q}] :: \gamma \cdot l(L), \Omega}\;\text{TR-}\oplus
$$

$$
\dfrac{q \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}?l(L)} q'}{s[\mathtt{p}]\!:\!q, \Lambda \mid s[\mathtt{q}][\mathtt{p}] :: l(L) \cdot \gamma, \Omega \to s[\mathtt{p}]\!:\!q', \Lambda \mid s[\mathtt{q}][\mathtt{p}] :: \gamma, \Omega}\;\text{TR-\&}
$$

These rules mimic exactly the semantics of communicating state machines.

We show that reductions for typing contexts are preserved when adding type bindings to the typing contexts.

Lemma 6.14. Let Λ_1, Λ_1', and Λ_2 be typing contexts and Ω_1, Ω_1', and Ω_2 be queue typing contexts. If $\Lambda_1 \mid \Omega_1 \to \Lambda_1' \mid \Omega_1'$, then $\Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \to \Lambda_1', \Lambda_2 \mid \Omega_1', \Omega_2$.

Proof. We do inversion on $\Lambda_1 \mid \Omega_1 \to \Lambda_1' \mid \Omega_1'$, yielding two cases. First, we have

$$\frac{q \xrightarrow{\mathsf{p} \triangleright \mathsf{q} ! l(L)} q'}{s[\mathsf{p}]{:}q, \hat{\Lambda}_1 \mid s[\mathsf{p}][\mathsf{q}] :: \gamma, \hat{\Omega}_1 \to s[\mathsf{p}]{:}q', \hat{\Lambda}_1 \mid s[\mathsf{p}][\mathsf{q}] :: \gamma \cdot l(L), \hat{\Omega}_1} \text{ TR-}\oplus$$

With this, it is obvious that the following holds:

$$\frac{q \xrightarrow{\mathsf{p} \triangleright \mathsf{q} ! l(L)} q'}{s[\mathsf{p}]{:}q, \hat{\Lambda}_1, \Lambda_2 \mid s[\mathsf{p}][\mathsf{q}] :: \gamma, \hat{\Omega}_1, \Omega_2 \to s[\mathsf{p}]{:}q', \hat{\Lambda}_1, \Lambda_2 \mid s[\mathsf{p}][\mathsf{q}] :: \gamma \cdot l(L), \hat{\Omega}_1, \Omega_2} \text{ TR-}\oplus$$

which is precisely what we have to show. The case for TR-& is analogous and, therefore, omitted. ☐

After this small intermezzo on reductions for typing contexts, we now define the typing rules for runtime configurations.

Definition 6.15. The typing rules for runtime configurations are the following:

$$\frac{}{\vdash \varepsilon{:}\Theta} \text{ RT-DEF-}\varepsilon \qquad \frac{\Theta \mid \vec{x}{:}\vec{L} \vdash P \qquad \Theta(\mathsf{Q}) = \vec{L}}{\vdash (\mathsf{Q}[\vec{x}] = P); \Delta{:}\Theta} \text{ RT-DEF} \qquad \frac{\Theta(\mathsf{Q}) = \vec{L}}{\Theta \mid \vec{c}{:}\vec{L} \mid \emptyset \vdash \mathsf{Q}[\vec{c}]} \text{ RT-Q}$$

$$\frac{}{\Theta \mid \emptyset \mid \emptyset \vdash 0} \text{ RT-0} \qquad \frac{\Theta \mid \Lambda \mid \Omega \vdash R \qquad \text{end}(q)}{\Theta \mid \Lambda, c{:}q \mid \Omega \vdash R} \text{ RT-END}$$

$$\frac{\Theta \mid \Lambda_1 \mid \Omega_1 \vdash R_1 \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R_1 \parallel R_2} \text{ RT-}\parallel$$

$$\frac{\delta(q) \supseteq \{(\mathsf{p} \triangleright \mathsf{q}_i ! l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \Lambda, c{:}q_i, \{c_j{:}L_j\}_{j \in I \setminus \{i\}} \mid \Omega \vdash P_i}{\Theta \mid \Lambda, c{:}q, \{c_i{:}L_i\}_{i \in I} \mid \Omega \vdash \bigoplus_{i \in I} c[\mathsf{q}_i]!l_i\langle c_i \rangle \,.\, P_i} \text{ RT-}\oplus$$

$$\frac{\delta(q) = \{(\mathsf{p} \triangleleft \mathsf{q}_i ? l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \Lambda, y_i{:}L_i, c{:}q_i \mid \Omega \vdash P_i}{\Theta \mid \Lambda, c{:}q \mid \Omega \vdash \mathop{\&}_{i \in I} c[\mathsf{q}_i]?l_i(y_i) \,.\, P_i} \text{ RT-\&}$$

$$\frac{\Lambda_s = \{s[\mathsf{p}]{:}\vec{q}_{\mathsf{p}}\}_{\mathsf{p} \in \mathcal{P}_G} \qquad \begin{array}{c}(\vec{q}, \xi) \in \text{reach}(\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p} \in \mathcal{P}_G}) \\ \Omega_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p}, \mathsf{q})\}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_G}\end{array} \qquad \Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash R}{\Theta \mid \Lambda \mid \Omega \vdash (\nu s{:}G)\, R} \text{ RT-}\nu$$

$$\frac{}{\Theta \mid \emptyset \mid \{s[\mathsf{p}][\mathsf{q}] :: \varepsilon\}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_s} \vdash s \blacktriangleright \varepsilon} \text{ RT-EmptyQueue}$$

$$\frac{\Theta \mid \Lambda \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto \vec{m}]}{\Theta \mid \Lambda, v{:}L \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: l(L) \cdot \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto l\langle v \rangle \cdot \vec{m}]}\ \text{RT-Queue}$$

where $\mathrm{reach}(\text{-})$ denotes the set of reachable configurations of the given CSM. Most rules are analogous to the rules for processes. For RT-ν, though, we do not require the CSM configuration to be initial but just reachable, yielding typability of runtime configurations during execution. The rules for queues are standard: RT-Queue allows to type queues from the first to the last message in the queue while RT-EmptyQueue types empty queues.

Example 6.16. In Example 6.11, we gave a typing derivation for a process P using delegation. It is straightforward that this typing derivation can be mimicked for $\lceil P \rceil$. Here, we want to give a typing derivation after one reduction step, for the case where delegation happens. We have

$$R' := (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ 0 \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]\ .$$

We give a typing derivation for $\emptyset \mid \emptyset \vdash R'$, with label (0). Again, we omit the process definition typing context since it is empty throughout.

$$\text{RT-EmptyQueue}\ \frac{}{\emptyset \mid \Omega_1 \vdash s_1 \blacktriangleright \varepsilon}$$

$$\frac{(6)}{s_1[\mathtt{p}]{:}q_0, \mid \Omega_2 \vdash s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]}$$

$$\frac{\dfrac{(3)}{s_2[\mathtt{p}]{:}q_5 \mid \emptyset \vdash 0} \qquad \dfrac{(4)}{s_1[\mathtt{q}]{:}q_2 \mid \emptyset \vdash P_{\mathtt{q}}} \qquad \dfrac{(5)}{s_2[\mathtt{r}]{:}q_7 \mid \emptyset \vdash P_{\mathtt{r}}}}{(2):\quad \Lambda_{s_1}, \Lambda'_{s_2} \mid \Omega_{s_1}, \Omega'_{s_2} \vdash 0 \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]}\ \text{PT-}\parallel\ \text{(4 times)}$$

$$\frac{\dfrac{(2)}{\Lambda_{s_1}, \Lambda'_{s_2} \mid \Omega_{s_1}, \Omega'_{s_2} \vdash 0 \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]} \quad \Lambda'_{s_2} = s_2[\mathtt{p}]{:}q_5, s_2[\mathtt{r}]{:}q_7 \qquad \Omega'_{s_2} = s_2[\mathtt{p}][\mathtt{r}] :: l_1(q_0), s_2[\mathtt{r}][\mathtt{p}] :: \varepsilon}{(1):\quad \Lambda_{s_1} \mid \Omega_{s_1} \vdash (\nu s_2{:}G_2)\ P_{\mathtt{p}} \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]}\ \text{RT-}\nu$$

$$\frac{\dfrac{(1)}{\Lambda_{s_1} \mid \Omega_{s_1} \vdash (\nu s_2{:}G_2)\ 0 \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]} \quad \Lambda_{s_1} = s_1[\mathtt{p}]{:}q_0, s_1[\mathtt{q}]{:}q_2 \qquad \Omega_{s_1} = s_1[\mathtt{p}][\mathtt{q}] :: \varepsilon, s_1[\mathtt{q}][\mathtt{p}] :: \varepsilon}{(0):\quad \emptyset \mid \emptyset \vdash (\nu s_1{:}G_1)(\nu s_2{:}G_2)\ 0 \parallel P_{\mathtt{q}} \parallel P_{\mathtt{r}} \parallel s_1 \blacktriangleright \varepsilon \parallel s_2 \blacktriangleright \varepsilon[(\mathtt{p},\mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]}\ \text{RT-}\nu$$

The typing derivations for (4) and (5) are analogous to the ones in Example 6.11. The typing derivation for (3) is straightforward with RT-END and RT-0, similar to what we presented for (3) in Example 6.11. We give the typing derivation for (6):

$$\dfrac{\emptyset \mid s_2[\mathtt{p}][\mathtt{r}] :: \varepsilon, s_2[\mathtt{r}][\mathtt{p}] :: \varepsilon \vdash s_2 \blacktriangleright \varepsilon}{(6): \quad s_1[\mathtt{p}]{:}q_0, \mid s_2[\mathtt{p}][\mathtt{r}] :: l_1(q_0), s_2[\mathtt{r}][\mathtt{p}] :: \varepsilon \vdash s_2 \blacktriangleright \varepsilon[(\mathtt{p}, \mathtt{r}) \mapsto l_1\langle s_1[\mathtt{p}]\rangle]} \text{ RT-QUEUE}$$

◀

The presentation of our type system is inspired by work from Scalas et al. [109]. Thus, we want to highlight key differences. First, we handle sender-driven choice, which allows to send to different receivers and to receive from different senders, while they only consider directed choice. In fact, our processes can choose between different options when sending messages while their work restricts to a single choice. However, their treatment could be combined with control flow like if-then-else to cover more scenarios. Also, Scalas et al. [109] employ subtyping similar to what we do for send actions so global types still can have multiple branches when sending. We refer to Section 6.5 for an explanation why we do not employ subtyping for receives in our type system. Second, Scalas et al. [109] do not consider a global protocol specification but only local types. Based on these local types, they prove properties using model checking. Third, Scalas et al. [109] do only consider one error scenario: a participant would like to receive something but the first message in the respective queue does not match. We generalise this scenario to our setting and require that the first message in all respective queues does not match. In addition, we consider the error case where a session ended with non-empty queues. In the next section, we will prove that our type system prevents both these scenarios.

Remark 6.17 (Adding base types)**.** The treatment of base types and expressions in type systems is well-understood. Hence, it should be straightforward to extend our type system to add expressions. More specifically, we would allow to send the result of expressions and, hence, variables can be bound to values. Provided with (Boolean) expressions, it is also standard to add features of control-flow like if-then-else. For most flexible use of our results, it would make most sense if a user provided a type system for the expressions and base types they need. Then, the type system would use what we defined for local types (and hence delegation) and the provided type system for expressions. There is one important difference between both type systems. While ours is linear, the one for expressions does not need to be. They usually allow to duplicate and drop type bindings from the typing context, using rules called *contraction* and *weakening*.

6.4 Soundness of Type System

For conciseness, we assume a process definition typing context Θ, typing contexts Λ, Λ_1, Λ_2, ..., and queue typing contexts Ω, Ω_1, Ω_2, ... in this section.

We presented a type system for processes and runtime configurations. While we closed the reduction semantics under structural precongruence $\sqsubseteq$, we have not stated the respective rules for our type system:

$$\frac{\Theta \mid \Lambda \vdash P \qquad P \sqsubseteq P'}{\Theta \mid \Lambda \vdash P'} \text{PT-}\sqsubseteq \qquad \frac{\Theta \mid \Lambda \mid \Omega \vdash R \qquad R \sqsubseteq R'}{\Theta \mid \Lambda \mid \Omega \vdash R'} \text{RT-}\sqsubseteq$$

We show that these rules are admissible, i.e. they can be added without changing the capabilities of the type system. This allows us not to consider these rules in the following proofs but still use them if convenient.

Lemma 6.18 (Admissibility of structural precongruence for runtime configuration typing)**.** If $\Theta \mid \Lambda \mid \Omega \vdash R_1$ and $R_1 \sqsubseteq R_2$, then $\Theta \mid \Lambda \mid \Omega \vdash R_2$.

Proof. We first consider the cases for structural congruence $\equiv$ and then the additional ones for structural precongruence. We do a case analysis on $\equiv$ and reason for both directions. Subsequently, we consider the two rules for $\sqsubseteq$.

- $R_1 \parallel R_2 \equiv R_2 \parallel R_1$:
  By inversion, we know that RT-$\parallel$ is the first rule applied in the typing derivation. This rule is symmetric so basically the typing derivation works.
- $(R_1 \parallel R_2) \parallel R_3 \equiv R_1 \parallel (R_2 \parallel R_3)$:
  By inversion, we know that RT-$\parallel$ is the first and second rule applied in the typing derivation. It is easy to see that the typing derivation can be rearranged to match the structure.
- $R \parallel 0 \equiv R$:
  First, assume that there is a typing derivation

$$\frac{\Theta \mid \Lambda_1 \mid \Omega_1 \vdash R \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash 0}{\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R \parallel 0} \text{RT-}\parallel$$

  We show there is a typing derivation $\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R$. Inversion yields that two rules can be applied for the given typing derivation $\Theta \mid \Lambda_2 \mid \Omega_2 \vdash 0$: RT-END and RT-0. Thus, it follows that $\Omega_2 = \emptyset$. Also, $\Lambda_2 = \{s[\mathtt{p}]{:}\vec{q}_{\mathtt{p}}\}_{\mathtt{p} \in S}$ for some set of participants S and $\text{end}(\vec{q}_{\mathtt{p}})$ for every $\mathtt{p} \in S$. By inversion, there is a typing derivation for $\Theta \mid \Lambda_1 \mid \Omega_1 \vdash R$. With $\Omega_2 = \emptyset$, it remains to show that there is a typing derivation $\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1 \vdash R$. The only difference is the typing context Λ_2. This, however, can be taken care of using RT-END as in the other typing derivation, concluding this case.
  Second, assume there is a typing derivation for R. We show there is a typing derivation for $\Theta \mid \Lambda \mid \Omega \vdash R \parallel 0$. We first apply RT-$\parallel$ to obtain

$$\frac{\Theta \mid \Lambda \mid \Omega \vdash R \qquad \dfrac{}{\Theta \mid \emptyset \mid \emptyset \vdash 0}\text{RT-0}}{\Theta \mid \Lambda \mid \Omega \vdash R \parallel 0} \text{RT-}\parallel$$

for which the right premise is met with RT-0 and the left premise is given by assumption.

- $(\nu s\!:\!G)\,(\nu s'\!:\!G')\,R \equiv (\nu s'\!:\!G')\,(\nu s\!:\!G)\,R$:
By inversion, both typing derivations need to apply RT-ν twice in the beginning. It is straightforward that both rule applications do not interfere with each other, yielding the same premise to prove:

$$\Theta \mid \Lambda, \Lambda_s, \Lambda_{s'} \mid \Omega, \Omega_s, \Omega_{s'} \vdash R$$

Thus, this is given by assumption.

- $(\nu s\!:\!G)\,(R_1 \parallel R_2) \equiv R_1 \parallel (\nu s\!:\!G)\,R_2$ and s is not free in R_1:
First, we assume there is a typing derivation for $(\nu s\!:\!G)\,(R_1 \parallel R_2)$ and show there is a typing derivation for $R_1 \parallel (\nu s\!:\!G)\,R_2$. Applying inversion twice yields

$$\dfrac{\dfrac{\Theta \mid \Lambda_1 \mid \Omega_1 \vdash R_1 \qquad \Theta \mid \Lambda_2, \Lambda_s \mid \Omega_2, \Omega_s \vdash R_2}{\Theta \mid \Lambda_1, \Lambda_2, \Lambda_s \mid \Omega_1, \Omega_2, \Omega_s \vdash R_1 \parallel R_2}\,\text{RT-}\parallel \qquad (\vec{q}, \xi) \in \text{reach}(\{\!\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\!\}_{\mathsf{p} \in \mathcal{P}_G}) \\ \Lambda_s = \{ s[\mathsf{p}] : \vec{q}_{\mathsf{p}} \}_{\mathsf{p} \in \mathcal{P}_G} \qquad \Omega_s = \{ s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p}, \mathsf{q}) \}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_G}}{\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash (\nu s\!:\!G)\,(R_1 \parallel R_2)}\,\text{RT-}\nu$$

where $\Lambda = \Lambda_1, \Lambda_2$ and $\Omega = \Omega_1, \Omega_2$. We claim we can assume that Λ_s and Ω_s are used in the typing derivation for $\Theta \mid \Lambda_2, \Lambda_s \mid \Omega_2, \Omega_s \vdash R_2$. By definition, these only contain type bindings related to s, which does not occur in R_1 by assumption. There might exist a typing derivation where parts of Λ_s or Ω_s appear in the typing derivation for R_1 but these can only removed with the rules RT-END (not even with RT-EMPTYQUEUE since this requires $s \blacktriangleright \varepsilon$). Hence, such derivations can be mimicked in the typing derivation for R_2, justifying our treatment of Λ_s and Ω_s. We construct a typing derivation:

$$\dfrac{\dfrac{(\vec{q}, \xi) \in \text{reach}(\{\!\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\!\}_{\mathsf{p} \in \mathcal{P}_G}) \qquad \Lambda_s = \{ s[\mathsf{p}] : \vec{q}_{\mathsf{p}} \}_{\mathsf{p} \in \mathcal{P}_G} \\ \Omega_s = \{ s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p}, \mathsf{q}) \}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_G} \qquad \Theta \mid \Lambda_2, \Lambda_s \mid \Omega_2, \Omega_s \vdash R_2}{\Theta \mid \Lambda_2 \mid \Omega_2 \vdash (\nu s\!:\!G)\,R_2}\,\text{RT-}\nu \\ \Theta \mid \Lambda_1 \mid \Omega_1 \vdash R_1}{\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R_1 \parallel (\nu s\!:\!G)\,R_2}\,\text{RT-}\parallel$$

All premises coincide with the ones of the original typing derivation, concluding this case.
Second, we assume there is a typing derivation for $R_1 \parallel (\nu s\!:\!G)\,R_2$ and show there is a typing derivation for $(\nu s\!:\!G)\,(R_1 \parallel R_2)$. The proof is analogous to the previous case but we do not need to reason about the treatment of Λ_s and Ω_s but it suffices to show there is one typing and we can choose the respective treatment.

- $(\nu s\!:\!G)\,s \blacktriangleright \varepsilon \sqsubseteq 0$:
We assume there is a typing derivation for $\Theta \mid \Lambda \mid \Omega \vdash (\nu s\!:\!G)\,s \blacktriangleright \varepsilon$. We show

there is a typing derivation for $\Theta \mid \Lambda \mid \Omega \vdash 0$. By inversion, we know that RT-ν is the last rule to be applied and we get one of the premises:

$$\Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash s \blacktriangleright \varepsilon$$

with $\Lambda_s = \{s[\mathtt{p}]:\vec{q}_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}_G}$ and $\Omega_s = \{s[\mathtt{p}][\mathtt{q}] :: \xi(\mathtt{p},\mathtt{q})\}_{(\mathtt{p},\mathtt{q})\in\mathsf{Chan}_G}$. By inversion, RT-EmptyQueue and RT-End are the only rules that can be applied in the typing derivation for $\Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash 0$. Thus, we have $\Omega = \emptyset$, changing our proof obligation to $\Theta \mid \Lambda \mid \Omega_s \vdash 0$. Since RT-EmptyQueue does only change the queue typing context, we have that $\Lambda = \bigcup_{s'\in\mathcal{S}}\{s'[\mathtt{p}]:\vec{q}_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}_{s'}}$ for a set of sessions $\mathcal{S}$ that does not contain s and $\mathrm{end}(\vec{q}_{\mathtt{p}})$ for every $\mathtt{p} \in \mathcal{P}_{s'}$ and $s' \in \mathcal{S}$. Therefore, we can also first apply RT-End $|\Lambda|$ times and last RT-EmptyQueue to obtain a typing derivation for $\Theta \mid \Lambda \mid \Omega_s \vdash 0$.

□

We proved the admissibility lemma for the type system for runtime configurations. The proof for the type system for processes is analogous for most cases.

Lemma 6.19 (Admissibility of Structural Precongruence for Process Typing)**.** If it holds that $\Theta \mid \Lambda \vdash P_1$ and $P_1 \sqsubseteq P_2$, then $\Theta \mid \Lambda \vdash P_2$.

Proof. The respective cases are analogous to the ones in the proof of Lemma 6.18. We only need to consider the case where $(\nu s{:}G)\, 0 \sqsubseteq 0$:
We assume there is a typing derivation for $\Theta \mid \Lambda \vdash (\nu s{:}G)\, 0$. We show there is a typing derivation for $\Theta \mid \Lambda \vdash 0$. By inversion, we know that PT-ν is the last rule to be applied and we get one of the premises: $\Theta \mid \Lambda, \Lambda_s \vdash 0$ with $\Lambda_s = \{s[\mathtt{p}]: \mathrm{init}(\mathscr{P}(G,\mathtt{p}))\}_{\mathtt{p}\in\mathcal{P}_G}$. By inversion, PT-0 and PT-End are the only rules that can be applied in the typing derivation for $\Theta \mid \Lambda, \Lambda_s \vdash 0$. Since PT-0 needs the second typing context to be empty, it is applied last and all other derivations are applications of PT-End. Therefore, $\Lambda = \bigcup_{s'\in\mathcal{S}}\{s'[\mathtt{p}]:\vec{q}_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}_{s'}}$ for some set of sessions $\mathcal{S}$ that does not contain s and $\mathrm{end}(\vec{q}_{\mathtt{p}})$ for every $\mathtt{p} \in \mathcal{P}_{s'}$ and $s' \in \mathcal{S}$. Therefore, we can also first apply RT-End $|\Lambda|$ times and last RT-0 to obtain a typing derivation for $\Theta \mid \Lambda \mid \emptyset \vdash 0$. □

We proceed with a few observations about our type system that we will use later in the proof for our main result.

To start, we show that a term x cannot appear in a runtime configuration if there is no type binding for it in the typing context.

Lemma 6.20. If $\Theta \mid \Lambda \mid \Omega \vdash R$ and x is not in Λ, then x cannot occur in R.

Proof. Towards a contradiction, assume that x occurs in R. Then, at some point in the typing derivation

$$\Theta \mid \Lambda \mid \Omega \vdash R$$

one of the following rules applies to handle x: RT-Q, RT-$\oplus$, or RT-&. Each of them requires all variables to occur in their respective typing contexts, yielding a contradiction. □

We defined a type system for processes and one for runtime configurations. Both are very similar and we defined runtime configurations to only have queues for active sessions. Once a process becomes active, we turn it into a runtime configuration using $\lceil \text{-} \rceil$. We show that this preserves typability with an empty queue typing context.

Lemma 6.21. If $\Theta \mid \Lambda \vdash P$, then $\Theta \mid \Lambda \mid \emptyset \vdash \lceil P \rceil$.

Proof. We prove this by induction on the structure of P.

For all except $P = P_1 \parallel P_2$ and $P = (\nu s{:}G)\, P'$, it holds that $\lceil P \rceil = P$. For all typing rules that processes and runtime configurations share, the queue typing context is not changed in the respective runtime configuration typing rule. Thus, $\Theta \mid \Lambda \mid \emptyset \vdash \lceil P \rceil$.

For $P = P_1 \parallel P_2$, the claim follows directly by induction hypothesis.

Last, we consider $P = (\nu s{:}G)\, P'$. We have the following typing derivation

$$\frac{\Lambda' = \{s[\mathtt{p}] : \mathrm{init}(\mathscr{P}(G, \mathtt{p}))\}_{\mathtt{p} \in \mathcal{P}_G} \qquad \Theta \mid \Lambda, \Lambda' \vdash P'}{\Theta \mid \Lambda \vdash (\nu s{:}G)\, P'} \text{ PT-}\nu$$

We show there is a typing derivation where the last rule is RT-ν. This requires a reachable configuration in the respective CSM: $(\vec{q}, \xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}_G})$. In this case, we choose the initial states and empty channels, which allows us to use RT-EmptyQueue for the queues.

$$\frac{\dfrac{\Theta \mid \Lambda, \Lambda_s \mid \emptyset \vdash P' \qquad \dfrac{}{\Theta \mid \emptyset \mid \Omega_s \vdash s \blacktriangleright \varepsilon} \text{ RT-EmptyQueue}}{\Theta \mid \Lambda, \Lambda_s \mid \Omega_s \vdash (P' \parallel s \blacktriangleright \varepsilon)} \text{ RT-}\parallel \quad (\vec{q}, \xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}_G}) \quad \Lambda_s = \{s[\mathtt{p}]{:}\vec{q}_{\mathtt{p}}\}_{\mathtt{p} \in \mathcal{P}_G} \qquad \Omega_s = \{s[\mathtt{p}][\mathtt{q}] :: \xi(\mathtt{p}, \mathtt{q})\}_{(\mathtt{p},\mathtt{q}) \in \mathsf{Chan}_G}}{\Theta \mid \Lambda \mid \emptyset \vdash (\nu s{:}G)\,(P' \parallel s \blacktriangleright \varepsilon)} \text{ RT-}\nu$$

where $\vec{q}_{\mathtt{p}} = \mathrm{init}(\mathscr{P}(G, \mathtt{p}))$ for every $\mathtt{p}$ and $\xi(\mathtt{p}, \mathtt{q}) = \varepsilon$ for every $\mathtt{p}, \mathtt{q}$. Thus, $(\vec{q}, \xi)$ is clearly reachable. Also, both second typing contexts then coincide: $\Lambda' = \Lambda_s$. Thus, $\Theta \mid \Lambda, \Lambda_s \mid \emptyset \vdash P'$, the last premise to satisfy, follows from induction hypothesis. □

With the following lemma, we show that, if there is a typing derivation for a process, then the queue typing context is empty.

Lemma 6.22. If P is a process such that $\Theta \mid \Lambda \mid \Omega \vdash P$, then $\Omega = \emptyset$.

Proof. We do induction on the depth of the typing derivation.

For the base case, we consider RT-0, for which the claim trivially holds, and RT-EmptyQueue, for which we reach a contradiction because $s \blacktriangleright \sigma$ is no process.

Let us turn to the induction step:

- RT-Q: trivially holds
- RT-End: by inversion and induction hypothesis
- RT-$\oplus$: by inversion and induction hypothesis for every $i \in I$
- RT-$\&$: by inversion and induction hypothesis for every $i \in I$
- RT-$\|$: by inversion and induction hypothesis twice
- RT-ν: by inversion and induction hypothesis
- RT-Queue: contradiction because $s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto l\langle v\rangle \cdot \vec{m}]$ is no process

□

In our type system, we type a queue from the first to the last element, when using RT-Queue. Thus, when applying inversion for this rule, we only get the type for the first element of a non-empty queue. The following lemma allows us to also obtain the type for its last element.

Lemma 6.23 (Message List Reversal)**.** Let γ be a queue type. If

$$\Theta \mid \Lambda \mid \Omega, s[\mathrm{p}][\mathrm{q}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto \vec{m}], \text{ then}$$

$$\Theta \mid \Lambda, v{:}L \mid \Omega, s[\mathrm{p}][\mathrm{q}] :: \gamma \cdot l(L) \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto \vec{m} \cdot l\langle v\rangle] \ .$$

Proof. We prove this claim by induction on the length n of $\vec{m} = l_1\langle v_1\rangle \cdot \ldots \cdot l_n\langle v_n\rangle$.

If $n = 0$, the claim is exactly the assumption.

For the induction step, we assume that $\vec{m} = l_1\langle v_1\rangle \cdot l_2\langle v_2\rangle \cdot \ldots \cdot l_n\langle v_n\rangle$ and the induction hypothesis holds for $l_2\langle v_2\rangle \cdot \ldots \cdot l_n\langle v_n\rangle$.

We want to show that

$$\Theta \mid \Lambda \mid \Omega, s[\mathrm{p}][\mathrm{q}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto l_1\langle v_1\rangle \cdot l_2\langle v_2\rangle \cdot \ldots \cdot l_n\langle v_n\rangle] \quad \text{implies}$$

$$\Theta \mid \Lambda, v{:}L \mid \Omega, s[\mathrm{p}][\mathrm{q}] :: \gamma \cdot l(L) \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto l_1\langle v_1\rangle \cdot l_2\langle v_2\rangle \cdot \ldots \cdot l_n\langle v_n\rangle \cdot l\langle v\rangle] \ .$$

By inversion of the premise, we know that $\Lambda = \Lambda', v_1{:}L_1$ and $\gamma = L_1 \cdot \gamma'$ in order to type v_1 with some type L_1. Thus, we can apply RT-Queue to our goal and then apply the induction hypothesis with $\Lambda = \Lambda'$ and $\gamma = \gamma'$ to conclude the proof. □

We show that the queue types reflect what is in the queues of runtime configurations.

Lemma 6.24. Assume that

$$\Theta \mid \Lambda \mid \Omega, s[\mathrm{p}][\mathrm{q}] :: l_1(L_1) \cdot \ldots \cdot l_k(L_k) \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto l'_1\langle v'_1\rangle, \ldots, l'_n\langle v'_n\rangle] \ .$$

Then $k = n$ and, for all $1 \leq i \leq k$, $l'_i = l_i$ and $v'_i{:}L_i$.

Proof. We do an induction on the depth of the typing derivation.

For the induction base, we have the following typing derivation:

$$\dfrac{}{\Theta \mid \emptyset \mid \{s[\mathsf{p}][\mathsf{q}] :: \varepsilon\}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_s} \vdash s \blacktriangleright \varepsilon} \text{ RT-EmptyQueue}$$

It is obvious that $k = 0 = n$ and there are no messages to consider.

For the induction step, we have the following typing derivation:

$$\dfrac{\Theta \mid \Lambda \mid \Omega, s[\mathsf{p}][\mathsf{q}] :: l_1(L_1) \cdot l_2(L_2) \cdot \ldots l_k(L_k) \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}) \mapsto l'_2\langle v'_2\rangle \cdot \ldots l'_n\langle v'_n\rangle]}{\Theta \mid \Lambda, v'_1 : L_1 \mid \Omega, s[\mathsf{p}][\mathsf{q}] :: l_1(L_1) \cdot l_2(L_2) \cdot \ldots l_k(L_k) \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}) \mapsto l_1\langle v'_1\rangle \cdot l'_2\langle v'_2\rangle \cdot \ldots l'_n\langle v'_n\rangle]} \text{ RT-Queue}$$

Let us first consider the length of the queue type and the queue: by induction hypothesis, we know that $n - 1 = k - 1$ and, thus, $k = n$. Second, let us consider the labels and payload types. For $i = 1$, the typing rule requires the labels to match and $v'_1 : L_1$ is required in the typing context. For $i > 1$, the induction hypothesis applies. □

We also provided typing context reductions. Here, we show that these actually preserve reachability for the CSM associated with a session.

Lemma 6.25 (Typing reductions preserve reachability)**.** Let $\Lambda = \hat{\Lambda}, \{\Lambda_s\}_{s \in \mathcal{S}}$ be a typing context and $\Omega = \hat{\Omega}, \{\Omega_s\}_{s \in \mathcal{S}}$ be a queue typing context with a set of sessions $\mathcal{S}$. Assume that

- $\hat{\Lambda}, \{\Lambda_s\}_{s \in \mathcal{S}} \mid \hat{\Omega}, \{\Omega_s\}_{s \in \mathcal{S}} \to \hat{\Lambda}', \{\Lambda'_s\}_{s \in \mathcal{S}} \mid \hat{\Omega}', \{\Omega'_s\}_{s \in \mathcal{S}}$, and
- for all $s \in \mathcal{S}$, it holds that there is $(\vec{q}, \xi) \in \text{reach}(\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p} \in \mathcal{P}_G})$ such that $\Lambda_s = \{s[\mathsf{p}] : q\}_{\mathsf{p} \in \mathcal{P}_G}$ and $\Omega_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p}, \mathsf{q})\}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_G}$

Then, for all $s \in \mathcal{S}$, it holds that there is $(\vec{q}', \xi') \in \text{reach}(\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p} \in \mathcal{P}_G})$ such that $\Lambda'_s = \{s[\mathsf{p}] : q'\}_{\mathsf{p} \in \mathcal{P}_G}$ and $\Omega'_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi'(\mathsf{p}, \mathsf{q})\}_{(\mathsf{p},\mathsf{q}) \in \mathsf{Chan}_G}$.

Proof. We do inversion on $\hat{\Lambda}, \{\Lambda_s\}_{s \in \mathcal{S}} \mid \hat{\Omega}, \{\Omega_s\}_{s \in \mathcal{S}} \to \hat{\Lambda}', \{\Lambda'_s\}_{s \in \mathcal{S}} \mid \hat{\Omega}', \{\Omega'_s\}_{s \in \mathcal{S}}$, yielding two cases.

First, we have

$$\dfrac{q \xrightarrow{\mathsf{p} \triangleright \mathsf{q} ! l(L)} q'}{s[\mathsf{p}] : q, \bar{\Lambda} \mid s[\mathsf{p}][\mathsf{q}] :: \gamma, \bar{\Omega} \to s[\mathsf{p}] : q', \bar{\Lambda} \mid s[\mathsf{p}][\mathsf{q}] :: \gamma \cdot l(L), \bar{\Omega}} \text{ TR-}\oplus$$

for some s, p, and q. For every $s' \neq s$, the claim trivially holds. For s, the changes to $s[\mathsf{p}]$ and $s[\mathsf{p}][\mathsf{q}]$ mimic the semantics of the CSM (cf. Section 2.3) while the premise $q \xrightarrow{\mathsf{p} \triangleright \mathsf{q} ! l(L)} q'$ ensures that such a transition is possible.

For the second case where we have

$$\dfrac{q \xrightarrow{\mathsf{p} \triangleleft \mathsf{q} ? l(L)} q'}{s[\mathsf{p}] : q, \bar{\Lambda} \mid s[\mathsf{q}][\mathsf{p}] :: l(L) \cdot \gamma, \bar{\Omega} \to s[\mathsf{p}] : q', \bar{\Lambda} \mid s[\mathsf{q}][\mathsf{p}] :: \gamma, \bar{\Omega}} \text{ TR-\&}$$

the reasoning is analogous but it mimics the receive case of the CSM semantics. □

With the substitution lemma, we prove that substituting a variable by a value with the same type preserves typability in our type system.

Lemma 6.26 (Substitution lemma)**.** For all L, if it holds that $\Theta \mid \Lambda, x:L \mid \Omega \vdash R$, then $\Theta \mid \Lambda, v:L \mid \Omega \vdash R[v/x]$,

Proof. We do an induction on the depth of the typing derivation and do a case analysis on the last applied rule of the derivation.

For the induction base, we consider both rules with depth 0 and show that there is no R' such that $R \to R'$. For both RT-0 and RT-EmptyQueue, the second typing context is empty, which contradicts our assumption that $x:L$ or $v:L$.

For the induction step, the induction hypothesis yields that the claim holds for typing derivations of smaller depth.

- RT-Q:
  We have that

$$\frac{\Theta(\mathsf{Q}) = L_1, \ldots, L_n}{\Theta \mid c_1:L_1, \ldots, c_{i-1}:L_{i-1}, x:L_i, c_{i+1}:L_{i+1}, \ldots c_n:L_n \mid \emptyset \vdash \mathsf{Q}[\vec{c}]} \text{RT-Q}$$

  It is straightforward that we need to show precisely the same premise for the desired typing derivation:

$$\Theta \mid c_1:L_1, \ldots, c_{i-1}:L_{i-1}, v:L_i, c_{i+1}:L_{i+1}, \ldots, c_n:L_n \mid \emptyset \vdash (\mathsf{Q}[\vec{c}])[v/x] \ .$$

- RT-End: We have two cases.
  First, we have

$$\frac{\Theta \mid \Lambda \mid \Omega \vdash R \qquad \text{end}(q)}{\Theta \mid x:q, \Lambda \mid \Omega \vdash R} \text{RT-End}$$

  We show that

$$\frac{\Theta \mid \Lambda \mid \Omega \vdash R[v/x] \qquad \text{end}(q)}{\Theta \mid v:q, \Lambda \mid \Omega \vdash R[v/x]} \text{RT-End}$$

  For the first typing derivation, we have $x:q, \Lambda$ as typing context. By the fact that Λ is a typing context (and no syntactic typing context), x does not occur in Λ. By inversion, we have $\Theta \mid \Lambda \mid \Omega \vdash R$. Thus, x cannot occur in R. If it did, R could only be typed with a typing context with x, which does not occur in Λ, given by contraposition of Lemma 6.20. Hence $R = R[v/x]$ and, thus, both premises coincide.

Second, we have

$$\dfrac{\Theta \mid \Lambda, x\!:\!L \mid \Omega \vdash R \qquad \mathrm{end}(q)}{\Theta \mid c\!:\!q, \Lambda, x\!:\!L \mid \Omega \vdash R}\ \text{RT-END}$$

We show that

$$\dfrac{\Theta \mid \Lambda, v\!:\!L \mid \Omega \vdash R[v/x] \qquad \mathrm{end}(q)}{\Theta \mid c\!:\!q, \Lambda, v\!:\!L \mid \Omega \vdash R[v/x]}\ \text{RT-END}$$

By inversion of the first typing derivation, we know that both premises hold. The second premise is the same for both derivations. For the first premise, the induction hypothesis applies.

- RT-$\oplus$:
  Here, we do a case analysis if $x = c$, $x = c_k$ for some $k \in I$ or neither of both.
  For the last case, we can apply inversion and the induction hypothesis applies to all cases of the right premise.
  For $x = c$, the second premise follows from inversion and the induction hypothesis, instantiated with $L = q_i$.
  We consider the case for $x = c_k$ in more detail.
  We have

$$\dfrac{\begin{array}{c}\delta(q) \supseteq \{(\mathtt{p}\triangleright\mathtt{q}_i!l_i(L_i), q_i) \mid i \in I\} \\ \forall i \in I \setminus \{k\}.\, \Theta \mid \Lambda, c\!:\!q_i, x\!:\!L, \{c_j\!:\!L_j\}_{j\in I\setminus\{i,k\}} \mid \Omega \vdash P_i \\ \Theta \mid \Lambda, c\!:\!q_i, \{c_j\!:\!L_j\}_{j\in I\setminus\{k\}} \mid \Omega \vdash P_k\end{array}}{\Theta \mid \Lambda, c\!:\!q, x\!:\!L, \{c_i\!:\!L_i\}_{i\in I\setminus\{k\}} \mid \Omega \vdash \bigoplus_{i\in I} c[\mathtt{q}_i]!l_i\langle c_i\rangle \,.\, P_i}\ \text{RT-}\oplus$$

By inversion, we obtain all premises. We show that

$$\dfrac{\begin{array}{c}\delta(q) \supseteq \{(\mathtt{p}\triangleright\mathtt{q}_i!l_i(L_i), q_i) \mid i \in I\} \\ \forall i \in I \setminus \{k\}.\, \Theta \mid \Lambda, c\!:\!q_i, v\!:\!L, \{c_j\!:\!L_j\}_{j\in I\setminus\{i,k\}} \mid \Omega \vdash P_i[v/x] \\ \Theta \mid \Lambda, c\!:\!q_i, \{c_j\!:\!L_j\}_{j\in I\setminus\{k\}} \mid \Omega \vdash P_k[v/x]\end{array}}{\Theta \mid \Lambda, c\!:\!q, v\!:\!L, \{c_i\!:\!L_i\}_{i\in I\setminus\{k\}} \mid \Omega \vdash (\bigoplus_{i\in I} c[\mathtt{q}_i]!l_i\langle c_i\rangle \,.\, P_i)[v/x]}\ \text{RT-}\oplus$$

The first premise is the same. The second premise, for every $i \in I \setminus \{k\}$, follows by the induction hypothesis. For the third premise, we claim that x cannot occur in P_k. In the conclusion of the first typing derivation, we have the typing context $\Lambda, c\!:\!q, x\!:\!L, \{c_i\!:\!L_i\}_{i\in I\setminus\{k\}}$. Thus, by assumption that each element has at most one type in a typing context, we know that x cannot occur in $\Lambda, c\!:\!q, \{c_i\!:\!L_i\}_{i\in I\setminus\{k\}}$. With Lemma 6.20, x cannot occur in P_k. Thus, $P_k[v/x] = P_k$ and the third premise in both typing derivations coincide, concluding this case.

- RT-&:
  We have

$$\frac{\delta(q) = \{(\mathtt{p} \triangleleft \mathtt{q}_i ?l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \Lambda, y_i : L_i, c : q_i \mid \Omega \vdash P_i}{\Theta \mid \Lambda, c : q \mid \Omega \vdash \&_{i \in I}\, c[\mathtt{q}_i] ?l_i(y_i) \,.\, P_i} \text{ RT-\&}$$

  We do a case analysis if $x = c$ or not.
  If not, the claim follows by inversion and induction hypothesis for the second premise.
  If $x = c$, there is $x : q_i$ in the second premise to type $P_i[v/x]$. The existence of such a typing derivation follows from inversion and induction hypothesis when instantiated with $L = q_i$.
- RT-‖:
  There are two symmetric cases. We only consider one of both. For this, we have

$$\frac{\Theta \mid \Lambda_1, x : L \mid \Omega_1 \vdash R_1 \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \Lambda_1, x : L, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R_1 \parallel R_2} \text{ RT-}\parallel$$

  We show that there is a typing derivation for

$$\Theta \mid \Lambda_1, v : L, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash (R_1 \parallel R_2)[v/x]$$

  By our assumption that typing contexts have at most one type per element, Λ_1 and Λ_2 cannot share any names. By inversion, we have $\Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2$. Thus, by Lemma 6.20, x cannot occur in R_2. Hence, $(R_1 \parallel R_2)[v/x] = R_1[v/x] \parallel R_2$. We claim the following typing derivation exists:

$$\frac{\Theta \mid \Lambda_1, v : L, \mid \Omega_1 \vdash R_1 \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \Lambda_1, v : L, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R_1[v/x] \parallel R_2} \text{ RT-}\parallel$$

  The second premise is the same as in the original typing derivation. The first premise can be obtained by inversion on the original typing derivation and applying the induction hypothesis.
- RT-ν:
  We have a typing derivation for

$$\Theta \mid \Lambda, x : L, \Lambda_s \mid \Omega, \Omega_s \vdash R \;\; .$$

  This case follows easily from inversion and applying the induction hypothesis to obtain a typing derivation for

$$\Theta \mid \Lambda, v : L, \Lambda_s \mid \Omega, \Omega_s \vdash R[v/x] \;\; .$$

- RT-Queue:
  We have a typing derivation for

$$\frac{\Theta \mid \Lambda \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto \vec{m}]}{\Theta \mid \Lambda, v'\!:\!L \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: l(L) \cdot \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto l\langle v' \rangle \cdot \vec{m}]}\ \text{RT-Queue}$$

  We do a case analysis if $x = v'$ or not. If so, the same typing derivation can simply be re-used as $v'\!:\!L$ also disappears in the original typing derivation. If not, the claim follows by inversion and application of the induction hypothesis.

This concludes the proof of the substitution lemma. □

Now, we turn to the main result about our type system: *subject reduction*. In short, if there is a runtime configuration with a typing derivation that can take a step, then the typing contexts can also take a step and can be used to type the new runtime configuration.

Theorem 6.27 (Subject reduction)**.** Let R be a runtime configuration with a set of active sessions $\mathcal{S}$. If

(1) $\vdash \Delta\!:\!\Theta$,
(2) $\Theta \mid \Lambda \mid \Omega \vdash R$ with $\Lambda = \hat{\Lambda}, \{\Lambda_s\}_{s \in \mathcal{S}}$ and $\Omega = \hat{\Omega}, \{\Omega_s\}_{s \in \mathcal{S}}$,
(3) for all $s \in \mathcal{S}$, it holds that there is $(\vec{q}, \xi) \in \text{reach}(\{\!\{\mathscr{P}(G,\mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}_G})$ such that $\Lambda_s = \{s[\mathtt{p}]\!:\!q\}_{\mathtt{p} \in \mathcal{P}_G}$ and $\Omega_s = \{s[\mathtt{p}][\mathtt{q}] :: \xi(\mathtt{p},\mathtt{q})\}_{(\mathtt{p},\mathtt{q}) \in \mathsf{Chan}_G}$
(4) $R \to R'$,

then there exist Λ' and Ω' with $\Lambda \mid \Omega \to \Lambda' \mid \Omega'$ such that $\Theta \mid \Lambda' \mid \Omega' \vdash R'$.

Proof. We do an induction on the depth of the typing derivation $\Theta \mid \Lambda \mid \Omega \vdash R$ and do a case analysis on the last applied rule of the derivation.

For the induction base, we consider both rules with depth 0 and show that there is no R' such that $R \to R'$.

- RT-0:
  It is trivial that 0 cannot reduce.
- RT-EmptyQueue:
  In this case $R = s \blacktriangleright \varepsilon$ for which none of the reduction rules apply.

For the induction step, the induction hypothesis yields that the claim holds for typing derivations of smaller depth. We do a case analysis on the typing rule that was applied last.

- RT-Q:
  We have that

$$\frac{\Theta(\mathsf{Q}) = L_1, \ldots, L_n}{\Theta \mid c_1\!:\!L_1, \ldots, c_n\!:\!L_n \mid \emptyset \vdash \mathsf{Q}[\vec{c}]}\ \text{RT-Q}$$

Thus, we know that $R = \mathsf{Q}[\vec{c}]$. However, none of the reduction rules apply, contradicting (4).

- RT-END:
  We have that

$$\frac{\Theta \mid \Lambda \mid \Omega \vdash R \qquad \mathrm{end}(q)}{\Theta \mid c{:}q, \Lambda \mid \Omega \vdash R}\ \text{RT-END}$$

  We have to show that

$$\frac{\Theta \mid \Lambda' \mid \Omega' \vdash R' \qquad \mathrm{end}(q)}{\Theta \mid c{:}q, \Lambda' \mid \Omega' \vdash R'}\ \text{RT-END}$$

  The second premise $\mathrm{end}(q)$ is the same for both. The first premise follows by the induction hypothesis. The induction hypothesis also yields that $\Lambda \mid \Omega \to \Lambda' \mid \Omega'$, concluding this case.

- RT-$\oplus$:
  Inversion on the typing derivation yields that $R = \oplus_{i \in I}\, s[\mathsf{p}][\mathsf{q}_i]!l_i\langle v_i\rangle \,.\, P_i$. None of the reduction rules apply and, thus, there is no R' such that $R \to R'$, contradicting (4).

- RT-$\&$:
  Inversion on the typing derivation yields that $R = \&_{i \in I}\, s[\mathsf{p}][\mathsf{q}_i]?l_i(L_i) \,.\, P_i$. None of the reduction rules apply and, thus, there is no R' such that $R \to R'$, contradicting (4).

- RT-$\|$:
  We do inversion on the reduction (4). Because of Lemma 6.18, we do not need to consider RR-$\sqsubseteq$.

  - RR-Q:
    We have

$$\frac{\text{RT-Q}\ \dfrac{\Theta(\mathsf{Q}) = \vec{L}}{\Theta \mid \vec{c}{:}\vec{L} \mid \emptyset \vdash \mathsf{Q}[\vec{c}]} \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \vec{c}{:}\vec{L}, \Lambda_2 \mid \Omega_2 \vdash \mathsf{Q}[\vec{c}] \parallel R_2}\ \text{RT-}\|$$

    By inversion on (4), we have

$$\frac{\Delta(\mathsf{Q}, \vec{c}) \parallel R_2 \to R'}{\mathsf{Q}[\vec{c}] \parallel R_2 \to R'}\ \text{RR-Q}$$

    By assumption that Q is defined in Δ and by definition of Δ, we have that

$$\Delta = \Delta_1; (\mathsf{Q}[\vec{x}] = P); \Delta_2$$

for some Δ_1 and Δ_2.
We claim there is a typing derivation for

$$\Theta \mid \vec{c}{:}\vec{L} \vdash P[\vec{c}/\vec{x}]$$

(1) states that $\vdash \Delta{:}\Theta$. Inversion on (1) for $|\Delta_1|$ times yields

$$\Theta \mid \vec{x}{:}\vec{L} \vdash P \ .$$

We obtain a typing derivation after $|\vec{x}|$ applications of the substitution lemma (Lemma 6.26).
We do a case analysis on the structure of P and simultaneously if $R' = \texttt{err}$ for second case.

* $P = \oplus_{i\in I}\, x[\mathsf{q}_i]!l_i\langle x_i\rangle \,.\, P_i$:
  Let us rewrite the typing context: $\vec{c}{:}\vec{L} = (x{:}q, \{x_j{:}L_j\}_{j\in I}, \Lambda_1)[\vec{c}/\vec{x}]$. Without loss of generality, let $x[\vec{c}/\vec{x}] = s[\mathsf{p}]$. Then $R' = \lceil P_k[\vec{c}/\vec{x}]\rceil \parallel s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k) \mapsto \vec{m}\cdot l_k\langle v_k\rangle]$ for some $k \in I$. We show there is a typing derivation for any $k \in I$:

$$\dfrac{\begin{array}{c}\Theta \mid (x{:}q_k, \{x_j{:}L_j\}_{j\in I\setminus\{k\}}, \Lambda_1)[\vec{c}/\vec{x}] \mid \emptyset \vdash \lceil P_k[\vec{c}/\vec{x}]\rceil \\ \Theta \mid (c_k{:}L_k)[\vec{c}/\vec{x}], \Lambda_2 \mid \hat{\Omega}_2, s[\mathsf{p}][\mathsf{q}_k] :: \gamma\cdot l_k(L_k[\vec{c}/\vec{x}]) \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k)\mapsto \vec{m}\cdot l_k\langle v_k\rangle]\end{array}}{\Theta \mid (x{:}q_k, \{x_j{:}L_j\}_{j\in I}, \Lambda_1)[\vec{c}/\vec{x}], \Lambda_2 \mid \hat{\Omega}_2, s[\mathsf{p}][\mathsf{q}_k] :: \gamma\cdot l_k(L_k) \vdash R'}\ \text{RT-}\parallel$$

  By inversion on

$$\Theta \mid (x{:}q, \{x_j{:}L_j\}_{j\in I}, \Lambda_1)[\vec{c}/\vec{x}] \vdash (\bigoplus_{i\in I} x[\mathsf{q}_i]!l_i\langle x_i\rangle \,.\, P_i)[\vec{c}/\vec{x}] \ ,$$

  we get $\Theta \mid (x{:}q_i, \{x_j{:}L_j\}_{j\in I\setminus\{i\}}, \Lambda_1)[\vec{c}/\vec{x}] \vdash P_i[\vec{c}/\vec{x}]$ for every $i \in I$. Instantiating $i = k$ and applying Lemma 6.21 yields the desired premise:

$$\Theta \mid (x{:}q_i, \{x_j{:}L_j\}_{j\in I\setminus\{i\}}, \Lambda_1)[\vec{c}/\vec{x}] \mid \emptyset \vdash \lceil P_i[\vec{c}/\vec{x}]\rceil$$

  We show there is a typing derivation

$$\dfrac{\Theta \mid \Lambda_2 \mid \hat{\Omega}_2, s[\mathsf{p}][\mathsf{q}_k] :: \gamma \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k)\mapsto \vec{m}]}{\Theta \mid (c_k{:}L_k)[\vec{c}/\vec{x}], \Lambda_2 \mid \hat{\Omega}_2, s[\mathsf{p}][\mathsf{q}_k] :: \gamma\cdot l_k(L_k[\vec{c}/\vec{x}]) \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k)\mapsto \vec{m}\cdot l_k\langle v_k\rangle]}\ \text{RT-Queue}$$

  By inversion on $R \to R'$, we have that $R_2 = s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k)\mapsto \vec{m}]$. By inversion on $\Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2$, we obtain

$$\Theta \mid \Lambda_2 \mid \hat{\Omega}_2, s[\mathsf{p}][\mathsf{q}_k] :: \gamma \vdash s \blacktriangleright \sigma[(\mathsf{p},\mathsf{q}_k)\mapsto \vec{m}]$$

  which is the desired premise.
  It is straightforward that there is a typing context reduction for the corresponding typing contexts, using TR-$\oplus$.

* $P = \&_{i\in I}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i$ and $R' \neq \mathtt{err}$:
Let us rewrite the typing context: $\vec{c}{:}\vec{L} = (x{:}q, \Lambda_1)[\vec{c}/\vec{x}]$. Without loss of generality, let $x[\vec{c}/\vec{x}] = s[\mathtt{p}]$. Then $R' = \lceil P_k[v_k/y_k]\rceil \parallel s \blacktriangleright [(\mathtt{q}_k, \mathtt{p}) \mapsto \vec{m}]$ and $R_2 = s \blacktriangleright [(\mathtt{q}_k, \mathtt{p}) \mapsto l_k\langle v_k\rangle \cdot \vec{m}]$. We show there is a typing derivation for any $k \in I$:

$$\dfrac{\begin{array}{c}\Theta \mid v_k{:}L_k, (x{:}q_k, \Lambda_1)[\vec{c}/\vec{x}] \mid \emptyset \vdash \lceil P_k[\vec{c}/\vec{x}][v_k/y_k]\rceil \\ \Theta \mid \Lambda_2 \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p}, \mathtt{q}_k) \mapsto \vec{m}]\end{array}}{\Theta \mid (x{:}q_k, \Lambda_1)[\vec{c}/\vec{x}], \Lambda_2 \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \vdash \lceil P_k[\vec{c}/\vec{x}][v_k/y_k]\rceil \parallel s \blacktriangleright \sigma[(\mathtt{p}, \mathtt{q}_k) \mapsto \vec{m}]}\ \text{RT-}\parallel$$

By inversion on $\Theta \mid (x{:}q, \Lambda_1)[\vec{c}/\vec{x}] \vdash (\&_{i\in I}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i)[\vec{c}/\vec{x}]$, we get

$$\Theta \mid (x{:}q_i, y_i{:}L_i, \Lambda_1)[\vec{c}/\vec{x}] \vdash P_i[\vec{c}/\vec{x}]$$

for every $i \in I$. Instantiating $i = k$ and applying Lemma 6.21 yields the desired premise:

$$\Theta \mid v_k{:}L_k, (x{:}q_k, \Lambda_1)[\vec{c}/\vec{x}] \mid \emptyset \vdash \lceil P_k[\vec{c}/\vec{x}][v_k/y_k]\rceil$$

The second premise is obtained by inversion on

$$\Theta \mid v_k{:}L_k, \Lambda_2 \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{p}_k] :: l_k(L_k) \cdot \gamma \vdash s \blacktriangleright [(\mathtt{q}_k, \mathtt{p}) \mapsto l_k\langle v_k\rangle \cdot \vec{m}]\ .$$

It is straightforward that there is a typing context reduction for the corresponding typing contexts, using TR-&.

* $P = \&_{i\in I}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i$ and $R' = \mathtt{err}$:
Let us rewrite the typing context: $\vec{c}{:}\vec{L} = (x{:}q, \Lambda_1)[\vec{c}/\vec{x}]$. Without loss of generality, let $x[\vec{c}/\vec{x}] = s[\mathtt{p}]$. Then, by inversion on $R \to R'$, we have $R_2 = s \blacktriangleright \sigma$ and for every $i \in I$, $\sigma(\mathtt{q}_i, \mathtt{p}) = l\langle_\rangle \cdot \vec{m}$ and $l_i \neq l$. We claim that there is no typing derivation

$$\Theta \mid (x{:}q, \Lambda_1)[\vec{c}/\vec{x}], \Lambda_2 \mid \Omega_2 \vdash (\underset{i\in I}{\&}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i)[\vec{c}/\vec{x}] \parallel s \blacktriangleright \sigma\ .$$

By inversion, such a typing derivation must have the following shape:

$$\dfrac{\dfrac{\begin{array}{c}\delta(q) = \{(\mathtt{p}{\triangleleft}\mathtt{q}_i?l_i(L_i), q_i) \mid i \in I\} \\ \forall i \in I.\, \Theta \mid (x{:}q_i, \Lambda_1)[\vec{c}/\vec{x}], y_i{:}L_i, \mid \emptyset \vdash P_i\end{array}}{\Theta \mid (x{:}q, \Lambda_1)[\vec{c}/\vec{x}] \mid \emptyset \vdash (\underset{i\in I}{\&}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i)[\vec{c}/\vec{x}]}\ \text{RT-}\& \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash s \blacktriangleright \sigma}{\Theta \mid (x{:}q, \Lambda_1)[\vec{c}/\vec{x}], \Lambda_2 \mid \Omega_2 \vdash (\underset{i\in I}{\&}\, x[\mathtt{q}_i]?l_i(y_i)\,.\,P_i)[\vec{c}/\vec{x}] \parallel s \blacktriangleright \sigma}\ \text{RT-}\parallel$$

Let us rewrite the typing and queue typing context:

$$\begin{aligned}(x{:}q, \Theta)[\vec{c}/\vec{x}], \Lambda_2 &= \hat{\Lambda}, \Lambda_s \\ \Omega_2 &= \hat{\Omega}, \Omega_s\end{aligned}$$

By assumption, we know that there is $(\vec{q}, \xi) \in \text{reach}(\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}_G})$ such that $\Lambda_s = \{s[\mathrm{p}]{:}q\}_{\mathrm{p} \in \mathcal{P}_G}$ and $\Omega_s = \{s[\mathrm{p}][\mathrm{q}] :: \xi(\mathrm{p}, \mathrm{q})\}_{(\mathrm{p},\mathrm{q}) \in \mathsf{Chan}_G}$. Recall the condition on the reduction semantics: $\forall i \in I.\, \sigma(\mathrm{q}_i, \mathrm{p}) = l\langle_\rangle \cdot \vec{m}$ and $l_i \neq l$. Thus, with Lemma 6.24, it follows that, for all $i \in I$, $\xi(\mathrm{q}_i, \mathrm{p}) = l'_i(_) \cdot _$ with $l'_i \neq l_i$. For the CSM $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p} \in \mathcal{P}_G}$, this entails that p expects to receive a message from a set of other participants, ranged over by q_i, but the first message in each channel does not match. This yields a contradiction to Corollary 5.17, which states that the subset projection satisfies feasible eventual reception.

- RR-ctx:
  For the context rule, two cases apply: $R \parallel \mathbb{C}$ or $\mathbb{C} \parallel R$. Both cases can be proven analogous, which is why we only prove the first. We have that

$$\dfrac{\Theta \mid \Lambda_1 \mid \Omega_1 \vdash R_1 \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \vdash R_1 \parallel R_2}\ \text{RT-}\parallel$$

  and we want to show that

$$\dfrac{\Theta \mid \Lambda'_1 \mid \Omega'_1 \vdash R'_1 \qquad \Theta \mid \Lambda_2 \mid \Omega_2 \vdash R_2}{\Theta \mid \Lambda'_1, \Lambda_2 \mid \Omega'_1, \Omega_2 \vdash R'_1 \parallel R_2}\ \text{RT-}\parallel$$

  The second premise is trivially satisfied. The first premise follows from the induction hypothesis, which also yields that $\Lambda_1 \mid \Omega_1 \to \Lambda'_1 \mid \Omega'_1$. We can apply Lemma 6.14 to obtain $\Lambda_1, \Lambda_2 \mid \Omega_1, \Omega_2 \to \Lambda'_1, \Lambda_2 \mid \Omega'_1, \Omega_2$, concluding this case.
- RR-out:
  With three inversions on the typing derivation, we have a typing derivation with the following shape:

$$\dfrac{\dfrac{\delta(q) \supseteq \{(\mathrm{p} \triangleright \mathrm{q}_i ! l_i(L_i), q_i) \mid i \in I\} \qquad \forall i \in I.\, \Theta \mid \hat{\Lambda}_1, s[\mathrm{p}]{:}q_i, \{v_j{:}L_j\}_{j \in I \setminus \{i\}} \mid \Omega_1 \vdash P_i}{\Theta \mid \hat{\Lambda}_1, s[\mathrm{p}]{:}q, \{v_i{:}L_i\}_{i \in I} \mid \Omega_1 \vdash \bigoplus_{i \in I} s[\mathrm{p}][\mathrm{q}_i]!l_i\langle v_i\rangle \,.\, P_i}\ \text{RT-}\oplus \qquad \dfrac{\vdots}{\Theta \mid \Lambda_2 \mid \hat{\Omega}_2, s[\mathrm{p}][\mathrm{q}_k] :: \gamma \vdash s \blacktriangleright \sigma[(\mathrm{p}, \mathrm{q}) \mapsto \vec{m}]}\ \text{RT-Queue}}{\Theta \mid \hat{\Lambda}_1, s[\mathrm{p}]{:}q, \{v_i{:}L_i\}_{i \in I}, \Lambda_2 \mid \Omega_1, \hat{\Omega}_2, s[\mathrm{p}][\mathrm{q}_k] :: \gamma \vdash \bigoplus_{i \in I} s[\mathrm{p}][\mathrm{q}_i]!l_i\langle v_i\rangle \,.\, P_i \parallel \sigma[(\mathrm{p}, \mathrm{q}_k) \mapsto \vec{m}]}\ \text{RT-}\parallel$$

  By inversion, we obtain all premises. We show that

$$\dfrac{\dfrac{\dfrac{\vdots}{\Theta \mid \hat{\Lambda}_1, s[\mathtt{p}]:q_k, \{v_i:L_i\}_{i\in I\setminus\{k\}} \mid \Omega_1 \vdash \lceil P_k \rceil}\text{RT-}\oplus \qquad \dfrac{\vdots}{\Theta \mid \Lambda_2, v_k:L_k \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \cdot l_k\langle L_k\rangle \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}_k) \mapsto \vec{m} \cdot l_k\langle v_k\rangle]}\text{RT-Queue}}{}}{\Theta \mid \hat{\Lambda}_1, s[\mathtt{p}]:q_k, \{v_i:L_i\}_{i\in I\setminus\{k\}}, v_k:L_k, \Lambda_2 \mid \Omega_1, \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \cdot l_k\langle L_k\rangle \vdash R'}\text{RT-}\|$$

for $R' = \lceil P_k \rceil \parallel \sigma[(\mathtt{p},\mathtt{q}_k) \mapsto \vec{m} \cdot l_k\langle v_k\rangle]$.
First, we show there is a typing derivation for

$$\Theta \mid \hat{\Lambda}_1, s[\mathtt{p}]:q_k, \{v_i:L_i\}_{i\in I\setminus\{k\}} \mid \Omega_1 \vdash \lceil P_k \rceil$$

first. We instantiate the premise

$$\forall i \in I.\, \Theta \mid \hat{\Lambda}_1, s[\mathtt{p}]:q_i, \{v_j:L_j\}_{j\in I\setminus\{i\}} \mid \Omega_1 \vdash P_i$$

for $i = k$ and obtain

$$\Theta \mid \hat{\Lambda}_1, s[\mathtt{p}]:q_k, \{v_j:L_j\}_{j\in I\setminus\{k\}} \mid \Omega_1 \vdash P_k$$

By Lemma 6.22, we know that $\Omega_1 = \emptyset$ and, thus, Lemma 6.21 applies and concludes this case.
Second, we show there is a typing derivation for

$$\Theta \mid \Lambda_2, v_k:L_k \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \cdot l_k\langle L_k\rangle \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}_k) \mapsto \vec{m} \cdot l_k\langle v_k\rangle]$$

From inversion of the original typing derivation, we have

$$\Theta \mid \Lambda_2 \mid \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}_k) \mapsto \vec{m}]$$

With Lemma 6.23, the claim follows.
It remains to show that there is a transition for the respective typing contexts:

$$\begin{aligned}
\Lambda &:= \hat{\Lambda}_1, s[\mathtt{p}]:q, \{v_i:L_i\}_{i\in I}, \Lambda_2 \\
\Lambda' &:= \hat{\Lambda}_1, s[\mathtt{p}]:q_k, \{v_i:L_i\}_{i\in I\setminus\{k\}}, v_k:L_k, \Lambda_2 \\
\Omega &:= \Omega_1, \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \\
\Omega' &:= \Omega_1, \hat{\Omega}_2, s[\mathtt{p}][\mathtt{q}_k] :: \gamma \cdot l_k\langle L_k\rangle
\end{aligned}$$

Note that the change from Λ to Λ' is solely the type of $s[\mathtt{p}]$ while $s[\mathtt{p}][\mathtt{q}_k]$ is the only change from Λ to Λ'. Thus, we can simply apply TR-$\oplus$ to obtain a typing context reduction.

– RR-IN:
With three inversions on the typing derivation, we have a typing derivation with the following shape:

$$
\dfrac{
\dfrac{
\delta(q) = \{(\mathsf{p} \triangleleft \mathsf{q}_i ? l_i(L_i), q_i) \mid i \in I\} \quad \forall i \in I .\, \Theta \mid \hat{\Lambda}_1, y_i : L_i, s[\mathsf{p}] : q_i \mid \Omega_1 \vdash P_i
}{
\Theta \mid \hat{\Lambda}_1, s[\mathsf{p}] : q \mid \Omega_1 \vdash \mathop{\&}_{i \in I} s[\mathsf{p}][\mathsf{q}_i] ? l_i(y_i) \,.\, P_i
}\;\text{RT-\&}
\quad
\dfrac{
\Theta \mid \hat{\Lambda}_2 \mid \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto \vec{m}]
}{
\Theta \mid \hat{\Lambda}_2, v_k : L_k, \mid \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: l_k(L_k) \cdot \gamma \vdash s \blacktriangleright \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto l_k\langle v_k \rangle \cdot \vec{m}]
}\;\text{RT-Queue}
}{
\Theta \mid \hat{\Lambda}_1, s[\mathsf{p}] : q, \hat{\Lambda}_2, v_k : L_k \mid \Omega_1, \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: l_k(L_k) \cdot \gamma \vdash R'
}\;\text{RT-}\|
$$

for $R' \mathbin{\&}_{i \in I} s[\mathsf{p}][\mathsf{q}_i] ? l_i(y_i) \,.\, P_i \parallel \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto l_k\langle v_k \rangle \cdot \vec{m}]$.
By inversion, we obtain all the premises. We show that there is a typing derivation of shape

$$
\dfrac{
\dfrac{\vdots}{
\Theta \mid \hat{\Lambda}_1, s[\mathsf{p}] : q_k, v_k : L_k \mid \Omega_1 \vdash \lceil P_k[v_k / y_k] \rceil
}\;\text{RT-\&}
\quad
\dfrac{\vdots}{
\Theta \mid \hat{\Lambda}_2 \mid \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto \vec{m}]
}\;\text{RT-Queue}
}{
\Theta \mid \hat{\Lambda}_1, s[\mathsf{p}] : q, v_k : L_k, \hat{\Lambda}_2 \mid \Omega_1, \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: \gamma \vdash \lceil P_i[v_k / y_k] \rceil \parallel \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto \vec{m}]
}\;\text{RT-}\|
$$

First, we show there is a typing derivation for

$$\Theta \mid \hat{\Lambda}_1, s[\mathsf{p}] : q_k, v_k : L_k \mid \Omega_1 \vdash \lceil P_k[v_k / y_k] \rceil \quad .$$

From inversion, we get the following premise from the original typing derivation:

$$i \in I .\, \Theta \mid \hat{\Lambda}_1, y_i : L_i, s[\mathsf{p}] : q_i \mid \Omega_1 \vdash P_i$$

which we instantiate with $i = k$ to obtain:

$$\Theta \mid \hat{\Lambda}_1, y_k : L_k, s[\mathsf{p}] : q_k \mid \Omega_1 \vdash P_k \quad .$$

With Lemma 6.26, we get

$$\Theta \mid \hat{\Lambda}_1, v_k : L_k, s[\mathsf{p}] : q_k \mid \Omega_1 \vdash P_k[v_k / y_k] \quad .$$

By Lemma 6.22, we know that $\Omega_1 = \emptyset$ and, thus, Lemma 6.21 applies and concludes this case.
Second, there is a typing derivation for

$$\Theta \mid \hat{\Lambda}_2 \mid \hat{\Omega}_2, s[\mathsf{q}_k][\mathsf{p}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathsf{q}_k, \mathsf{p}) \mapsto \vec{m}]$$

by inversion on the original typing derivation.
It remains to show that there is a transition for the respective typing contexts:

$$\begin{aligned}\Lambda &:= \hat{\Lambda}_1, s[\mathrm{p}]:q, \hat{\Lambda}_2, v_k:L_k \\ \Lambda' &:= \hat{\Lambda}_1, s[\mathrm{p}]:q_k, v_k:L_k, \hat{\Lambda}_2 \\ \Omega &:= \Omega_1, \hat{\Omega}_2, s[\mathrm{q}_k][\mathrm{p}] :: l_k(L_k) \cdot \gamma \\ \Omega' &:= \Omega_1, \hat{\Omega}_2, s[\mathrm{q}_k][\mathrm{p}] :: \gamma\end{aligned}$$

Note that the change from Λ to Λ' is solely the type of $s[\mathrm{p}]$ while $s[\mathrm{q}_k][\mathrm{p}]$ is the only change from Λ to Λ'. Thus, we can simply apply TR-& to obtain a typing context reduction.

– RR-ERR1:
By assumption, we have a typing derivation for $\&_{i\in I}\, s[\mathrm{p}][\mathrm{q}_i]?l_i(y_i)\,.\,P_i \parallel s \blacktriangleright \sigma$ and it holds that $\forall i \in I.\, \sigma(\mathrm{q}_i, \mathrm{p}) = l\langle_\rangle \cdot \vec{m}$ and $l_i \neq l$. By inversion and Lemma 6.22, the typing derivation must have the following shape:

$$\dfrac{\dfrac{\begin{array}{c}\delta(q) = \{(\mathrm{p}\triangleleft\mathrm{q}_i?l_i(L_i), q_i) \mid i \in I\} \\ \forall i \in I.\, \Theta \mid \hat{\Lambda}_1, y_i:L_i, s[\mathrm{p}]:q_i \mid \emptyset \vdash P_i\end{array}}{\Theta \mid \hat{\Lambda}_1, s[\mathrm{p}]:q \mid \emptyset \vdash \underset{i\in I}{\&}\, s[\mathrm{p}][\mathrm{q}_i]?l_i(y_i)\,.\,P_i}\ \text{RT-}\& \qquad \dfrac{\vdots}{\Theta \mid \Lambda_2 \mid \Omega_2 \vdash s \blacktriangleright \sigma}}{\Theta \mid \hat{\Lambda}_1, s[\mathrm{p}]:q, \Lambda_2 \mid \Omega_2 \vdash \underset{i\in I}{\&}\, s[\mathrm{p}][\mathrm{q}_i]?l_i(y_i)\,.\,P_i \parallel s \blacktriangleright \sigma}\ \text{RT-}\|$$

Let us rewrite the typing and queue typing context:

$$\begin{aligned}\hat{\Lambda}_1, s[\mathrm{p}]:q, \Lambda_2 &= \hat{\Lambda}, \Lambda_s \\ \Omega_2 &= \hat{\Omega}, \Omega_s\end{aligned}$$

By assumption, we know that there is $(\vec{q}, \xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}_G})$ such that $\Lambda_s = \{s[\mathrm{p}]:q\}_{\mathrm{p}\in\mathcal{P}_G}$ and $\Omega_s = \{s[\mathrm{p}][\mathrm{q}] :: \xi(\mathrm{p}, \mathrm{q})\}_{(\mathrm{p},\mathrm{q})\in\mathsf{Chan}_G}$. Recall the condition on the reduction semantics: $\forall i \in I.\, \sigma(\mathrm{q}_i, \mathrm{p}) = l\langle_\rangle \cdot \vec{m}$ and $l_i \neq l$. Thus, with Lemma 6.24, it follows that, for all $i \in I$, $\xi(\mathrm{q}_i, \mathrm{p}) = l'_i(_) \cdot _$ with $l'_i \neq l_i$. For the CSM $\{\!\{\mathscr{P}(G, \mathrm{p})\}\!\}_{\mathrm{p}\in\mathcal{P}_G}$, this entails that p expects to receive a message from a set of other participants, ranged over by q_i, but the first message in each channel does not match. Thus, none of them will ever be received. This yields a contradiction to Corollary 5.17, which states that the subset projection satisfies feasible eventual reception.

- RT-ν:
By inversion on the typing derivation, there is a typing derivation

$$\Theta \mid \Lambda \mid \Omega \vdash (\nu s:G)\, R \ .$$

We do inversion on (4), yielding two reduction rules that apply.

- RR-CTX:
  We have

$$\frac{\begin{array}{cc}(\vec{q},\xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}_G}) & \Lambda_s = \{s[\mathsf{p}]{:}\vec{q}_{\mathsf{p}}\}_{\mathsf{p}\in\mathcal{P}_G} \\ \Omega_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G} & \Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash R\end{array}}{\Theta \mid \Lambda \mid \Omega \vdash (\nu s{:}G)\, R}\ \text{RT-}\nu$$

  By inversion, we obtain all premises. We show that

$$\frac{\begin{array}{cc}(\vec{q}',\xi') \in \mathrm{reach}(\{\!\{\mathscr{P}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}_G}) & \Lambda'_s = \{s[\mathsf{p}]{:}q'_{\mathsf{p}}\}_{\mathsf{p}\in\mathcal{P}_G} \\ \Omega'_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi'(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G} & \Theta \mid \Lambda, \Lambda'_s \mid \Omega, \Omega'_s \vdash R'\end{array}}{\Theta \mid \Lambda \mid \Omega \vdash (\nu s{:}G)\, R'}\ \text{RT-}\nu$$

  The first premise is the same as for the original typing derivation. We know that $\Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash R$. With the induction hypothesis, we get

$$\Theta \mid \Lambda, \Lambda'_s \mid \Omega, \Omega'_s \vdash R' \text{ and } \Lambda, \Lambda_s \mid \Omega, \Omega_s \to \Lambda, \Lambda'_s \mid \Omega, \Omega'_s \ .$$

  The first fact proves the last premise for the new typing derivation. For the remaining ones, we apply Lemma 6.25, which yields that there is $(\vec{q}',\xi') \in \mathrm{reach}(\{\!\{\mathscr{P}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}_G})$ such that $\Lambda'_s = \{s[\mathsf{p}]{:}q'\}_{\mathsf{p}\in\mathcal{P}_G}$ and $\Omega'_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi'(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G}$. These are precisely the remaining premises for the new typing derivation. It is obvious that there is a reduction for the typing contexts, which concludes this case.

- RR-ERR2:
  We have a typing derivation for

$$\Theta \mid \Lambda \mid \Omega \vdash (\nu s{:}G)\ s \blacktriangleright \sigma$$

  and know $\sigma(\mathsf{p},\mathsf{q}) \neq \varepsilon$ for some p, q. We do inversion on the typing derivation:

$$\frac{\begin{array}{cc}\dfrac{\vdots}{\Theta \mid \Lambda, \Lambda_s \mid \Omega, \Omega_s \vdash s \blacktriangleright \sigma}\ \text{RT-Queue} & (\vec{q},\xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G,\mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}_G}) \\ \Lambda_s = \{s[\mathsf{p}]{:}\vec{q}_{\mathsf{p}}\}_{\mathsf{p}\in\mathcal{P}_G} & \Omega_s = \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G}\end{array}}{\Theta \mid \Lambda \mid \Omega \vdash (\nu s{:}G)\ s \blacktriangleright \sigma}\ \text{RT-}\nu$$

  By definition of Λ_s, there is a type $s[\mathsf{p}]{:}q$ for every $\mathsf{p} \in \mathcal{P}_G$. There is a typing derivation for

$$\Theta \mid \Lambda, \{s[\mathsf{p}]{:}\vec{q}_{\mathsf{p}}\}_{\mathsf{p}\in\mathcal{P}_G} \mid \Omega, \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G} \vdash s \blacktriangleright \sigma \ .$$

The only applicable typing rules (in the whole derivation) are RT-END, RT-EMPTYQUEUE, and RT-QUEUE. By our assumption that there is a strict partial order for the global types in our system, $s[\text{-}]$ does not appear in σ.
Thus, RT-END needs to be applied to reduce the typing context $\{s[\mathtt{p}]{:}\vec{q}_{\mathtt{p}}\}_{\mathtt{p}\in\mathcal{P}_G}$ to only contain Λ, which can then be used to type the queue with RT-QUEUE. The premise of RT-END requires that $\mathrm{end}(q)$, i.e. q is a final state and has not outgoing receive transition. This, however, entails that $(\vec{q},\xi)$ is a non-final configuration where all participants are in final states and the channels are not empty, yielding a deadlock. This contradicts the fact that $\{\!\{\mathscr{P}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}_G}$ is deadlock-free (Theorem 5.14), concluding this case.

- RT-QUEUE:
  We have the typing derivation

$$\dfrac{\Theta \mid \Lambda \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: \gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto \vec{m}]}{\Theta \mid \Lambda, v{:}L \mid \Omega, s[\mathtt{p}][\mathtt{q}] :: l(L)\cdot\gamma \vdash s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto l\langle v\rangle\cdot\vec{m}]}\ \text{RT-QUEUE}$$

  However, there is not R' such that $s \blacktriangleright \sigma[(\mathtt{p},\mathtt{q}) \mapsto l\langle v\rangle\cdot\vec{m}] \to R'$, contradicting (4).

□

From subject reduction, *type safety* follows: if a process can be typed, any runtime configuration that can be reached from this process cannot contain an error.

Corollary 6.28 (Type safety)**.** Assume that $\vdash \Delta{:}\Theta$ and $\Theta \mid \emptyset \vdash P$. If $\lceil P\rceil \to^* R$, then, $R \neq \mathtt{err}$.

Proof. From Lemma 6.21, we know that $\Theta \mid \emptyset \mid \emptyset \vdash \lceil P\rceil$. By definition $\to^* := \{\to^k \mid k \geq 0\}$. We prove a stronger claim: For all $k \geq 0$, if $\lceil P\rceil \to^k R$, then,

- $\Theta \mid \Lambda \mid \Omega \vdash R$ with $\Lambda = \hat{\Lambda}, \{\Lambda_s\}_{s\in\mathcal{S}}$ and $\Omega = \hat{\Omega}, \{\Omega_s\}_{s\in\mathcal{S}}$, and
- for all $s \in \mathcal{S}$, it holds that there is $(\vec{q},\xi) \in \mathrm{reach}(\{\!\{\mathscr{P}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}_G})$ such that $\Lambda_s = \{s[\mathtt{p}]{:}q\}_{\mathtt{p}\in\mathcal{P}_G}$ and $\Omega_s = \{s[\mathtt{p}][\mathtt{q}] :: \xi(\mathtt{p},\mathtt{q})\}_{(\mathtt{p},\mathtt{q})\in\mathsf{Chan}_G}$

This claim entails that $R \neq \mathtt{err}$ because $\mathtt{err}$ cannot be typed but R can be typed.

We prove the claim by induction on k.

For $k = 0$, the claim trivially follows because both the typing and queue typing context is empty, trivially satisfying the conditions.

For the induction step, we have $\lceil P\rceil \to^k R$, the claim holds for R, and $R \to R'$. With Subject Reduction (Theorem 6.27), we proved precisely what we need to show for R'. □

Subject reduction shows that any step of a runtime configuration can be mimicked by the typing contexts and these can be used to type the new runtime configuration. Since `err` cannot be typed, this shows that a typed runtime configuration can never reduce to `err`, yielding type safety. While this is a safety property, *session fidelity* deals with progress. Roughly speaking, if the typing contexts can take a step, then the runtime configuration can also take a step. In most MST frameworks, this can only be proven in the presence of a single session. Thus, we define the following restriction of our type system.

Definition 6.29. We define $\Vdash_{SF}$ to be $\vdash$ but without the rules PT-ν and RT-ν. Using this, we define $\vdash_{SF}$ for processes as follows:

$$\frac{\forall \mathsf{p} \in \mathcal{P}_G.\ \forall c{:}q \in \Lambda_\mathsf{p}.\ \mathrm{end}(q) \qquad \forall \mathsf{p} \in \mathcal{P}_G.\ \Theta \mid \Lambda_\mathsf{p}, s[\mathsf{p}]: \mathrm{init}(\{\!\{G{\restriction}_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}_G}) \Vdash_{SF} Q_\mathsf{p}}{\Theta \mid \{\Lambda_\mathsf{p}\}_{\mathsf{p}\in\mathcal{P}_G} \vdash_{SF} (\nu s{:}G)\,(\prod_{\mathsf{p}\in\mathcal{P}_G} Q_\mathsf{p})}\ \text{PT-}\nu\text{'}$$

We also define $\vdash_{SF}$ for runtime configurations:

$$\frac{(\vec{q},\xi) \in \mathrm{reach}(\{\!\{G{\restriction}_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}_G}) \qquad \forall \mathsf{p} \in \mathcal{P}_G.\ \forall c{:}q' \in \Lambda_\mathsf{p}.\ \mathrm{end}(q') \qquad \forall \mathsf{p} \in \mathcal{P}_G.\ \Theta \mid \Lambda_\mathsf{p}, s[\mathsf{p}]{:}\vec{q}_\mathsf{p} \Vdash_{SF} Q_\mathsf{p} \qquad \Theta \mid \Lambda' \mid \{s[\mathsf{p}][\mathsf{q}] :: \xi(\mathsf{p},\mathsf{q})\}_{(\mathsf{p},\mathsf{q})\in\mathsf{Chan}_G} \Vdash_{SF} s \blacktriangleright \sigma}{\Theta \mid \{\Lambda_\mathsf{p}\}_{\mathsf{p}\in\mathcal{P}_G}, \Lambda' \mid \emptyset \vdash_{SF} (\nu s{:}G)\,(\prod_{\mathsf{p}\in\mathcal{P}_G} Q_\mathsf{p}) \parallel s \blacktriangleright \sigma}\ \text{RT-}\nu\text{'}$$

If $\Theta \mid \{\Lambda_\mathsf{p}\}_{\mathsf{p}\in\mathcal{P}_G}, \Lambda' \mid \emptyset \vdash_{SF} (\nu s{:}G)\,(\prod_{\mathsf{p}\in\mathcal{P}_G} Q_\mathsf{p}) \parallel s \blacktriangleright \sigma$ holds, we know that we can obtain the premises by inversion. For conciseness, we use the following notation to refer to the CSM configuration $(\vec{q},\xi)$: $\Theta \mid \{\Lambda_\mathsf{p}\}_{\mathsf{p}\in\mathcal{P}_G}, \Lambda' \mid \emptyset \overset{(\vec{q},\xi)}{\vdash_{SF}} (\nu s{:}G)\,(\prod_{\mathsf{p}\in\mathcal{P}_G} Q_\mathsf{p}) \parallel s \blacktriangleright \sigma$.

Intuitively, $\vdash_{SF}$ allows to have one restriction with global type G and requires that all different participants of G are played by different processes in parallel.

Proposition 6.30. Let P be a process, R be a runtime configuration and assume $\vdash \Delta{:}\Theta$. If $\Theta \mid \Lambda \vdash_{SF} (\nu s{:}G)\, P$, then P is restriction-free. If $\Theta \mid \Lambda \mid \emptyset \vdash_{SF} (\nu s{:}G)\, R$, then R is restriction-free.

We state the statements of session fidelity and deadlock freedom. To simplify the statement, and keep it closer to standard statements of MST frameworks, we assume that all projections are sink-final, i.e. there are no intermediate final states, as is always the case for FSMs constructed from local types.

Conjecture 6.31 (Session fidelity with sink-final projections)**.** Let G be a global type such that $G{\restriction}_\mathsf{p}$ is sink-final for every $\mathsf{p} \in \mathcal{P}_G$ and let R be a runtime configuration. We assume that

(1) $\vdash \Delta{:}\Theta$,

(2) $\Theta \mid \Lambda \mid \emptyset \vdash_{SF}^{(\vec{q},\xi)} (\nu s{:}G)\, R$, and
(3) $(\vec{q}, \xi) \to (\vec{q}'', \xi'')$ for some $\vec{q}''$ and ξ''.

Then, there is $(\vec{q}', \xi')$ with $(\vec{q}, \xi) \to (\vec{q}', \xi')$ and R' with $R \to R'$ such that

$$\Theta \mid \Lambda \mid \emptyset \vdash_{SF}^{(\vec{q}',\xi')} (\nu s{:}G)\, R' \enspace .$$

With session fidelity (and subject reduction), one could show deadlock freedom.

Conjecture 6.32 (Deadlock freedom with sink-final projections)**.** Let G be a global type such that $G{\upharpoonright}_{\mathtt{p}}$ is sink-final for every $\mathtt{p} \in \mathcal{P}_G$ and let P be a process. We assume that

- $\vdash \Delta{:}\Theta$,
- $\Theta \mid \Lambda \vdash_{SF} (\nu s{:}G)\, P$,
- $G{\upharpoonright}_{\mathtt{p}}$ is sink-final for every $\mathtt{p} \in \mathcal{P}_G$, and
- $\lceil(\nu s{:}G)\, P\rceil \to^* R$.

Then, it holds that $R \sqsubseteq 0$ or there is R' such that $R \to R'$.

6.5 On Subtyping

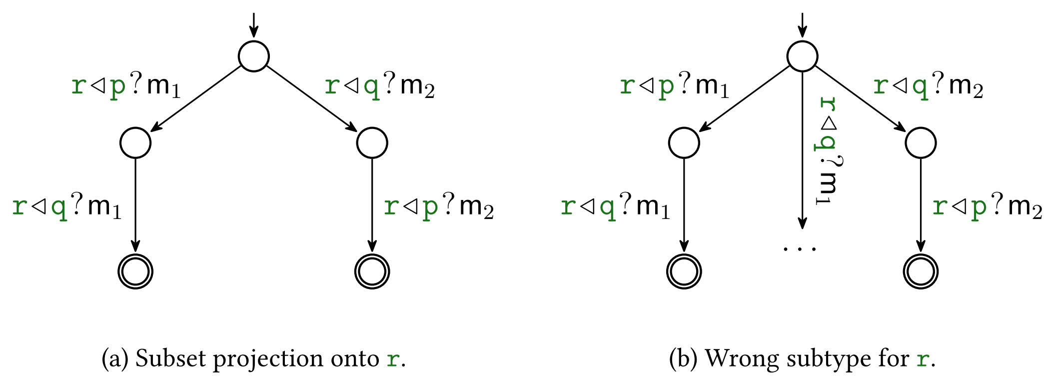
Most MST frameworks employ subtyping in their type system. Intuitively, they allow to remove send branches and to add receive branches [67]. While removing send branches simply disables branches in the global type, adding receive branches is allowed only if they can never be executed. With directed choice, receivers expect to receive, at any time, from one dedicated sender. However, in a setting with sender-driven choice, this is not true and, thus, needs more careful treatment. This is why we only allow not to implement branches for internal choice in our type system.

Example 6.33 (Subtleties of Subtyping with Sender-driven Choice)**.** Consider the global type

$$G := + \begin{cases} \mathtt{p} \to \mathtt{q} : \mathtt{m}_1 \,.\, \mathtt{p} \to \mathtt{r} : \mathtt{m}_1 \,.\, \mathtt{q} \to \mathtt{r} : \mathtt{m}_1 \,.\, 0 \\ \mathtt{p} \to \mathtt{q} : \mathtt{m}_2 \,.\, \mathtt{q} \to \mathtt{r} : \mathtt{m}_2 \,.\, \mathtt{p} \to \mathtt{r} : \mathtt{m}_2 \,.\, 0 \end{cases}$$

The projection onto $\mathtt{r}$ is illustrated in Fig. 6.3a. Following the folklore on subtyping, we can add a receive transition from the initial state to obtain the state machine in Fig. 6.3b. However, this transition will actually be enabled, leading to a deadlock and breaking deadlock-freedom. ◀

The previous example shows that subtyping will also need to account for the availability of messages. We refer to [88] for details while Section 10.2.3 provides background on related work.



(a) Subset projection onto r.

(b) Wrong subtype for r.

Figure 6.3: Subset projection and a wrong subtype.

Chapter 7

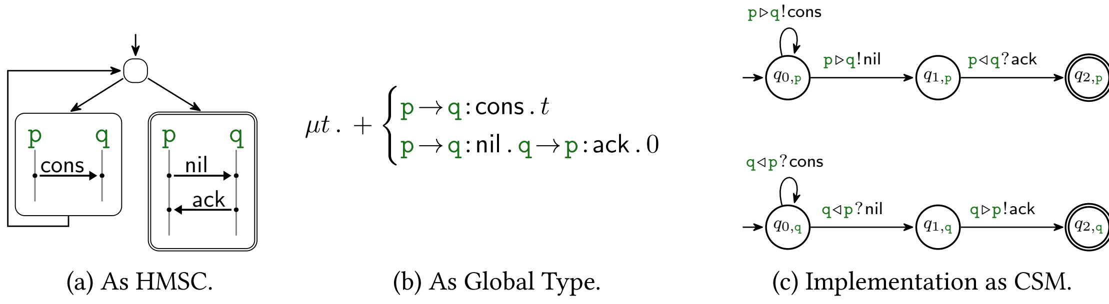
Channel Restrictions of Protocols and Communicating State Machines

In the previous chapter we showed how to integrate CSMs into a session type system. CSMs are a Turing-powerful computational model, making any non-trivial property undecidable in general. Still, with our subset projection, we can obtain a CSM as candidate implementation and use it to check if the global type is implementable. For these checks, the idea of sender-driven choice was crucial as it allowed us to know when a participant committed to a branch. The Turing-completeness proof for CSMs hinges on the fact that channels can be used as memory. It needs to, because each participant has only finite control. We classify restrictions on channels which have been proposed to work around the undecidability of verification questions. We compare half-duplex communication, existential B-boundedness, and k-synchronisability. These restrictions do not prevent the communication channels from growing arbitrarily large but still restrict the power of the model. Each restriction gives rise to a set of languages so, for every pair of restrictions, we check whether one subsumes the other or if they are incomparable. We investigate their relationship in two different contexts: first, the one of protocol specifications and second, the one of CSMs. Surprisingly, these two contexts yield different conclusions.

7.1 Channel Restrictions

In this section, we present different channel restrictions and their implications on decidability of interesting verification questions.

It is important to note that we use the term *restriction* as a property of a system which occurs naturally and not something that is imposed on its semantics. However, both have a tight connection: a system *naturally* satisfies a restriction if *imposing* the restriction does not change its possible behaviours. If this is the case, this can be exploited for algorithmic verification and only check behaviours that satisfy the restriction without harming correctness. Note that, in contrast to this thesis, most of the works we will



(a) As HMSC. (b) As Global Type. (c) Implementation as CSM.

Figure 7.1: List-sending Protocol revisited:
specifications and implementation.

consider here do solely consider finite runs or words. For some, adapting them to a setting with infinite words should be straightforward while this is not necessarily the case for others. We leave a thorough investigation for future work.

Our list-sending protocol from Chapter 2 satisfies all the channel restrictions we consider. Thus, before giving the formal definition, we provide intuition using this example. We recall the protocol specifications and implementation in Fig. 7.1.

7.1.1 Half-duplex Communication

Cécé and Finkel [34, Def. 8] introduced the restriction of half-duplex communication, which intuitively requires that, for any two participants p and q, the channel from p to q is empty before q sends a message to p.

Example 7.1. Communication in the list-sending protocol is half-duplex. At first, the channel from p to q is used to send the list. Only after receiving nil, q sends back the acknowledgement. ◀

Definition 7.2 (Half-duplex)**.** A sequence of events w is called *half-duplex* if for every prefix w' of w and pair of participants p and q, one of the following holds: $\mathcal{V}(w'\Downarrow_{p \triangleright q!_}) = \mathcal{V}(w'\Downarrow_{q \triangleleft p?_})$ or $\mathcal{V}(w'\Downarrow_{q \triangleright p!_}) = \mathcal{V}(w'\Downarrow_{p \triangleleft q?_})$. A language $L \subseteq \Gamma^\infty$ is half-duplex if every word $w \in L$ is half-duplex.

We define the restriction of half-duplex on sequences of events. This definition is equivalent to the original definition by Cécé and Finkel. To be precise, we consider the *natural generalisation* [34, Sec. 4] of the half-duplex definition [34, Def. 8].

Definition 7.3 (Natural generalisation of half-duplex [34])**.** A CSM is called *half-duplex* if for each reachable configuration and for every pair of participants p and q, at most one of the channels (p, q) and (q, p) is non-empty.

Lemma 2.5 relates the channel contents of a CSM upon executing a sequence of events. With this, the equivalence of both definitions follows directly.

Proposition 7.4. Both definitions of half-duplex communication, i.e. Definition 7.2 and Definition 7.3, are equivalent.

Algorithmic Verification. We first consider CSMs with two participants. Checking if its communication is half-duplex is decidable [34, Thm. 31]. The set of reachable configurations is computable in polynomial time [34, Thm. 26] which renders many verification questions like the *unspecified reception problem* decidable (see [34, Def. 13] for a detailed list of verification problems) while model checking PLTL (propositional linear temporal logic) or CTL (computation tree logic) is still undecidable. Half-duplex CSMs with more than two participants are Turing-powerful [34, Thm. 38] so verification becomes undecidable. As alluded to before, most of the prior results have only be shown for finite runs or words but it should be feasible to generalise some of them to the infinite case.

7.1.2 Existential B-boundedness

While the previous property restricts the channel for at least one direction to be empty, one can also bound the size of channels and consider linearisations that are possible adhering to such bounds. On the one hand, one can consider a universal bound that applies for every linearisation. However, this yields finite-state systems [52] and even disallows the list-sending protocol. On the other hand, one can consider an existential bound on the channels which solely asks that there is one linearisation of the distributed run for which the channels are bounded. This allows infinite-state systems and admits the list-sending protocol.

Example 7.5. The list-sending protocol is not universally B-bounded for any B. For any given B, one can easily schedule $\mathtt{p}$ to send $B + 1$ elements of the list before $\mathtt{q}$ starts receiving. Intuitively, existential B-boundedness allows to reorder events using $\sim$. Thus, every receive event by $\mathtt{q}$ can be reordered right after the corresponding send event, yielding existential 1-boundedness of the list-sending protocol. ◀

Definition 7.6 (B-bounded [52])**.** Let $B \in \mathbb{N}$ be a natural number. A word w is *B-bounded* if for every prefix w' of w and pair of participants $\mathtt{p}$ and $\mathtt{q}$, it holds that $|w'{\Downarrow}_{\mathtt{p}\triangleright\mathtt{q}!_}| - |w'{\Downarrow}_{\mathtt{q}\triangleleft\mathtt{p}?_}| \leq B$.

Definition 7.7 (Existentially B-bounded [52])**.** Let $B \in \mathbb{N}$. A prefix MSC M is *existentially B-bounded* if there is a B-bounded linearisation for M. A sequence of events w is *existentially B-bounded* if $\operatorname{msc}(w)$ is defined and *existentially B-bounded.* We may use *not existentially bounded* as abbreviation for not existentially B-bounded for any B.

Remark 7.8 ($\exists B$-boundedness and infinite words)**.** Existential B-boundedness was previously only defined for finite words. However, for infinite words, the semantics of HMSCs do not employ any fairness assumptions on scheduling events from different participants. Thus, the semantics of the list-sending protocol contains $(\mathtt{p}\triangleright\mathtt{q}!\mathit{cons})^\omega$ but also $(\mathtt{p}\triangleright\mathtt{q}!\mathit{cons} \cdot \mathtt{q}\triangleleft\mathtt{p}?\mathit{cons})^\omega$.

With the indistinguishability relation $\sim$ and its extension to infinite words $\preceq^{\omega}_{\sim}$, we capture this phenomenon.

Definition 7.9 ($\exists B$-boundedness for languages with infinite words)**.** A language $L \subseteq \Gamma^{\infty}$ is *existentially B-bounded* if every finite word $w \in L$ is existentially B-bounded and if for every infinite word $w \in L$, there is $w' \in L$ such that $w \preceq^{\omega}_{\sim} w'$ and w' is existentially B-bounded.

Remark 7.10 (An MSC-based alternative)**.** We could have also used MSCs for this definition. Then, the condition for infinite words would have required that, for w with $\operatorname{msc}(w) = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$, there is an existentially B-bounded word $w' \in L$ with $\operatorname{msc}(w') = (N', p', f', l', (\leq'_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$ such that $N \subseteq N'$ and p, l, f, and $(\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$ as well as their counterparts p', l', f', and $(\leq'_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}}$ agree for nodes from N.

Algorithmic Verification. For CSMs, membership is undecidable, unless CSMs are known to be deadlock-free and B is given [52, Prop. 5.5]. For protocols, we will see that they are always existentially B-bounded for some B and thus a correct implementation of a protocol also is. It is quite straightforward that control-state reachability is decidable but not typically studied for these systems [21]. Intuitively, it can be solved by exhaustively enumerating the reachability graph of the CSM while pruning configurations exceeding the bound B. For HMSCs, model checking is undecidable for LTL (linear temporal logic) [7, Thm. 3] and decidable for MSO [92, Thm. 1]. As alluded to before, most of the prior results have only be shown for finite runs or words but it should be feasible to generalise some of them to the infinite case.

7.1.3 k-synchronisability

The restriction of k-synchronisability was introduced for mailbox communication [22] and later refined and adapted to the point-to-point setting [59] – the one we consider.

Intuitively, k-synchronisability requires that every run can be split into alternating phases of at most k send and at most k receive events. Messages are received either in the next receive phase or never. As for B-boundedness, one could distinguish between universal and existential k-synchronisability, i.e. to distinguish the existence of a k-synchronisable linearisation rather than all linearisations being k-synchronisable. However, the universal version does not make much sense in practice. Thus, we omit the term existential.

Example 7.11. Let us consider the list-sending protocol again. As for existential boundedness, it is easy to see that every run can be reordered, using $\sim$, such that every receive event happens right after its corresponding send event. Thus, the list-sending protocol is 1-synchronisable. Note that a receive phase can be empty and, thus, also its infinite runs are 1-synchronisable. ◀

We define k-synchronisability following definitions by Giusto et al. [59, Defs. 6 and 7]. The definition of k-synchronisability builds upon the notion when a prefix MSC is k-synchronous. Its first condition requires that there is some linearisation of the prefix MSC while its second condition requires causal delivery to hold. In contrast to the mailbox setting, the first condition always entails the second condition for the point-to-point setting.

Point-to-point Communication Implies Causal Delivery. We first adapt the definition of causal delivery [59, Def. 4] for point-to-point FIFO channels [59, Sec. 6]. Unfortunately, this discussion leaves room for interpretation what causal delivery exactly is for point-to-point systems. Based on the description that a participant p can receive messages from two distinct participants q and r in any order, regardless of the dependency between the corresponding send events, we decided to literally adapt the definition of causal delivery as follows.

Definition 7.12 (Causal delivery). Let $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$ be an MSC. We say that M satisfies *causal delivery* if there is a linearisation $w = e_1 \ldots$ of N such that for any two events $e_i \leq_M e_j$ with $e_i = \mathtt{p} \triangleright \mathtt{q} !_$ and $e_j = \mathtt{p} \triangleright \mathtt{q} !_$, either e_j is unmatched in w or there are $e_{i'} \leq_M e_{j'}$ such that $e_i \vdash\!\dashv e_{i'}$ and $e_j \vdash\!\dashv e_{j'}$ in w.

We show that $\operatorname{msc}(w)$ for every w (if defined) satisfies causal delivery.

Lemma 7.13. Let $w \in \Gamma^\infty$ be a word for which $\operatorname{msc}(w)$ is defined. Then, $\operatorname{msc}(w)$ satisfies causal delivery.

Proof. Let $w = e_1 \ldots$ be a sequence of events. We claim that w is the witness for causal delivery of $\operatorname{msc}(w)$. Let $e_i \leq_{\operatorname{msc}(w)} e_j$ be two distinct events such that $e_i = \mathtt{p} \triangleright \mathtt{q} !_$ and $e_j = \mathtt{p} \triangleright \mathtt{q} !_$. Notice that $i < j$ since w is a linearisation of $\operatorname{msc}(w)$. We do a case analysis whether e_j is matched in w. Suppose that e_j is unmatched in w, then causal delivery holds. Suppose that e_j is matched in w. Then, there is some $e_{j'}$ with $e_j \leq_{\operatorname{msc}(w)} e_{j'}$ and thus $j < j'$ such that $e_j \vdash\!\dashv e_{j'}$. By definition, it holds that

$$\mathcal{V}((e_1 \ldots e_j) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} !_}) = \mathcal{V}((e_1 \ldots e_{j'}) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ?_}).$$

We know that $\mathcal{V}((e_1 \ldots e_i) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} !_}) \leq \mathcal{V}((e_1 \ldots e_j) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} !_})$. Therefore, there is a prefix $\mathcal{V}((e_1 \ldots e_{i'}) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ?_})$ of the sequence $\mathcal{V}((e_1 \ldots e_{j'}) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ?_})$ for some i' such that $\mathcal{V}((e_1 \ldots e_{i'}) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ?_}) = \mathcal{V}((e_1 \ldots e_i) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} !_})$. Hence, it holds that $e_{i'} \leq_{\operatorname{msc}(w)} e_{j'}$. For causal delivery, it remains to show that $i < i'$. Towards a contradiction, suppose that $i' \leq i$. Since every event either represents a send or receive event, it cannot hold that $i' = i$ and therefore $i' < i$. However, in combination with $\mathcal{V}((e_1 \ldots e_i) \Downarrow_{\mathtt{p} \triangleright \mathtt{q} !_}) = \mathcal{V}((e_1 \ldots e_{i'}) \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ?_})$, $e_1 \ldots e_i$ and, thus, w is not FIFO-compliant. This contradicts the condition for $\operatorname{msc}(\text{-})$ to be defined. Thus, $\operatorname{msc}(w)$ would be undefined, yielding a contradiction. $\square$

In combination with the fact that, given a linearisation w of a prefix MSC M, $\mathrm{msc}(w)$ is isomorphic to M, this yields that causal delivery is satisfied if there is a linearisation.

Corollary 7.14. Every prefix MSC with a linearisation satisfies causal delivery.

With this, we can simplify the definition by omitting this second condition without changing its meaning. In addition, we extend it to apply for MSCs with infinite sets of event nodes.

Definition 7.15 (k-synchronous and k-synchronisable)**.** Let $k \in \mathbb{N}$ be a positive natural number. We say a prefix MSC $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$ is k-*synchronous* if

1. there is a linearisation of event nodes w compliant with $\leq_M$ which can be split into a sequence of k-*exchanges* (also called *message exchange* if k not given or clear from context), i.e. $w = w_1 \ldots$ such that $l(w_i) \in S^{\leq k} \cdot R^{\leq k}$ for all i; and
2. for all e, e' in w such that $e \vdash\!\dashv e'$, there is some i with e, e' in w_i.[1]

A linearisation w is k-synchronisable if $\mathrm{msc}(w)$ is k-synchronous. A language L is k-synchronisable if every word $w \in L$ is. We may use *not synchronisable* as abbreviation for not k-synchronisable for any k.

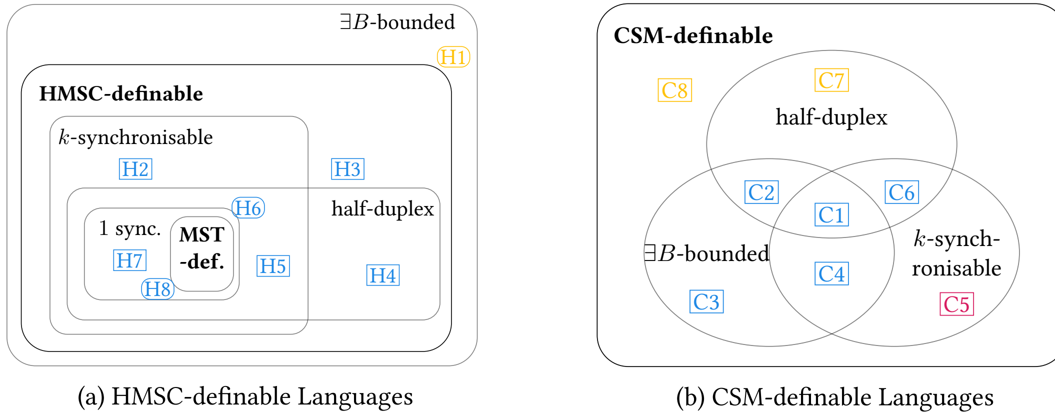
Algorithmic Verification. For CSMs, membership for a given k is decidable in EXPTIME [21, Rem. 30], originally shown decidable by Di Giusto et al. [59], while it is undecidable if k is not given [21, Thm. 22]. For HMSCs, both questions are decidable in polynomial time, while we show that global types are always 1-synchronisable. Model checking for k-synchronisable systems is decidable and in EXPTIME when formulas are represented in LCPDL (propositional dynamic logic with loops and converse). This follows from combining that such systems have bounded (special) tree-width [21, Prop. 28] and results by Bollig and Finkel [20]. Control-state reachability is decidable for k-synchronisable systems [59, Thm. 6]. As alluded to before, most of the prior results have only be shown for finite runs or words but it should be feasible to generalise some of them to the infinite case.

7.1.4 Channel Restrictions and Indistinguishability Relation $\sim$

We show that closing a word or language under $\sim$ does not change the channel restrictions it satisfies.

Theorem 7.16. For FIFO-compliant words, the indistinguishability relation $\sim$ preserves satisfaction of the three channel restrictions under consideration: half-duplex communication, existential B-boundedness, and k-synchronisability.

[1] This is equivalent to the following: for all e and $f(e)$ in w, there is some i with e, $f(e)$ in w_i.



(a) HMSC-definable Languages (b) CSM-definable Languages

Figure 7.2: Comparing half-duplex, existential B-bounded, and k-synchronisable protocols and systems. The results are known results, results are new, and the result disproves an existing result. Hypotheses with rounded corners indicate inclusions while pointed corners indicate incomparability results.

Proof. Let w be a FIFO-compliant word.

First, suppose that w is half-duplex. It is straightforward to check that $\sim$ does not swap any two events $\texttt{q}\triangleleft\texttt{p}?_$ and $\texttt{q}\triangleright\texttt{p}!_$ and therefore the condition for half-duplex is preserved.

Second, suppose that w is existentially B-bounded for some B. Then, we know that $\mathrm{msc}(w)$ is a prefix MSC which admits a B-bounded linearisation. Analogous to the proof of Lemma 2.21, we can show that any (even infinite) prefix MSC is closed under $\sim$. Therefore, for any w' for which $w' \sim w$, it holds that $\mathrm{msc}(w) = \mathrm{msc}(w')$ and the latter still admits the same B-bounded linearisation.

Third, suppose that w is k-synchronisable. Recall that k-synchronisability is also defined on $\mathrm{msc}(w)$ and hence the same reasoning as for the second case applies. □

7.2 Channel Restrictions of Protocols

Figure 7.2a summarises our results. It was only known that every HMSC-definable language is existentially B-bounded for some B [52]. For restrictions which differ (H2 to H5, and H7), we give distinguishing examples. When one restriction subsumes another one (H1, H6, and H8), we prove it. For instance, H6 proves that 1-synchronisability entails half-duplex communication while H5 is an example which is half-duplex, existentially B-bounded, k-synchronisable but not 1-synchronisable.

7.2.1 Channel Restrictions of High-level Message Sequence Charts

We say that an HMSC is half-duplex, existentially B-bounded or k-synchronisable respectively if its language is. It is straightforward that checking an HMSC for k-synchronisability amounts to checking its BMSCs.

Proposition 7.17. An HMSC H is k-synchronisable if and only if all of its BMSCs are k-synchronous.

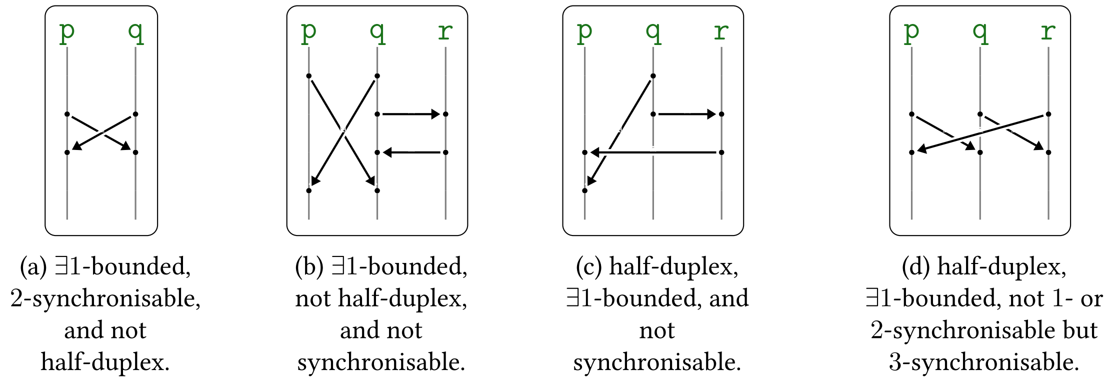
For the presentation of our results, we follow the numbering laid out in Fig. 7.2a. Note that any BMSC can always be turned into a HMSC with a single initial and final vertex. Therefore, it is trivial that all BMSC examples also apply to HMSCs.

Lemma 7.18 ([52], Prop. 3.1)**.** (H1): Any HMSC H is existentially B-bounded for some B.

Proof. We prove this result in a slightly different way than Genest et al. [52]. Basically, one computes the bound for the BMSC of every vertex in H and takes the maximum. This works since every MSC of H is a concatenation of individual BMSCs, which can be scheduled in a way that the channels are empty after each BMSC.

Given a single BMSC M', half the number of events is a straightforward bound on channels. One can also compute a tightest bound B for which M' is existentially B-bounded. For more details, we refer to work by Genest et al. [52, Sec. 3.3] where they define a linearisation function $\mathrm{OPT}(\text{-})$ for MSCs similar to $\mathrm{qOPT}(\text{-})$ in Section 4.1 but optimised for least channel bound. For HMSCs, any $H = (V, E, v^I, V^T, \mu)$ is constructed using a finite number of BMSCs in $\mu(V)$. Consider the maximum bound B of all individual bounds for BMSCs in $\mu(V)$. We claim that H is existentially B-bounded so we need to show that every MSC M of H is existentially B-bounded. By construction, M is the concatenation of (a possibly infinite number of) BMSCs $M'_1 \ldots$. We can construct a linearisation $w = w_1 \ldots$ of M such that w_i is a linearisation of M'_i for every i. By definition of BMSCs, there is a receive event for every send event and hence all channels are empty after $w_1 \ldots w_i$ for every i. Combining these observations yields that w is B-bounded and therefore M is existentially B-bounded. □

Example 7.19. H2: $\exists B$-bounded, k-synchronisable, and not half-duplex. Consider the BMSC in Fig. 7.3a. It is existentially 1-bounded as there is one message per channel, 2-synchronisable since the message exchange can be split into one phase of two sends and two subsequent receives and not half-duplex because both messages can traverse their channel at the same time. ◀



(a) $\exists 1$-bounded, 2-synchronisable, and not half-duplex.

(b) $\exists 1$-bounded, not half-duplex, and not synchronisable.

(c) half-duplex, $\exists 1$-bounded, and not synchronisable.

(d) half-duplex, $\exists 1$-bounded, not 1- or 2-synchronisable but 3-synchronisable.

Figure 7.3: BMSCs that satisfy different channels restrictions.

Example 7.20. H3: $\exists B$-bounded, not half-duplex, and not synchronisable. It is obvious that the BMSC M in Fig. 7.3b is not half-duplex. We show that M is not k-synchronous for any k. Let us denote the event nodes for each participant p with $p_1, \ldots$ as ordered by the total participant order. It is straightforward that one of p_1 and q_1 has to be part of the first k-exchange. However, since the respective corresponding reception happens after the other's event node, both have to be a part of the first k-exchange. Since these receive event nodes (transitively) depend on all other event nodes, all event nodes have to be part of a single k-exchange for M. However, r first has to receive from q in order to send back to it and therefore, there is no single k-exchange for M and M is not k-synchronous for any k. ◀

Example 7.21. H4: half-duplex, $\exists B$-bounded, and not synchronisable. Let us consider the BMSC in Fig. 7.3c. It is straightforward that it is half-duplex and existentially 1-bounded. However, it is not k-synchronisable for any k. In particular, the first and last event node (of any total order induced by the BMSC) must belong to the same message exchange but two more linearly dependent message exchanges need to happen in between. ◀

Example 7.22. H5: half-duplex, $\exists B$-bounded, k-synchronisable but not 1-synchronisable. Consider the BMSC in Fig. 7.3d. It is easy to see that it is neither 1- nor 2-synchronisable but 3-synchronisable, half-duplex and existentially 1-bounded. Note that it is straightforward to amend the example in a way that it is still half-duplex but the parameters B and k need to be increased. ◀

Lemma 7.23. (H6): Every 1-synchronisable HMSC is half-duplex.

Proof. Intuitively, in any BMSC of an HMSC, every send event node has a corresponding receive event. Therefore, a message that has been sent needs to have been received directly afterwards to be 1-synchronisable. The per-participant order is total so any participant has to receive a message before it sends a message back.

Formally, let H be an 1-synchronisable HMSC. We show that $\mathcal{L}(H)$ is half-duplex. To this end, it suffices to show that $\mathcal{L}(M)$ for every MSC M of H is half-duplex. Let $M = (N, p, f, l, (\leq_{\mathtt{p}})_{\mathtt{p} \in \mathcal{P}})$ be an MSC of H. By assumption, M is 1-synchronisable. Let $\bar{w}$ be the linearisation of M from Definition 7.15. Since M is an MSC and the way $\bar{w}$ is chosen as witness for 1-synchronisability, every send event is immediately followed by its corresponding receive event in $\bar{w}$.

Let $w \in \mathcal{L}(M)$ be any word. We show that w is half-duplex. Towards a contradiction, suppose that the channel from $\mathtt{q}$ to $\mathtt{p}$ is not empty and $\mathtt{p}$ attempts to send a message to $\mathtt{q}$ after the prefix $w_1 \ldots w_j$ of w: $w_{j+1} = \mathtt{p} \triangleright \mathtt{q} ! _$ and $\mathcal{V}(w_1 \ldots w_j \Downarrow_{\mathtt{p} \triangleleft \mathtt{q} ? _}) \leq \mathcal{V}(w_1 \ldots w_j \Downarrow_{\mathtt{q} \triangleright \mathtt{p} ! _})$. Thus, there is $i \leq j$ such that $w_i = \mathtt{q} \triangleright \mathtt{p} ! _$ which is unmatched. By definition, the per-participant order $\leq_M \cap \, (p^{-1}(\mathtt{p}) \times p^{-1}(\mathtt{p}))$ is total for every participant $\mathtt{p}$. In the linearisation $\bar{w}$, the corresponding receive event happens directly after w_i so the next event by $\mathtt{p}$ must be the corresponding reception, which contradicts the assumption that w_{j+1} is a send event of $\mathtt{p}$. □

7.2.2 Channel Restrictions of Global Types

Example 7.24. [H7]: half-duplex, ∃1-bounded, 1-synchronisable but not in MSTs. Intuitively, 1-synchronisability ensures that every receive event can happen immediately after its send event, precisely what message interactions in global types specify. With Proposition 2.26, we showed that $G = 0$ is the only global type whose semantics contains ε. Thus, any half-duplex, existentially 1-bounded and 1-synchronisable language L that contains ε and $|L| > 1$ cannot be represented as non-deterministic global type, which is most expressive class of global types. If ε is not in the semantics, there actually is always is a non-deterministic global type. This is surprising, considering the syntactic restrictions, but we prove this in Section 8.2.2. For global types with directed or sender-driven choice, a language cannot contain any prefix of a maximal trace. The reason is that $\mathtt{p} \to \mathtt{q} : \mathsf{m} + 0$ is syntactically invalid in a global type. For mixed-choice, the situation is more complicated: it can be satisfied by using message interactions that are related by $\sim$. For instance, consider

$$G := \mathtt{p} \to \mathtt{q} : \mathsf{m} \,.\, 0 + \mathtt{r} \to \mathtt{s} : \mathsf{m} \,.\, \mathtt{p} \to \mathtt{q} : \mathsf{m} \,.\, 0 \ .$$

Its semantics contains both

$$w_1 := \mathtt{p} \triangleright \mathtt{q} ! \mathsf{m} \cdot \mathtt{q} \triangleleft \mathtt{p} ? \mathsf{m} \quad \text{and}$$
$$w_2 := \mathtt{r} \triangleright \mathtt{s} ! \mathsf{m} \cdot \mathtt{s} \triangleleft \mathtt{r} ? \mathsf{m} \cdot \mathtt{p} \triangleright \mathtt{q} ! \mathsf{m} \cdot \mathtt{q} \triangleleft \mathtt{p} ? \mathsf{m} \ .$$

It holds that $w_1 \in \text{pref}(\mathcal{C}^{\sim}(w_2)) \subseteq \mathcal{L}(G)$. Our workflow in Section 8.2.2 also applies to sink-final state machines over Σ without mixed-choice states and preserves their given choice restrictions (cf. Lemma 8.37). ◀

We show that protocols specified as global types satisfy all discussed channel restrictions (with the minimal reasonable parameter).

Theorem 7.25. (H8): Let G be a non-deterministic global type. Its semantics $\mathcal{L}(G)$ is half-duplex, existentially 1-bounded, and 1-synchronisable.

Proof. The semantics of G are defined using the state machine $\mathsf{GAut}(G)$. Its language $\mathcal{L}(\mathsf{GAut}(G))$ solely consists of words where send events are immediately followed by receive events. Thus, it is half-duplex, existentially 1-bounded, and 1-synchronisable by construction. With Theorem 7.16, we showed that all channel restrictions are preserved by $\sim$. Thus, it holds that $\mathcal{C}^{\sim}(\mathsf{GAut}(G))$ and, thus, $\mathcal{L}(G)$ is half-duplex, existentially 1-bounded and 1-synchronisable. □

Remark 7.26 (Choreography automata are half-duplex, ∃1-bounded, and 1-synchronisable)**.** In this section, we looked at global types from MSTs, which are rooted in process algebra. With *choreography automata* [11], a similar concept has been studied using automata. Basically, a protocol specification is an automaton whose transitions are labelled by $\mathtt{p} \to \mathtt{q} : \mathtt{m}$. In contrast to global types, they do not impose constraints on choice, i.e. there does not need to be a unique participant chooses which branch to take next and do not employ an indistinguishability relation but require to explicitly spell out all possible reorderings. This feature can lead to complications with regard to implementing such protocols but does not change the satisfaction of channel restrictions. In fact, protocols specified by choreography automata are also half-duplex, existentially 1-bounded, and 1-synchronisable.

7.3 Channel Restrictions of CSMs

We say that an CSM is half-duplex, existentially B-bounded, or k-synchronisable respectively if its language is. Again, we follow the outline presented in Fig. 7.2b.

Example 7.27. [C1]: half-duplex, $\exists B$-bounded, and k-synchronisable. The implementation for every implementable global type is ∃1-bounded, 1-synchronisable, and half-duplex, e.g. the CSM in Figure 7.1c. ◀

Any BMSC can easily be implemented with an CSM by simple letting each participant follow its linear trajectory of event nodes. We call this projection. Therefore, we can use three of the BMSCs presented in Fig. 7.3 to show the hypotheses for CSMs:

Example 7.28. For, [C2], the projection of Fig. 7.3c (used to show [H4]) is half-duplex, $\exists B$-bounded, and not synchronisable. For [C3], the projection of Fig. 7.3b (used to show [H3]) is $\exists B$-bounded, not half-duplex, and not synchronisable. For [C4], the projection of Fig. 7.3a (used to show [H2]) is $\exists B$-bounded, k-synchronisable, and not half-duplex. ◀

Figure 7.4: CSM with FSMs for p (left) and for q (right).

Example 7.29. C5: k-synchronisable, not half-duplex and not $\exists$-bounded; C6: k-synchronisable, half-duplex and not $\exists$-bounded. We consider two CSMs constructed from the state machines in Fig. 7.4. For C5, we consider the CSM consisting of both state machines. It is 1-synchronisable but not existentially bounded and not half-duplex. It is 1-synchronisable because every linearisation can be split into single send events that constitute 1-exchanges. It is neither existentially B-bounded for any B nor half-duplex since none of the messages will be received so both channels can grow arbitrarily. For C6, it can easily be turned into a half-duplex CSM by removing one of the send events. Then, the CSM is 1-synchronisable and half-duplex but not existentially bounded. ◀

This example disproves a result from the literature [87, Thm. 7.1], which states that every k-synchronisable system is existentially B-bounded for some B. In the proof, it is neglected that unreceived messages remain in the channels after a message exchange. Our example satisfies their assumption that CSMs only have states that are either final, have send options to choose from, or receive options to choose from. We do not impose any assumptions on states of CSMs in this work. Still, all the presented examples do satisfy this condition so the presented relationships also hold for this subset of CSMs.

Corollary 7.30. Existential B-boundedness and k-synchronisability for CSMs are incomparable.

The previous result follows immediately from the CSMs constructed in Example 7.29. Our result considers the point-to-point FIFO setting. For the mailbox setting, the analogous question is an open problem [22].

Turing-powerful Encodings. On the one hand, it is well-known that CSMs are Turing-complete [23] and Cécé and Finkel [34, Thm. 36] showed that half-duplex communication does not impair expressiveness of CSMs with more than two participants. On the other hand, each of existential B-boundedness and k-synchronisability render some verification questions decidable. Therefore, the encodings of Turing-completeness [23, 34] are examples for CSMs which are not existentially B-bounded for any B nor k-synchronisable for any k and either half-duplex (C7) or not half-duplex (C8).

Chapter 8

A Unifying Protocol Specification Formalism

So far, we considered HMSCs and global types from MSTs as protocol specifications. In previous chapters, we showed that global types can be encoded as HMSCs and that both satisfy a number of channel restrictions, including existential boundedness. However, HMSCs cannot specify all such protocols. In this chapter, we define *protocol state machines* as novel protocol specification formalism. Intuitively, it is a finite state machine where transitions are labelled with send and receive events. On the one hand, we will show that this formalism subsumes both global types and HMSCs. On the other hand, we will also show that some of the structural restrictions of global types do actually not impede expressivity.

8.1 Protocol State Machines

In contrast to global types and HMSCs, we will allow to specify send and receive events individually rather than jointly. Thus, we need to require that channels are used in a FIFO manner.

Definition 8.1. We define the set of B-bounded FIFO words:
$\mathsf{FIFO}_B := \{w \in \mathsf{FIFO} \mid w \text{ is } B\text{-bounded}\}$.

We define *protocol state machines* as syntactic objects that characterise existentially bounded finite-control protocols.

Definition 8.2 (Protocol State Machine (PSM))**.** A dense FSM $P = (Q, \Gamma, \delta, q_0, F)$ is a B-PSM if $\mathcal{L}(P) \subseteq \mathsf{FIFO}_B$. The semantics of P is defined as $\mathcal{S}(P) := \mathcal{C}^{\sim}(\mathcal{L}(P))$. Moreover, P is a PSM if it is a B-PSM for some B.

We restrict PSMs to be dense, which means that there can be ε-transitions but they are the only ones from this state. This simplifies the definition of choice restrictions for PSMs. In fact, in terms of expressivity, we could have assumed there are no ε-transitions, as it is standard to remove ε-transitions for FSMs. However, our definition allows us to specify the semantics of global types as PSMs. Without loss of generality, we assume there are no ε-transitions in a PSM P if $\mathcal{L}(P) = \{\varepsilon\}$.

Note that we require $\mathcal{L}(P) \subseteq \mathsf{FIFO}_B$ in our definition. For most protocols, it should be straightforward to check if this holds. If one only uses transitions of shape $_ \to _ : _$, this trivially holds. Thus, one only needs to check for channels $(\mathtt{p}, \mathtt{q})$ for which transitions of shape $\mathtt{p} \triangleright \mathtt{q} ! _$ and $\mathtt{q} \triangleleft \mathtt{p} ? _$ occur. For these, it is then, for instance, necessary that loops have no net effect on the channel.

Still, let us elaborate how to check $\mathcal{L}(P) \subseteq \mathsf{FIFO}_B$ algorithmically. Intuitively, one projects P onto each channel to obtain an FSM A and checks if every word in A is in FIFO_B. For finite words, this is rather simple: one constructs a finite state machine for FIFO_B, complements it and checks if the intersection with A is empty. If so, A only contains words in FIFO_B. If not, there is at least one word that is either not B-bounded or not FIFO-compliant. To properly account for infinite words, the situation is slightly more difficult. Recall that we defined the semantics such that infinite words are contained if they have an infinite run in the state machine. Thus, we cannot apply the same technique for infinite words. However, FIFO_B is prefix-closed. This is why we can add a bad state to the automaton for FIFO_B that is reached if the conditions for FIFO_B are not satisfied. We construct the synchronous cross-product with A and check if the added bad state is reachable. This also checks if finite words are FIFO-compliant.

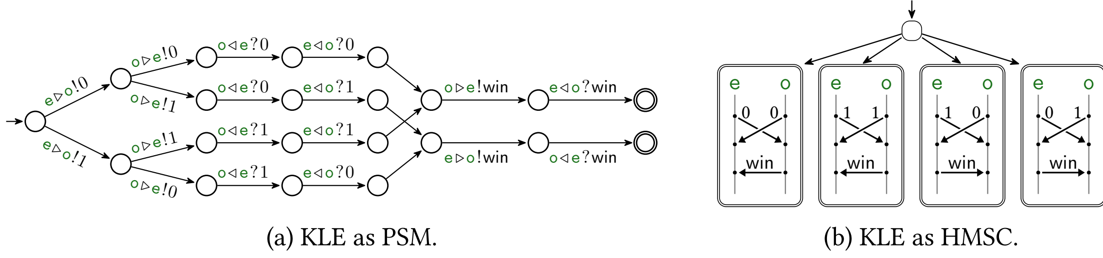
For B, a finite state machine for FIFO_B consists of one state for every possible channel of size at most B, yielding

$$\sum_{i=0}^{B} |\mathcal{V}|^i = \frac{|\mathcal{V}|^{B+1} - 1}{|\mathcal{V}| - 1} \in O(|\mathcal{V}|^{B+1})$$

states, provided there is more than one message in $\mathcal{V}$. We have to construct the synchronous product of size $O(|P| \cdot |\mathcal{V}|^{B+1})$ for every channel. Note that we would not need to construct FIFO_B upfront but could construct the necessary parts on-the-fly. This should yield significant run time improvements in practice. If B is not given, it should be sufficient to consider the number of transitions (of the projected automaton) as upper bound for B. If one checks iteratively, one can stop as soon as one finds a word which is not FIFO-compliant and one only needs to continue if the bound might have been too small.

Let us give an example for a PSM.

Example 8.3 (Kindergarten Leader Election)**.** We consider a protocol between two participants $\mathtt{e}$ (evens) and $\mathtt{o}$ (odds). It can be used to quickly settle a dispute between children (hence the name). Both children pick 0 or 1 and tell each other their pick at the same time. Child $\mathtt{e}$ wins if the sum is even while $\mathtt{o}$ wins if the sum is odd. The protocol is visualised as PSM in Fig. 8.1a. At the end, the looser concedes by sending the message win to the winner. Note that this communication behaviour hinges on the possibility to specify send and receive events independently. If one tries to model this protocol with a global type, one child will always know the choice of the other before picking its number,



(a) KLE as PSM.

(b) KLE as HMSC.

Figure 8.1: Kindergarten Leader Election (KLE).

undermining the purpose of the protocol. We also depict the protocol as an HMSC in Fig. 8.1b. ◀

By definition, PSMs specify FIFO languages. Still, they might still be unruly: for example, they might describe protocols where not all channels are empty at the end of a finite word. MSTs and HMSCs have syntactic restrictions that ensure the described protocol is sane. Let us recall and define some sanity properties.

We a word is complete if it is infinite or all send and receive events are matched (Definition 2.3).

A sanity check that may be desirable in some applications is to check whether the protocol is (always) terminating.

Definition 8.4 (Terminating)**.** A language L is *terminating* if $L \subseteq L_{\text{fin}}$. A PSM P is terminating if $\mathcal{S}(P)$ is terminating.

Another sanity check is whether the protocol *can* be terminated. We already defined this notion for global types and HMSCs (Definition 4.16). For languages, it can be defined as follows.

Definition 8.5 (0-reachability)**.** A language L is 0-*reachable* if $\forall w \in L_{\text{inf}}. \forall w' \leq w. \exists w_{\text{fin}} \in L_{\text{fin}}.\ w' \leq w_{\text{fin}}$. In other words, a 0-reachable language is one where all prefixes of infinite words can be completed to finite accepted words. A PSM P is 0-reachable if $\mathcal{S}(P)$ is 0-reachable.

Lemma 8.6 (Sanity checks for PSMs)**.** The following properties are all decidable for B-PSMs: 1. emptiness ($\mathcal{S}(P) = \emptyset$); 2. checking membership for finite words; 3. completeness; 4. termination; 5. 0-reachability.

Proof. Let P be a B-PSM. Emptiness is trivial. Word membership is a consequence of the finiteness of $\mathcal{C}^{\sim}(\{w\})$ for finite w. To check if all words are complete, it is straightforward to adapt the procedure that checks inclusion in FIFO_B as described above. Both remaining properties are preserved and reflected by $\mathcal{C}^{\sim}(\text{-})$ so it is sufficient to check them on the core language $\mathcal{L}(P)$. Termination can be checked by checking for loops. 0-reachability holds if from every reachable state there is a path to a final one. □

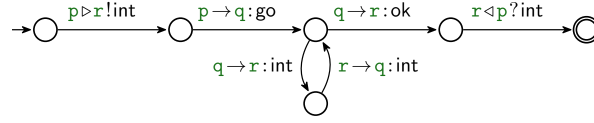


Figure 8.2: A PSM whose semantics cannot be represented as HMSC. Transitions labelled with p→q:m should be interpreted as emitting the sequence p▷q!m, q◁p?m.

Remark 8.7 (On infinite words)**.** While we require finite words to end in a final state, an infinite word is in the semantics of a PSM (but also global type) if it has an infinite runs. This can be turned into a (very particular kind of) Büchi automaton: one where we mark all states and add self-loops to previously final states, ignoring the transition label in the semantics. It is also mostly the infinite words which make sanity checks harder. We conjecture that sanity checks are decidable if they can be represented as Büchi automaton that preserves and reflects $\sim$, a key assumption that allows to only check the core language. We consider both the generalisation of semantics to Büchi conditions and investigating a more general class of sanity checks as interesting directions for future work.

PSMs are more expressive than HMSCs. Using loops, they allow protocols with an arbitrarily long duration between a send event and its corresponding receive event. Figure 8.2 is an example of a PSM that cannot be expressed as an HMSC. In that protocol, p commits to some integer (abstracted as the label int) at the beginning by sending it to r and sends a go signal to q. Note that here we use the paired send and receive notation p→q:ok to emit the two events in sequence. Then q and r engage in some negotiation of arbitrary length until q decides to exit the loop, at which point r is finally allowed to receive the message sent by p. No HMSC can represent such a protocol: the matching events p▷r!int and r◁p?int are separated by an arbitrary number of events (with no opportunity for reordering up to $\mathcal{C}^{\sim}(\text{-})$); since in HMSCs matching events need to belong to the same basic block, such a block would need to contain the arbitrarily many events in between as well, which is impossible.

As for global types and HMSCs, we define various restrictions on choice as structural property of PSMs.

Definition 8.8 (Mixed, sender-driven and directed choice for PSMs)**.** Let $P = (Q, \Gamma, \delta, q_0, F)$ be a PSM. We say P satisfies *mixed choice* if it is deterministic, P is a *sender-driven* PSM if there is a function $\lambda \colon Q \to \mathcal{P}$ such that for all states q, q' with $q \xrightarrow{x} q'$ with $x \in \Gamma_!$, it holds that $\lambda(q)$ is the sender for x, i.e. $x = \lambda(q) \triangleright _!_$. We call P *directed*, if, for every state q, there is q such that all transition labels from q are of the form $\lambda(q) \triangleright$q$!_$,

It is straightforward to construct λ from a PSM. This attempt simply fails if it does not exist.

8.2 Expressivity Results

In this section, we first show that the FSM for the semantics of every global type is a special kind of PSM. Second, we show that the structural restrictions on global types do not impede expressivity, compared to defining protocols as PSMs with languages over Σ where every final state is a sink state. Third, we show that every HMSC can be transformed into a PSM with the same semantics.

8.2.1 Global Types as Special Class of PSMs

We showed that the semantics of global types are $\exists 1$-bounded, which requires a bound of 1 per channel. In fact, global types are even more restrictive: all channels together are bounded by 1. To capture this, we introduce the notion of existential $\sum$-boundedness.

Definition 8.9 (ΣB-bounded)**.** Let $B \in \mathbb{N}$ be a natural number. A word w is ΣB-*bounded* if for every prefix w' of w, the following holds:

$$\sum_{\mathtt{p} \neq \mathtt{q} \in \mathcal{P}} \left(|w' \Downarrow_{\mathtt{p} \triangleright \mathtt{q} ! _}| - |w' \Downarrow_{\mathtt{q} \triangleleft \mathtt{p} ? _}| \right) \leq B \enspace .$$

Definition 8.10 (Existentially ΣB-bounded)**.** Let $B \in \mathbb{N}$. A word w is *existentially* ΣB-*bounded*, also denoted by $\exists \Sigma B$-bounded if there is a ΣB-bounded word w' such that $w' \sim w$. Let $L \subseteq \Gamma^\infty$ be a language. We say that L is $\exists \Sigma B$-bounded if every word in L_{fin} is $\exists \Sigma B$-bounded and for every $w \in L_{\text{inf}}$, there is w' such that $w \preceq^\omega_\sim w'$ and w' is existentially $\exists \Sigma B$-bounded.

For any global type G, the language of $\mathsf{GAut}(G)$ is $\Sigma 1$-bounded. Thus, its semantics is $\exists \Sigma 1$-bounded.

Proposition 8.11. The semantics $\mathcal{L}(G)$ for every global type is $\exists \Sigma 1$-bounded.

We define restrictions on PSMs that describe the implicit restrictions of global types when regarded as PSMs.

Definition 8.12. A PSM P is a $\Sigma 1$-PSM if its core language $\mathcal{L}(P)$ is $\Sigma 1$-bounded.

Recall that we call a PSM sink-final if each state is final if and only if it is a sink state.

Proposition 8.13. Let G be a global type. Then, $\mathsf{GAut}(G)$ is a sink-final $\Sigma 1$-PSM. If G is non-deterministic (mixed-choice, sender-driven, or directed respectively), then $\mathsf{GAut}(G)$ is non-deterministic (mixed-choice, sender-driven, or directed respectively).

8.2.2 From $\Sigma 1$-PSM to Global Types

We explain a workflow to compute a global type from any $\Sigma 1$-PSM whose language does not contain ε. From Proposition 2.26, we know that 0 is the only global type to specify ε. Thus, the only PSM P with ε in its semantics, for which we can find a global type, is the one where $\mathcal{S}(P) = \{\varepsilon\}$, which we call *trivial* PSM. For a sink-final PSM, it is easy to see that it is either the trivial PSM or does not contain ε. Given a sink-final sender-driven $\Sigma 1$-PSM, this workflow yields a sender-driven global type.

Theorem 8.14. For every sink-final $\Sigma 1$-PSM P, there is a global type G with the same core language. If P is non-deterministic (mixed-choice, sender-driven, or directed respectively), G is non-deterministic (mixed-choice, sender-driven, or directed respectively). Every $\Sigma 1$-PSM P with $\varepsilon \notin \mathcal{S}(P)$ can be represented as a non-deterministic global type with the same core language.

For $\Sigma 1$-PSMs that are not sink-final, our workflow yields non-deterministic global types, a trait that cannot be avoided when preserving the protocol.

Most of the workflow applies to more general alphabets, which we also elaborate on at the end of this section. Here, we choose $\Sigma 1$-PSMs, which allows us to reason how sender-driven choice is preserved. The reasoning for mixed and directed choice is analogous and, thus, omitted for conciseness.

Overview of the Workflow for Sender-driven PSMs

(0) make the PSM sink-final for the price of introducing non-determinism

(1) compute a regular expression for the initial state of the sink-final PSM

(2) convert regular expression to an ancestor-recursive, non-merging, dense, and intermediate-recursion-free PSM

(3) if the original PSM is a $\Sigma 1$-PSM, transform the result from the previous step to a global type

Without loss of generality, we assume that every sink state is final. Any state, for which this is not the case, can simply be removed while preserving the core language and semantics of a PSM.

For the last step, we only consider $\Sigma 1$-PSMs because global types always jointly specify send and receive events.

Step (0): Sink State iff Final State

This can be considered to be a preprocessing step for PSMs that are not sink-final, making the workflow more general. This transformation step simply introduces a new final sink state to which transitions can lead non-deterministically.

We give a construction with a single fresh final state, which can be (non-deterministically) reached instead of any previous final state.

Procedure 8.15 (PSM: Sink State iff Final State)**.** Let $P = (Q, \Gamma, \delta, q_0, F)$ be a PSM with $\varepsilon \notin \mathcal{S}(P)$. We define a function that turns P into a sink-final PSM:

$$\text{psm2sink-final-psm}(P) := (Q \uplus \{q_f\}, \Gamma, \delta', q_0, \{q_f\})$$

where $(q_1, x, q_2) \in \delta'$ if $(q_1, x, q_2) \in \delta$ as well as $(q_1, x, q_f) \in \delta'$ if $(q_1, x, q_2) \in \delta$ and $q_2 \in F$.

The condition that $\varepsilon \notin \mathcal{S}(P)$ ensures that there is a predecessor state for every final state to which we can add the transition.

Proposition 8.16. For any Σ1-PSM P with $\varepsilon \notin \mathcal{S}(P)$, the PSM $\text{psm2sink-final-psm}(P)$ is sink-final.

It is obvious that this construction introduces non-determinism and, thus, does not preserve sender-driven choice.

Step (1): From Sink-final PSMs to Regular Expressions

This transformation step translates a sink-final PSM to a regular expression over Γ that specifies the same core language. It is well-known that this can be done using Arden's Lemma [9]. We cannot apply the standard technique though, as it would produce as many regular expressions as final states. Such treatment makes it very hard to argue about the preservation of sender-driven choice. Instead, we exploit the fact that the PSM is sink-final and produce a single regular expression for the initial state. This also enables the treatment of infinite words, which solely require an infinite run that necessarily does not end in a final state.

We define regular expressions and include infinite words in their semantics.

Definition 8.17 (Regular Expressions)**.** Let Δ be an alphabet. Regular expressions (REs) over Δ are inductively defined by the following grammar where $a \in \Delta$:

$$r ::= \varepsilon \mid a \mid r + r \mid r \cdot r \mid r^*$$

The concatenation operator $\cdot$ has precedence over $+$. We define $\mathcal{L}_{\text{fin}}(a) = \{a\}$, $\mathcal{L}_{\text{fin}}(r_1 + r_2) = \mathcal{L}_{\text{fin}}(r_1) \cup \mathcal{L}_{\text{fin}}(r_2)$, $\mathcal{L}_{\text{fin}}(r_1 \cdot r_2) = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}_{\text{fin}}(r_1), w_2 \in \mathcal{L}_{\text{fin}}(r_2)\}$, and $\mathcal{L}_{\text{fin}}(r^*) = \{w_1 \ldots w_n \mid n \in \mathbb{N}, \forall i \leq n.\, w_i \in \mathcal{L}_{\text{fin}}(r)\}$. The infinite language $\mathcal{L}_{\text{inf}}(r)$ is defined as $\{w \in \Delta^\omega \mid \forall w' \in \text{pref}(w).\, w' \in \text{pref}(\mathcal{L}_{\text{fin}}(r))\}$. The language $\mathcal{L}(r)$ is the union of $\mathcal{L}_{\text{fin}}(r)$ and $\mathcal{L}_{\text{inf}}(r)$. The function $\text{sym}(r)$ is the set of all letters in r, i.e. the smallest subset $\Delta' \subseteq \Delta$ such that $\mathcal{L}(r) \subseteq (\Delta')^\infty$. We denote the set of all regular expressions over Δ with $\mathcal{R}_\Delta$.

For a state machine, an infinite word is part of its semantics if there is an infinite run. Here, we mimic this idea: an infinite word is in the semantics of a regular expression if every prefix of the word is a prefix of a word in the finite semantics.

Instead of constructing the regular expressions for final states, as is standard with Arden's Lemma, we construct one for the initial state. This is sound because a state is a sink if and only if it is final. It also lets us handle infinite words.

Lemma 8.18 (Arden's Lemma – Swapped)**.** Let r_1 and r_2 be two regular expressions over an alphabet Δ. If r_1 does not contain the empty string, i.e. $\varepsilon \notin \mathcal{L}_{\text{fin}}(r_1)$, then $r_3 = r_2 + (r_1 \cdot r_3)$ has a unique solution that is $r_3 = r_1^* \cdot r_2$.

Proof. The proof is analogous to the original one:

$$\begin{aligned} r_3 &= r_2 + r_1 \cdot r_3 \\ &= r_2 + r_1 \cdot (r_2 + r_1 \cdot r_3) \\ &= r_2 + r_1 \cdot r_2 + r_1 \cdot r_1 \cdot r_3 \\ &= \ldots \\ &= r_2 + r_1 \cdot r_2 + r_1^2 \cdot r_3 + r_1^3 \cdot r_3 + \ldots \\ &= (\varepsilon + r_1 + r_1^2 + r_1^3 + \ldots) \cdot r_2 \\ &= r_1^* \cdot r_2 \end{aligned}$$

□

For sender-driven PSMs, we want to show that sender-driven choice is preserved. Therefore, we need a notion of sender-driven choice for regular expressions. We define this following work on deterministic regular expressions [26].

Definition 8.19 (Marking and unmarking regular expressions)**.** Let Δ be an alphabet and $r \in \mathcal{R}_\Delta$ be a regular expression. We define a function $\text{mark}(r)$ that simply subscripts every letter in r with a distinct index and the inverse function $\text{unmark}(r)$, which is also defined for words over Δ.

Definition 8.20 (Mixed-choice, sender-driven and directed regular expressions)**.** Let $r \in \mathcal{R}_\Gamma$. We say that r is a *sender-driven* regular expression if the following holds: for every $u \in \Gamma^*$ and $x, y \in \Gamma$, if $ux \in \text{pref}(\mathcal{L}(\text{mark}(r)))$, $uy \in \text{pref}(\mathcal{L}(\text{mark}(r)))$ and $x \neq y$, then $\text{unmark}(x) \neq \text{unmark}(y)$ as well as $\text{unmark}(x) \in \{\texttt{p}\triangleright\texttt{q}!_ \mid \texttt{q} \in \mathcal{P}\}$ and $\text{unmark}(y) \in \{\texttt{p}\triangleright\texttt{q}!_ \mid \texttt{q} \in \mathcal{P}\}$ for some $\texttt{p} \in \mathcal{P}$. For *directed choice*, we also require $\texttt{q}$ to be the same for both x and y and, for *mixed choice*, we solely require $\text{unmark}(x) \neq \text{unmark}(y)$.

Compared to deterministic regular expressions, our definition requires the special alphabet Γ (and adds a condition for sender-driven and directed choice).

Proposition 8.21. Every mixed-choice, sender-driven or directed regular expression is a deterministic regular expression.

Definition 8.22. Let Δ be an alphabet and $L \subseteq \Delta^\infty$. We define a function that collects all first letters of L: $\text{first}(L) := \text{pref}(L) \cap \Delta$. The function $\text{follow}(L, a)$ collects all letters that can occur after a in L: $\text{follow}(L, a) := \{b \mid wab \in \text{pref}(L)\}$ and $\text{follow}(L, \varepsilon) := \text{first}(L)$.

The following lemma follows from a straightforward adaption of Lemma 2.2 by Brüggemann-Klein et al. [26].

Lemma 8.23. An RE $r \in \mathcal{R}_\Gamma$ is a sender-driven RE if and only if, for every $z \in \text{sym}(\text{mark}(r)) \uplus \{\varepsilon\}$ and every $x, y \in \text{follow}(\text{mark}(r), z)$, if $x \neq y$, then $\text{unmark}(x) \neq \text{unmark}(y)$, as well as $\text{unmark}(x) \in \mathcal{L}(\Gamma_{\mathtt{p}})$ and $\text{unmark}(y) \in \mathcal{L}(\Gamma_{\mathtt{p}})$ for some $\mathtt{p} \in \mathcal{P}$.

Intuitively, one can check if an RE over Γ is an sender-driven RE as follows. For every subexpression of the form $r_1 + r_2$ and $r_1^* \cdot r_2$, the REs r_1 and r_2 should not share any first letters and the union of their first letters belongs to the same participant. It suffices to consider these operators as these are the only ones where lookahead to take a decision about the path in the RE is needed.

Procedure 8.24 (PSM to RE)**.** Let $P = (Q, \Gamma, \delta, q_0, F)$ be a sink-final PSM. We generate a system of equations. For every $q_1 \in Q$, we introduce r_{q_1} as follows:

$$r_{q_1} = \sum_{(q_1, x, q_2) \in \delta} x \cdot r_{q_2}$$

Given the initial state q_0, we can solve the system of equations for r_{q_0} with Lemma 8.18, yielding a regular expression $\text{psm2regex}(P)$.

The following lemma states the correctness of the previous procedure.

Lemma 8.25. For every sink-final PSM P, it holds that $\mathcal{L}(\text{psm2regex}(P)) = \mathcal{L}(P)$. If P is a sender-driven PSM, then $\text{psm2regex}(P)$ is a sender-driven RE. If $\varepsilon \notin \mathcal{S}(P)$, then ε does not occur in $\text{psm2regex}(P)$.

Proof. With the sink-final assumption, the first claim easily follows from Lemma 8.18. For the second claim, let us investigate how the system of equations, from which $\text{psm2regex}(P)$ is obtained, is solved. We observe that every equation is guarded, i.e. there is a letter from Σ before an occurrence of r_q for some state q. Solving the system of equations for the initial state r_{q_0} can only involve substitution and the application of Lemma 8.18. For both, sender-driven choice of P is preserved for the RE across all equations, yielding a sender-driven RE for r_{q_0}. For the third claim, it suffices to observe that no ε is introduced in the system of equations. ☐

Step (2): From Regular Expressions to Ancestor-recursive Non-merging Dense Intermediate-recursion-free PSMs

After the transformation, we want the PSM to be ancestor-recursive, non-merging, dense and intermediate-recursion-free by construction. We need to carefully design this transformation because the standard approach introduces non-determinism, for instance for union. Resolving this non-determinism would easily break the desired structural properties, making the whole workflow pointless. We apply the idea of derivatives in order not to introduce non-determinism. To preserve sender-driven choice, we also ensure that sender-driven regular expressions are closed under Brzozowski Derivatives. Given a regular expression r and a letter a, these allow to construct a regular expression that specifies the language of words in the semantics of r which start with a and omits a. We apply a similar idea to PSMs in order not to introduce non-determinism when constructing PSMs from regular expressions.

Definition 8.26 ([27]). Let Δ be an alphabet. We define the Brzozowski derivative $\text{brz-deriv}\colon \Delta \times \mathcal{R}_\Delta \to \mathcal{R}_\Delta$ as follows:

$$\text{brz-deriv}(a, r) := \begin{cases} \varepsilon & \text{if } r = a \\ \text{brz-deriv}(a, r_1) + \text{brz-deriv}(a, r_2) & \text{if } r = r_1 + r_2 \\ \text{brz-deriv}(a, r_1) \cdot r_2 & \text{if } r = r_1 \cdot r_2 \wedge \varepsilon \notin \mathcal{L}(r_1) \\ \text{brz-deriv}(a, r_1) \cdot r_2 + \text{brz-deriv}(a, r_2) & \text{if } r = r_1 \cdot r_2 \wedge \varepsilon \in \mathcal{L}(r_1) \\ \text{brz-deriv}(a, r_1) \cdot r_1^* & \text{if } r = r_1^* \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 8.27 (Correctness of Brzozowski Derivative [27]). Let r be a regular expression over an alphabet Δ and $a \in \Delta$ be a letter. If $\text{brz-deriv}(a, r)$ is defined, it holds that

$$\mathcal{L}_{\text{fin}}(\text{brz-deriv}(a, r)) = \{w \mid aw \in \mathcal{L}_{\text{fin}}(r)\} \ .$$

If $\text{brz-deriv}(a, r)$ is not defined, it holds that $\{w \mid aw \in \mathcal{L}_{\text{fin}}(r)\} = \emptyset$.

We extend this result to infinite words.

Lemma 8.28 (Brzozowski Derivative for Infinite Words). Let r be a regular expression over an alphabet Δ and $a \in \Delta$ be a letter. If $\text{brz-deriv}(a, r)$ is defined, it holds that

$$\mathcal{L}_{\text{inf}}(\text{brz-deriv}(a, r)) = \{w \mid aw \in \mathcal{L}_{\text{inf}}(r)\} \ .$$

If $\text{brz-deriv}(a, r)$ is not defined, it holds that $\{w \mid aw \in \mathcal{L}_{\text{inf}}(r)\} = \emptyset$.

Proof. For the first claim, we consider infinite words. By definition, an infinite word is in a language if all its prefixes are a prefix of some word in the finite language. Thus, it suffices to show that $\text{pref}(\mathcal{L}_{\text{fin}}(\text{brz-deriv}(a, r))) = \text{pref}(\{w \mid aw \in \mathcal{L}_{\text{inf}}(r)\} \cap \Delta^*)$. By definition, the prefixes of infinite words and finite words are the same for a regular expression. Thus, it remains to show that

$$\text{pref}(\mathcal{L}_{\text{fin}}(\text{brz-deriv}(a, r))) = \text{pref}(\{w \mid aw \in \mathcal{L}_{\text{fin}}(r)\} \cap \Delta^*) \ .$$

This follows from Lemma 8.27.

The second claim simply follows from Lemma 8.27 and the definition of $\mathcal{L}_{\text{inf}}(\text{-})$, which requires $\mathcal{L}_{\text{fin}}(\text{-})$ to be non-empty. □

From the correctness of Brzozowski derivatives, this observation follows directly.

Corollary 8.29. Let r be a regular expression over an alphabet Δ and $D \subseteq \Delta$ be the first letters in words in r, i.e. $D := \{a_1 \mid a_1 \ldots a_n \in \mathcal{L}_{\text{fin}}(r)\}$. Then, it holds that

$$\mathcal{L}_{\text{fin}}(r) = \biguplus_{a \in D} \{a \cdot w \mid w \in \mathcal{L}_{\text{fin}}(\text{brz-deriv}(a, r))\} \ .$$

Intuitively, we pull out every first letter for union and concatenation to avoid the introduction of ε-transitions. For this to work, we need to introduce a PSM derivative (function). If we used the Brzozowski Derivative, we could not apply structural induction to prove equivalence of the regular expression and the PSM. Still, we show that sender-driven choice is preserved by the Brzozowski Derivative.

Lemma 8.30. Let r be an sender-driven RE and $x \in \text{first}(\mathcal{L}(r))$. Then, it holds that $\text{brz-deriv}(x, r)$ is a sender-driven RE.

Proof. Brüggemann-Klein et al. [26] show that deterministic REs, which they call 1-unambiguous, are closed under the Brzozowski derivative. Their result generalises to sender-driven REs. They define star normal form for regular expressions [26, Def. 3.3]. They recall that deterministic REs can always be specified by an RE in star normal form [25]. With [26, Thm. B], they show that the Brzozowski derivative of a deterministic RE in star normal form is again deterministic and in star normal form. The conditions on sender-driven choice for REs do not restrict representability in star normal form and, thus, the result generalises to sender-driven REs. □

The last ingredient for our transformation is a procedure that applies the derivative to PSMs, preserving the properties of interest.

Lemma 8.31 (PSM for Derivative)**.** Let $P = (Q, \Gamma, \delta, q_0, F)$ be a PSM that is ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final. and let $a \in \text{first}(\mathcal{L}(P))$. Then, there is a PSM, denoted by $\text{psm-deriv}(a, P)$, such that $\mathcal{L}(\text{psm-deriv}(a, P')) = \text{brz-deriv}(a, \mathcal{L}(P))$, which is ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final. If P is sender-driven, $\text{psm-deriv}(a, P)$ is sender-driven.

Proof. Let q_0 be the initial state of P, Q_1 be the states with incoming transitions from q_0, and Q_2 be the states with outgoing transition to q_0. (Without loss of generality, we can assume that these are ε-transitions.) Formally $Q_1 := \{q_1 \mid (q_0, _, q_1) \in \delta\}$ and $Q_2 := \{q_2 \mid (q_2, \varepsilon, q_0) \in \delta\}$. By assumption that $a \in \text{first}(\mathcal{L}(P))$, we know that there is $q_1 \in Q_1$ such that $(q_0, a, q_1) \in \delta$. We construct $\text{psm-deriv}(a, P)$ as follows. We take q_1 as its initial state and do only keep the states for which q_1 is an ancestor. For every state q_2 in Q_2, which was not deleted, we copy the original PSM P, remove the state q_2 and replace it by the initial state from the copy. By assumption that the original PSM P is ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final, this construction yields a PSM $\text{psm-deriv}(a, P)$ with the same properties and $\mathcal{L}(\text{psm-deriv}(a, P)) = \text{brz-deriv}(a, \mathcal{L}(P))$. By construction, $\text{psm-deriv}(a, P)$ is sender-driven if P is. □

With this, we can provide the procedure that translates REs to PSMs.

Procedure 8.32 (RE to PSM)**.** Given a regular expression r without ε over Γ, we inductively construct the PSM $\text{regex2psm}(r)$:

1. a: one initial state with one transition labelled a to one final state;
2. $r_1 + r_2$: We add one initial state. For r_1, we compute $\text{psm-deriv}(a, \text{regex2psm}(r_1))$ for every first letter a and add a transition labelled with the letter from the initial state to the initial state of the PSM. We do the same for r_2.
3. $r_1 \cdot r_2$: We construct the automaton for r_1. For r_2, we apply the derivative idea again: we compute $\text{psm-deriv}(a, \text{regex2psm}(r_2))$ for every first letter a and copy each automaton as often as there are final states in the automaton for r_1 and add transitions from each such final state.
4. r^*: For Kleene Star, we construct the automaton for the inner regex and connect the final state(s) with the initial one by an ε-transition and make the initial one final. (These are backward transitions and, thus, should to be labelled ε for the PSM to be dense.)

We prove the previous procedure to be correct.

Lemma 8.33. Let r be a regular expression over Γ without ε and $\text{regex2psm}(r)$ be a PSM. Then, the core language of both are the same, i.e. $\mathcal{L}(r) = \mathcal{L}(\text{regex2psm}(r))$. It holds that $\text{regex2psm}(r)$ is ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final. If r is a sender-driven RE, then $\text{regex2psm}(r)$ is a sender-driven PSM.

Proof. We prove the claims by induction on the structure of the regular expression r.

The case for a single letter $r = a$ is obvious.

Let $r = r_1 \cdot r_2$. We first show that $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(\text{regex2psm}(r_1 \cdot r_2))$. The PSM construction applies the PSM derivative $\text{psm-deriv}(a, \text{regex2psm}(r_2))$ for every $a \in \text{first}(\mathcal{L}(\text{regex2psm}(r_2))$ and copies the resulting PSM for every final state of $\text{regex2psm}(r_1)$ and adds a transition with label a. Thus, for every word w in $\text{regex2psm}(r_1 \cdot r_2)$, we have that

- $w \in \mathcal{L}(\text{regex2psm}(r_1)) \cap \Gamma^\omega$ or
- $w = u \cdot a \cdot v$ with $u \in \mathcal{L}(\text{regex2psm}(r_1))$, $a \in \text{first}(\mathcal{L}(\text{regex2psm}(r_2)))$, and $v \in \mathcal{L}(\text{psm-deriv}(a, \text{regex2psm}(r_2)))$.

By induction hypothesis, we have that $\mathcal{L}(r_1) = \mathcal{L}(\text{regex2psm}(r_1))$ and $\mathcal{L}(r_2) = \mathcal{L}(\text{regex2psm}(r_2))$. By Lemma 8.31, $\mathcal{L}(\text{psm-deriv}(a, r_2)) = \mathcal{L}(\text{brz-deriv}(a, r_2))$. Hence, we obtain:

- $w \in \mathcal{L}(r_1) \cap \Gamma^\omega$ or
- $w = u \cdot a \cdot v$ with $u \in \mathcal{L}(r_1)$, $a \in \text{first}(\mathcal{L}(r_2))$, and $v \in \mathcal{L}(\text{brz-deriv}(a, r_2))$.

By the semantics of regular expressions and Lemmas 8.27 and 8.28, it follows that $w \in \mathcal{L}(r_1 \cdot r_2)$, which shows language equality.

By induction hypothesis, we know that $\text{regex2psm}(r_1)$ and $\text{regex2psm}(r_2)$ are ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final. By Lemma 8.31, for every $a \in \text{first}(\mathcal{L}(\text{regex2psm}(r_2)))$, $\text{psm-deriv}(a, \text{regex2psm}(r_2))$ is ancestor-recursive, non-merging, dense, intermediate-recursion-free, sink-final and sender-driven if $\text{regex2psm}(r_2)$ is. Thus, by construction, the PSM $\text{regex2psm}(r_1 \cdot r_2)$ is ancestor-recursive, non-merging, dense, intermediate-recursion-free, and sink-final, where the multiple copies ensure ancestor-recursiveness and non-merging property and the derivatives preserve density; and sender-driven choice is also preserved.

For $r_1 + r_2$, the construction applies the PSM derivative to avoid introducing non-determinism (as is common in standard constructions for FSMs from REs) if it was not present before. If it was there before, it preserves it to avoid subsequent merging and, thus, avoids introducing non-sink final states. For sender-driven choice, the assumption for the regular expression yields that the first letters are pair-wise distinct and, thus, the newly introduced branching satisfies the sender-driven choice condition for PSMs. The remaining reasoning is very similar to the previous case for concatenation and, hence, omitted. Here, both r_1 and r_2 are treated the same, like the second part of concatenation.

For r_1^*, we simply introduce a backward transition, which ought to be labelled by ε and it is. In fact, these are the only ε-transition, ensuring that the PSM is dense. Note that we construct a PSM without forward transitions that are labelled with ε. This is different from state machines for global types where every subterm of shape $\mu t \,.\, G$ has only one incoming backward and one outgoing forward transition labelled by ε. In this construction, we basically merge the states for $\mu t \,.\, G$ and G. It is also the place where recursion is introduced, ensuring ancestor-recursion and intermediate recursion freedom. For sender-driven choice, analogously, the assumption for the regular expression yields that the first letters are pair-wise distinct and, thus, the branch that decides whether to start or repeat with r_1 or continue with the next regular expression satisfies the sender-driven choice condition for the PSMs. ☐

Step (3): From Ancestor-recursive Non-merging Dense Intermediate-recursion-free PSMs to Global Types

While the previous steps apply to arbitrary (sink-final) PSMs, this one only applies for Σ1-PSMs since global types specify send and receive events together. This transformation is rather straightforward. The global type can be constructed via a traversal of the Σ1-PSM.

Procedure 8.34 (Σ1-PSM to Global Type)**.** Let P be ancestor-recursive, non-merging, dense, and intermediate-recursion-free Σ1-PSM. As a preprocessing step, we merge asynchronous events and assume P works on the alphabet of synchronous events $\Sigma_{\mathcal{P}}$. We start with an empty global type and start the traversal from initial state:

- If the state is final: add 0 and return;
- if the state has an incoming transition:
  add $\mu t.$ for fresh t and store t for this state;
- if the state has an outgoing transition to previously seen state:
  add t for the destination of the outgoing transition and return;
- if the state has outgoing transitions to unseen states:
  add $\Sigma_{i \in I}$ with a fresh index set I (for $|I|$ branches) with one branch for each next state with according transition label and recurse for each of next states.

Note that a state can have an incoming transition and more than one outgoing transitions, in contrast to the state machine of a global type where every subterm of shape $\mu t \,.\, G$ has only one incoming and one outgoing transition labelled by ε. In this construction, the states for $\mu t \,.\, G$ and G are merged. We denote the result of this procedure with $\mathrm{psm2gt}(P)$.

The following lemma states the correctness of the previous procedure.

Lemma 8.35. Let P be ancestor-recursive non-merging dense intermediate-recursion-free sink-final $\Sigma 1$-PSM and $\mathrm{psm2gt}(P)$ be the global type constructed from P. Then, their core languages are the same: $\mathcal{L}(P) = \mathcal{L}(\mathrm{psm2gt}(P))$. If P is a sender-driven PSM, then $\mathrm{psm2gt}(P)$ is a sender-driven global type.

Proof. The assumptions guarantee that the traversal does not revisit states and only sink states are final. The preprocessing simplifies the translation to the corresponding terms of a global type. The claim then follows easily by construction. We sketch how to formalise it. One can define a formalism that jointly/recursively represents languages starting from states in an FSM and (partial) global types. The construction iteratively refines this representation, preserving the specified language and sender-driven choice if given. □

Wrapping Up: From PSMs to Global Types

Let us first observe that part of this workflow can be applied when using the more general alphabet Γ where send and receive events may not happen next to each other.

Lemma 8.36. For every PSM P with $\varepsilon \notin \mathcal{S}(P)$, there is an ancestor-recursive, non-merging, dense, and intermediate-recursion-free PSM P' with the same core language. If P is sink-final and satisfies mixed choice (sender-driven choice, or directed choice respectively), then P' is sink-final and satisfies mixed choice (sender-driven choice, or directed choice respectively). If P is not sink-final, restrictions on choice are not preserved.

Proof. Let P be a sink-final PSM with $\varepsilon \notin \mathcal{S}(P)$. We do a case analysis if P is sink-final. If P is sink-final,

$$\mathrm{regex2psm}(\mathrm{psm2regex}(P))$$

is such a PSM and any restriction on choice is preserved by Lemmas 8.25 and 8.33. If P is not sink-final,

$$\mathrm{regex2psm}(\mathrm{psm2regex}(\text{psm2sink-final-psm}(P)))$$

is such a PSM by Proposition 8.16 and Lemmas 8.25 and 8.33. □

For the special case of $\Sigma 1$-PSMs, we can convert such PSMs to global types, proving the main result in this section.

Theorem 8.14. For every sink-final $\Sigma 1$-PSM P, there is a global type G with the same core language. If P is non-deterministic (mixed-choice, sender-driven, or directed respectively), G is non-deterministic (mixed-choice, sender-driven, or directed respectively). Every $\Sigma 1$-PSM P with $\varepsilon \notin \mathcal{S}(P)$ can be represented as a non-deterministic global type with the same core language.

Proof. For the first claim, let P be a sink-final Σ1-PSM. We do a case analysis if $\varepsilon \in \mathcal{S}(P)$.

If so, we know that $\mathcal{S}(P) = \{\varepsilon\}$ because P is sink-final and no transition is labelled by ε. Then, $G = 0$ has the same core language.

If not, we can apply our workflow and construct the global type

$$\text{psm2gt}(\text{regex2psm}(\text{psm2regex}(P)))$$

which represents the same core language and any restriction on choice is preserved by Lemmas 8.25, 8.33 and 8.35.

For the second claim, let P be a non-sink-final Σ1-PSM with $\varepsilon \notin \mathcal{S}(P)$. Then,

$$\text{psm2gt}(\text{regex2psm}(\text{psm2regex}(\text{psm2sink-final-psm}(P))))$$

is a non-deterministic global type that represents the same core language by Proposition 8.16 and Lemmas 8.25, 8.33 and 8.35. □

More General Applicability of This Transformation. Our constructions are not restricted to PSMs but can be applied to FSMs over other alphabets. For such FSMs, the reasoning about preserving sender-driven choice often translates to preserving determinism. Thus, it shows that the above structural conditions do not change expressivity for sink-final deterministic FSMs. For instance, the previous workflow can also be applied to a sink-final FSM over Γ_{p} for a participant p. If the FSM has no mixed-choice states, it will produce a local type. If it does, the structure will still resemble the one of local types but requires to specify receiving and sending at the same time.

Lemma 8.37. Let p be a participant and A_{p} be a sink-final state machine over Γ_{p} without mixed-choice states. Then, one can construct a local type L_{p} for p such that $\mathcal{L}(L_{\mathsf{p}}) = \mathcal{L}(A_{\mathsf{p}})$.

This can be used to turn any implementation where each state machine is sink-final and has no mixed-choice states into a collection of local types that implements it.

Corollary 8.38. Let $L \subseteq \Gamma^\infty$ be a language. If L is implementable by a CSM $\{\!\{A_{\mathsf{p}}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ where every A_{p} is sink-final and has no mixed-choice states for every $\mathsf{p} \in \mathcal{P}$, then there is a local type L_{p} for every $\mathsf{p} \in \mathcal{P}$ such that $\{\!\{\mathsf{LAut}(L_{\mathsf{p}})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ implements L.

With the completeness result for our subset projection operator for global types (Theorem 5.21), we showed that mixed-choice states are not necessary to implement (sender-driven) global types. With regard to the restriction to be sink-final, we showed that (sender-driven) global types are softly implementable if and only if the subset projection is sink-final for every participant (Theorem 5.23). Thus, for a solution to the soft implementatability problem, we can always construct a collection of local types with the same behaviours when interpreted as CSM.

8.2.3 From HMSCs to PSMs

We can compute a PSM from an HMSC. We will show how to preserve restrictions on choice if given. For this, let us first define choice for HMSCs.

Definition 8.39 (Sender-driven choice for HMSCs)**.** Let $H = (V, E, v^I, V^T, \mu)$ be an HMSC. We say H is *sender-driven* if there is a choice function $\lambda\colon V \to \mathcal{P}$ such that $\lambda(v)$ has a minimal event e_u in every $\mu(u)$ for u with $(v, u) \in E$ and the labels of these minimal events are pair-wise distinct, i.e. $\forall (v, u_1), (v, u_2) \in E,\ u_1 \neq u_2 \implies l_1(e_{u_1}) \neq l_2(e_{u_2})$ where $\mu(u_i) = (_, _, _, l_i)$. We say that H is *directed* if the receiver for each e_v is the same across all branches.

Note that we gave an HMSC specification of the Kindergarten Leader Election protocol Fig. 8.1 but it is not sender-driven and cannot be specified as such. In contrast, the given PSM is even directed.

We now define our procedure to turn an HMSC into a PSM, preserving sender-driven choice if given.

Procedure 8.40 (From HMSCs to PSMs)**.** Let $H = (V, E, v^I, V^T, \mu, \lambda)$ be a sender-driven HMSC. We construct a PSM $\mathrm{hmsc2psm}(H) := (Q, \Gamma, \delta, q_0, F)$ as follows:

- For every $u \in V$, we compute the set of minimal event $E^u_{\min}$ of $\mu(u)$.
- For every $u \in V$ and $e \in E^u_{\min}$ where $\mathtt{p}$ is the sender of the label of e, we compute a linearisation of the event labels w of $\mu(v)$ such that the label of e is the first in w. For w, we construct a PSM $P_{u,\mathtt{p}}$ which does not branch.
- For q_0: we use the initial state of $P_{v^I,\mathtt{p}}$ for some $\mathtt{p}$ which has a minimal event in $\mu(v^I)$.
- For δ: for every $(v, u) \in E$, we (temporarily) add an ε-transition between the final state of every (defined) $P_{v,\mathtt{q}}$ for every $\mathtt{q}$ (except for the initial vertex where we only use $P_{v^I,\mathtt{p}}$ from before), and the initial state of $P_{u,\lambda(v)}$ and can simply remove it by removing the initial state of $P_{u,\lambda(v)}$.
- For Q: we simply union all states and omit the initial states we removed for the temporary ε-transitions (and the PSMs for the initial vertex v^I that we did not use).
- For F: these are the final states of all PSMs $P_{v,_}$ for which $v \in V^T$.

If there is no need to preserve sender-driven or directed choice, we do not need to copy the PSMs for the different minimal events, which simplifies the above procedure.

Lemma 8.41. Let H be an HMSC. Then, its PSM encoding $\mathrm{hmsc2psm}(H)$ specifies the same semantics, i.e. $\mathcal{L}(H) = \mathcal{S}(\mathrm{hmsc2psm}(H))$. If H is a sender-driven (resp. directed) HMSC, then $\mathrm{hmsc2psm}(H)$ is a sender-driven (resp. directed) PSM.

Proof. Let $H = (V, E, v^I, V^T, \mu)$ be the HMSC. By construction, it is easy to see that the PSM $P_{v,_}$ for every individual BMSC $\mu(v)$ is constructed such that it specifies one of its linearisations. The semantics of HMSCs is closed under the indistinguishability relation $\sim$ (Lemma 2.21) and, hence, is the semantics of BMSCs. The same holds for the semantics of PSMs by definition. It is also easy to see that the edges of H are mimicked correctly. It remains to argue that sender-driven (resp. directed) choice is preserved if H is a sender-driven (resp. directed) HMSC, i.e. with choice function λ. This is the case as we copy the individual BMSCs for every minimal event and use the choice function λ of H to determine how to connect the (linear) PSMs. By removing the initial states when joining two PSMs $P_{v,_}$ and $P_{u,\lambda(v)}$ for $(v, u) \in E$, we do not introduce ε-transitions – while introducing them would not yield a PSM. □

Chapter 9

Checking Implementability with Mixed Choice is Undecidable

In this section, we show that checking implementability for sink-final mixed-choice $\Sigma 1$-PSMs is undecidable in general. Together with our workflow to turn such PSMs into global types, this result carries over to mixed-choice global types.

Lohrey proved that implementability of HMSCs is undecidable in general [91, Thm. 3.4]. This proof entails undecidability of $\Sigma 1$-PSMs. However, we showed decidability of the implementability problem for global types. Thus, the proof for undecidability for PSM implementability ought to break. There are two possible reasons: first, the fact that PSMs employ fewer restrictions on protocol specifications, or, second, mixed choice. With our expressivity results, the first reason does not apply for sink-final PSMs and we will show that the undecidability result persists for sink-final PSMs. Thus, mixed choice must be the reason. However, the original proof does not give any insights where and how this happens. The underlying idea of the proof was apparently common back then but, without this context, the proof is rather difficult to follow. In addition, it is, in fact, a combination of two proofs: Lohrey initially proves EXPSPACE-hardness for implementability of bounded HMSCs and, then, part of this proof is then used to prove undecidability. We decided to transcribe the proof in our terminology, emphasising the concept of choice and making the encoding sink-final.

Prior to giving the statement and the full proof, let us give a sketch of the main ideas. We reduce the acceptance problem for Turing Machines to checking implementability sink-final mixed-choice $\Sigma 1$-PSMs. We have five participants interact with each other: $\mathrm{p}_1, \ldots \mathrm{p}_5$. The two pairs p_1 and p_2 as well as p_4 and p_5 send possibly ill-formed Turing Machine configurations $(u_1, \ldots, u_m)$ for any $m \geq 1$ as messages to each other. We define two languages L_l and L_r, which are the same if and only if the Turing Machine has no accepting computation for the given word. Initially, p_3 decides which branch to choose by sending a message to p_2 and, thus, whether to follow L_l or L_r in the rest of the protocol. The remaining participants $\mathrm{p}_1, \mathrm{p}_4$ and p_5 never learn about this choice. However, we show that L_l is implementable. Overall, the PSM is implementable if and only if there is no accepting computation for the given word.

For L_l, we simply accept any two sequences of the same length (one for each pair) where the sequences do not need to but can represent valid Turing Machine computations. It is fairly straightforward to define such a PSM.

For L_r, we only accept two sequences of Turing Machine configurations of the same length (one for each pair) where sequences do not represent accepting Turing Machine computations. Constructing a PSM for L_r is more involved. We use a characterisation of how Turing Machine computations can go wrong: these are five conditions and we show how to specify PSMs for each of these languages. The language L_r is then the union of all of these. Interestingly, each individual PSM does not expose mixed choice but when combining them to obtain one PSM for L_r, the resulting PSM necessarily exposes mixed choice. Recall that the pairs $(\mathtt{p}_1, \mathtt{p}_2)$ and $(\mathtt{p}_4, \mathtt{p}_5)$ communicate Turing Machine configurations $C_1, \ldots, C_m$ and $D_1, \ldots, D_m$ for some m. For one condition, we check that $C_i \neq D_i$ for some $1 \leq i \leq m$, making the Turing Machine computation invalid. For another one, we check that C_{i+1} is no successor of D_i for some $1 \leq i \leq m$, again making the encoded Turing Machine computation invalid. For the first condition, the communication for C_i and D_i ought to be possible in parallel while, for the second condition, the communication for C_{i+1} and D_i ought to be possible in parallel. Merging the respective PSMs necessarily exposes mixed choice to allow for both and this is where imposing a restriction on choice would break the undecidability proof.

Theorem 9.1. The implementability problem for sink-final mixed-choice Σ1-PSMs is undecidable in general.

Proof. We consider the problem of checking if a word is accepted by a Turing Machine. This, as a variant of the halting problem [126], is known to be undecidable. We reduce it to checking implementability of a mixed-choice sink-final Σ1-PSM. We basically construct a PSM with two branches in the very beginning. For each, we construct a language and they coincide – which will be necessary for implementability – if and only if the Turing Machine does not halt in a final configuration. We assume familiarity with the concept of Turing Machines and refer to [69] for further details.

Let TM be a Turing Machine with tape alphabet Δ and states Q with $\Delta \cap Q = \emptyset$. We have that $q_0 \in Q$ is the initial state and $q_f \in Q$ is, without loss of generality, the only final state. A configuration of TM is given by a word $a_1, \ldots, a_i, q, b_1, \ldots, b_j \in \Delta^* Q \Delta^*$. The initial configuration for input word w is $q_0 w$ while any configuration from $\Delta^*\{q_f\}\Delta^*$ is final. A computation is a sequence of configurations $(u_1, \ldots, u_m)$ such that u_{i+1} is the next configuration of TM, also denoted by $u_i \vdash_{TM} u_{i+1}$. A computation $(u_1, \ldots, u_m)$ accepts w if $u_1 = q_0 w$ and $u_m \in \Delta^*\{q_f\}\Delta^*$.

For our encoding, we use five participants $\mathtt{p}_1, \ldots, \mathtt{p}_5$ who send configurations to each other. Therefore, messages are from the set $\Delta \uplus \{\circ, \langle\!\langle, \rangle\!\rangle, \bot\} \uplus Q$ where $\circ$ is sent by $\mathtt{p}_3$ to indicate the start of a new pair of configurations and $\langle\!\langle$ and $\rangle\!\rangle$ delimit a configuration.

Let us introduce some notation: $p \leftrightarrow q : m$ abbreviates $p \rightarrow q : m \cdot q \rightarrow p : m$. We only specify interactions using $_ \leftrightarrow _ : _$. Using these, we will also define regular expressions and complements thereof and consider $p \leftrightarrow q : m$ as their single letters. By construction, every PSM will be $\exists\Sigma 1$-bounded and, in fact, every message is immediately acknowledged.

For a word $w = w_1 \ldots w_i$, we write $p \leftrightarrow q : w$ for $p \leftrightarrow q : w_1 \cdots p \leftrightarrow q : w_i$.

For words $C_1, D_1, C_2, D_2 \ldots, C_m, D_m \in (\Delta \uplus Q)^*$, we define the word

$$
\begin{aligned}
w(C_1, D_1, C_2, D_2, \ldots, C_m, D_m) := \quad & p_3 \leftrightarrow p_2 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot p_2 \leftrightarrow p_1 : C_1 \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot \\
& p_3 \leftrightarrow p_4 : \circ \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot p_4 \leftrightarrow p_5 : D_1 \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \cdot \\
& p_3 \leftrightarrow p_2 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot p_2 \leftrightarrow p_1 : C_2 \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot \\
& p_3 \leftrightarrow p_4 : \circ \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot p_4 \leftrightarrow p_5 : D_2 \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \cdot \\
& \ldots \\
& p_3 \leftrightarrow p_2 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot p_2 \leftrightarrow p_1 : C_m \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot \\
& p_3 \leftrightarrow p_4 : \circ \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot p_4 \leftrightarrow p_5 : D_m \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \,.
\end{aligned}
$$

Intuitively, p_2 sends the sequence C_i to p_1 while p_4 sends the sequence D_i to p_5. Each sequence is started by a $\langle\!\langle$-message and finished by a $\rangle\!\rangle$-message between the respective pair. The participant p_3 starts each round by sending $\circ$. We give an illustration of the MSC $\text{msc}(w(C_1, D_1, C_2, D_2, \ldots, C_m, D_m))$ in Fig. 9.1 on p. 206. We will use the closure of $w(\text{-}, \ldots, \text{-})$ close under $\sim$, justifying the representation as BMSC.

We define two languages L_l and L_r, which we later use for two branches of a PSM.

$$
\begin{aligned}
L_l &:= \{\mathcal{C}^{\sim}(w(C_1, D_1, \ldots, C_m, D_m)) \mid m \geq 1, C_1, D_1, \ldots, C_m, D_m \in (\Delta \uplus Q)^*\} \\
L_r &:= L_l \setminus \{\mathcal{C}^{\sim}(w(u_1, u_1, \ldots, u_m, u_m)) \mid (u_1, \ldots, u_m) \text{ is an accepting computation}\}
\end{aligned}
$$

Note that the communication of C_i between p_1 and p_2 can happen concurrently to both D_{i-1} and D_i between p_4 and p_5. (This will later allow us to both detect if C_i and D_i do not coincide or C_i is no successor configuration of D_i.)

To make the resulting PSM sink-final, we define a sequence of messages that indicates the end of an execution

$$
w_{end} := p_3 \leftrightarrow p_2 : \bot \cdot p_2 \leftrightarrow p_1 : \bot \cdot p_3 \leftrightarrow p_4 : \bot \cdot p_4 \leftrightarrow p_5 : \bot
$$

and append it to obtain $L'_l := \{w \cdot w_{end} \mid w \in L_l\}$ and $L'_r := \{w \cdot w_{end} \mid w \in L_r\}$.

We will show that both L_l and L_r, and thus, L'_l and L'_r, can be specified as $\Sigma 1$-PSMs. Provided with PSMs for L_l and L_r, it is straightforward to construct a PSM P_{TM} with

$$
\mathcal{S}(P_{TM}) = \{p_2 \rightarrow p_3 : \mathsf{l} \cdot w \mid w \in L'_l\} \uplus \{p_2 \rightarrow p_3 : \mathsf{r} \cdot w \mid w \in L'_r\} \,.
$$

By definition of L'_l and L'_r, every word ends with w_{end} so P_{TM} is sink-final. (In fact, if there are FSMs for all participants, they will also be sink-final.) We will show that P_{TM} is implementable if and only if TM does not accept the input w.

For this, it suffices to establish the following four facts:

- Claim 1: L_l and L_r can be specified as $\Sigma 1$-PSMs.
- Claim 2: L_l is implementable.
- Claim 3: If TM has no accepting computation for w, then P_{TM} is implementable.
- Claim 4: If TM has an accepting computation for w, then P_{TM} is not implementable.

Claim 1: Both L_l and L_r can be specified as $\Sigma 1$-PSMs.

Proof of Claim 1. It is easy to construct a PSM from a regular expression. Thus, for conciseness, we may give regular expressions for the languages we consider or for their complements. For this, we introduce some more notation for concise specifications when using sets of messages:

$$\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \{x_1, \ldots, x_n\} := (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : x_1 + \ldots + \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : x_n) \ .$$

First, let us consider L_l. Inspired by the definition of $w(C_1, D_1, \ldots, C_m, D_m)$, we construct this regular expression r_l for L_l:

$$\begin{aligned}(&\mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : (\Delta \uplus Q))^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\ &\mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : (\Delta \uplus Q))^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle)^* \ .\end{aligned}$$

Second, let us consider L_r. Recall that L_r should admit the encoding of all sequences of configurations except for accepting ones. We provide an exhaustive list of how such a sequence can fail to be an accepting computation. We provide a language $L_{r,i}$ for each and L_r is their union.[1]

- $L_{r,1}$ contains all sequences of configurations for which some C_k or D_k is actually not a configuration, i.e. not from $\Delta^* Q \Delta^*$.
- $L_{r,2}$ contains all sequences for which C_1 is not the correct initial configuration, i.e. it does not have the shape $q_0, a_1, a_2, \ldots a_n$ where $w = a_1 \cdots a_n$.
- $L_{r,3}$ contains all sequences for which q_f does not occur in C_m.
- $L_{r,4}$ contains all sequences where C_k and D_k differ in some position.
- $L_{r,5}$ contains all sequences for which C_{k+1} is no successor configuration for D_k.

For each $L_{r,i}$, we show that it can be specified as PSM (or regular expression). It is straightforward to obtain a PSM for L_r by adding one initial state and adding a transition from this one to the initial state for the PSM of $L_{r,i}$ for each i.

[1]Some of Lohrey's construction for EXPSPACE-hardness does not apply to the undecidability proof so we renumbered the languages because some of his construction.

Language $L_{r,1}$:

We construct a regular expression for $w(C_1, D_1, \ldots, C_m, D_m)$ for any m such that there is some C_i or D_i with either no message from Q or at least two messages from Q:

$$
\begin{aligned}
r_1 := \quad & (\mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot \\
& \qquad (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : Q \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\
& \mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot \\
& \qquad (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : Q \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle)^* \cdot \\
& (r_{l0} + r_{l2} + r_{r0} + r_{r2}) \cdot r_l^*
\end{aligned}
$$

$$
\begin{aligned}
\text{where} \qquad r_{l0} := \; & \mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\
& \mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta \uplus Q)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle \\
r_{l2} := \; & \mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot \\
& \qquad (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : Q \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta)^* \cdot \\
& \qquad \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : Q \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta \uplus Q)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\
& \mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta \uplus Q)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle \\
r_{r0} := \; & \mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta \uplus Q)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\
& \mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle \\
r_{r2} := \; & \mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle \cdot (\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta \uplus Q)^* \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \rangle\!\rangle \cdot \\
& \mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \langle\!\langle \cdot \\
& \qquad (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : Q \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta)^* \cdot \\
& \qquad \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : Q \cdot (\mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \Delta \uplus Q)^* \cdot \mathtt{p}_4 \leftrightarrow \mathtt{p}_5 : \rangle\!\rangle \ .
\end{aligned}
$$

Let us explain how r_1 works. In the beginning, there are only C_i and D_i with one message from Q. At some point, one of the regular expressions r_{l0}, r_{l2}, r_{r0} or r_{r2} has to match. These specify some way how the number of messages for Q can be wrong: r_{l0} has no messages from Q between $\mathtt{p}_2$ and $\mathtt{p}_1$ while r_{r0} has no messages from Q between $\mathtt{p}_4$ and $\mathtt{p}_5$; r_{l2} has more than one message from Q between $\mathtt{p}_2$ and $\mathtt{p}_1$ while r_{r2} has more than one message from Q between $\mathtt{p}_4$ and $\mathtt{p}_5$. For each, the other pair can communicate any number of messages from Q to account for sequences where both C_i and D_i do not match. Subsequently, we use r_l to simply allow any configurations.

Language $L_{r,2}$:

Let $w = a_1 \ldots a_n$ be the input word for TM. Then, we can specify $L_{r,2}$ as follows:

$$\{\mathcal{C}^{\sim}(w(C_1, D_1, C_2, D_2 \ldots, C_m, D_m)) \mid m \geq 1 \wedge C_1 \neq q_0 a_1 \ldots a_n\}$$

It is easy to see that we can change r_l to obtain a regular expression for

$$\{w(C_2, D_2, \ldots, C_m, D_m) \mid m \geq 1\} \ .$$

Thus, it suffices to show that $w(C_1, D_1)$ with $C_1 \neq q_0, a_1, \ldots, a_n$ can be specified as PSM. Again, we give a regular expression for the complement. We observe that the

communication between p_2 and p_1 about the configuration can easily be specified as regular expression:

$$p_2 \leftrightarrow p_1 : q_0 \cdot p_2 \leftrightarrow p_1 : a_1 \ldots p_2 \leftrightarrow p_1 : a_n \ .$$

It is straightforward to construct a PSM for the complement of this regular expression. (Before we did not use the complement for $L_{r,1}$ because we could not guarantee that the same number of configurations would be communicated between both pairs.) When combined, this gives us the following regular expression (with the complement operator as syntactic sugar) for $L_{r,2}$:

$$\begin{aligned}
&p_3 \leftrightarrow p_2 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot \\
&\overline{p_2 \leftrightarrow p_1 : q_0 \cdot p_2 \leftrightarrow p_1 : a_1 \cdot \ldots \cdot p_2 \leftrightarrow p_1 : a_n} \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot \\
&p_3 \leftrightarrow p_4 : \circ \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot (p_4 \leftrightarrow p_5 : (\Delta \uplus Q))^* \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \cdot r_l
\end{aligned}$$

Language $L_{r,3}$:

The following regular expression specifies all sequences for which the last C_m does not contain the final state $q_f \in Q$:

$$\begin{aligned}
&r_l \cdot \\
&p_3 \leftrightarrow p_2 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot (p_2 \leftrightarrow p_1 : (\Delta \uplus Q \setminus \{q_f\}))^* \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot \\
&p_3 \leftrightarrow p_4 : \circ \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot (p_4 \leftrightarrow p_5 : (\Delta \uplus Q))^* \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle
\end{aligned}$$

Language $L_{r,4}$:

Intuitively, we can merge the loops for both C_i and D_i to check that some message at the same position is different. This is possible because all languages are closed under $\sim$ by definition. We introduce this notation

$$\{p_2 \leftrightarrow p_1 : x \cdot p_4 \leftrightarrow p_5 : x \mid x \in \{y_1, \ldots, y_n\}\}$$

which is an abbreviation for

$$(p_2 \leftrightarrow p_1 : y_1 \cdot p_4 \leftrightarrow p_5 : y_1) + \ldots + (p_2 \leftrightarrow p_1 : y_n \cdot p_4 \leftrightarrow p_5 : y_n) \ .$$

With this, the following is a regular expression for $L_{r,4}$:

$$\begin{aligned}
r_4 := \quad & r_l \cdot \\
& p_3 \leftrightarrow p_2 : \circ \cdot p_3 \leftrightarrow p_4 : \circ \cdot p_2 \leftrightarrow p_1 : \langle\!\langle \cdot p_4 \leftrightarrow p_5 : \langle\!\langle \cdot \\
& (\{p_2 \leftrightarrow p_1 : x_1 \cdot p_4 \leftrightarrow p_5 : x_1 \mid x_1 \in (\Delta \cup Q)\})^* \cdot \\
& (r_a + r_b + r_c) \cdot r_l
\end{aligned}$$

where
$$\begin{aligned}
r_a := \quad & (\{p_2 \leftrightarrow p_1 : x_2 \cdot p_4 \leftrightarrow p_5 : x_3 \mid x_2, x_3 \in (\Delta \cup Q) \text{ and } x_2 \neq x_3\}) \cdot \\
& (\{p_2 \leftrightarrow p_1 : x_4 \cdot p_4 \leftrightarrow p_5 : x_5 \mid x_4, x_5 \in (\Delta \cup Q)\})^* \cdot r_l \\
r_b := \quad & \{p_2 \leftrightarrow p_1 : x_7 \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \mid x_7 \in (\Delta \cup Q)\} \cdot (p_2 \leftrightarrow p_1 : \Delta \cup Q)^* \cdot p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot r_l \\
r_c := \quad & \{p_2 \leftrightarrow p_1 : \rangle\!\rangle \cdot p_4 \leftrightarrow p_5 : x_6 \mid x_6 \in (\Delta \cup Q)\} \cdot (p_4 \leftrightarrow p_5 : \Delta \cup Q)^* \cdot p_4 \leftrightarrow p_5 : \rangle\!\rangle \cdot r_l \ .
\end{aligned}$$

The regular expression checks that at some point two configurations C_i and D_i do not agree on some position (using r_a), or C_i is longer than D_i (using r_b), or D_i is longer than C_i (using r_c).

Language $L_{r,5}$:

We use the same idea of merging the loops to compare as for the previous case and also use the same notation. We can give a regular expression that consists of different phases. First, we let p_2 and p_1 communicate about C_1 in order to then compare D_i with C_{i+1} for any i in a loop. We want that C_{i+1} is no successor of D_i for some i. Thus, we check if the changes from D_i to C_{i+1} are a valid transition for TM. The regular expression r_5 is defined as follows:

$$\begin{aligned} r_5 := \quad & \mathrm{p}_3 \leftrightarrow \mathrm{p}_2 : \circ \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \langle\!\langle \cdot (\mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : (\Delta \uplus Q))^* \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \rangle\!\rangle \cdot \\ & (r_d + r_e)^* \cdot (r_f + r_g) \cdot \\ & \mathrm{p}_3 \leftrightarrow \mathrm{p}_4 : \circ \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \langle\!\langle \cdot (\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : (\Delta \uplus Q))^* \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \rangle\!\rangle \cdot \\ & r_l \end{aligned}$$

where

$$\begin{aligned} r_d := \quad & \mathrm{p}_3 \leftrightarrow \mathrm{p}_4 : \circ \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \langle\!\langle \cdot \mathrm{p}_3 \leftrightarrow \mathrm{p}_2 : \circ \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \langle\!\langle \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : x_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : x_1 \mid x_1 \in (\Delta \cup Q)\})^* \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : a_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : a_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : b_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : b_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : c_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : c_2 \\ & \quad \mid a_1, a_2, b_1, b_2, c_1, c_2 \in (\Delta \cup Q)^*, a_1 \neq a_2 \text{ and} \\ & \quad\; \exists w_1, w_2 \in \Delta^*.\, w_1 a_1 b_1 c_1 w_2 \vdash_{TM} w_1 a_2 b_2 c_2 w_2\}) \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : x_2 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : x_2 \mid x_2 \in (\Delta \cup Q)\})^* \cdot \\ & \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \rangle\!\rangle \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \rangle\!\rangle \\ r_e := \quad & \mathrm{p}_3 \leftrightarrow \mathrm{p}_4 : \circ \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \langle\!\langle \cdot \mathrm{p}_3 \leftrightarrow \mathrm{p}_2 : \circ \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \langle\!\langle \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : x_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : x_1 \mid x_1 \in (\Delta \cup Q)\})^* \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : a_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : a_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : b_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : b_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \rangle\!\rangle \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : c_2 \cdot \\ & \; \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \rangle\!\rangle \\ & \quad \mid a_1, a_2, b_1, b_2, c_2 \in (\Delta \cup Q)^*, a_1 \neq a_2 \text{ and } \exists w_1 \in \Delta^*.\, w_1 a_1 b_1 \vdash_{TM} w_1 a_2 b_2 c_2\}) \\ r_f := \quad & \mathrm{p}_3 \leftrightarrow \mathrm{p}_4 : \circ \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \langle\!\langle \cdot \mathrm{p}_3 \leftrightarrow \mathrm{p}_2 : \circ \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \langle\!\langle \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : x_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : x_1 \mid x_1 \in (\Delta \cup Q)\})^* \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : a_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : a_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : b_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : b_2 \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : c_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : c_2 \\ & \quad \mid a_1, a_2, b_1, b_2, c_1, c_2 \in (\Delta \cup Q)^*, a_1 \neq a_2 \text{ and} \\ & \quad\; \nexists w_1, w_2 \in \Delta^*.\, w_1 a_1 b_1 c_1 w_2 \vdash_{TM} w_1 a_2 b_2 c_2 w_2\}) \cdot \\ & (\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \Delta \cup Q)^* \cdot (\mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \Delta \cup Q)^* \cdot \\ & \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \rangle\!\rangle \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \rangle\!\rangle \\ r_g := \quad & \mathrm{p}_3 \leftrightarrow \mathrm{p}_4 : \circ \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \langle\!\langle \cdot \mathrm{p}_3 \leftrightarrow \mathrm{p}_2 : \circ \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \langle\!\langle \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : x_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : x_1 \mid x_1 \in (\Delta \cup Q))^* \cdot \\ & (\{\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : c_1 \cdot \mathrm{p}_2 \leftrightarrow \mathrm{p}_1 : \rangle\!\rangle \mid c_1 \in (\Delta \cup Q)^*\}) \cdot \\ & (\mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \Delta \cup Q)^* \cdot \mathrm{p}_4 \leftrightarrow \mathrm{p}_5 : \rangle\!\rangle \end{aligned}$$

We distinguish two types of transitions: the ones that simply change letters in the middle of the configurations (achieved with r_d) and the ones that extend the tape (achieved with r_e). If one is matched against, we recurse using $(r_d + r_e)^*$. If not, $(r_f + r_g)$ is matched against and we, subsequently, allow any possible subsequent pair configurations using r_l. The regular expression r_f checks that the transition is not possible while r_g checks if C_{i+1} is shorter than D_i. Without loss of generality, we can assume that the tape never shrinks (as it could be encoded using an extra tape alphabet letter). Note that the transition check is a local condition and it suffices to check at most two more messages after the first different message. In fact, the words w_1 and w_2 in the conditions do not matter: either it is a transition for all such pairs or none. After the mismatch, we let D_i catch up and continue with r_l.

Mixed choice:

We explained how to construct a PSM for $L_{r,i}$ for every i. These are Σ1-PSMs by construction. In fact, each of these also satisfies sender-driven choice. However, when we combine both PSMs for $L_{r,4}$ and $L_{r,5}$ in order to obtain a PSM for L_r, the resulting PSM exposes mixed choice. Intuitively, this happens because $L_{r,4}$ checks C_i against D_i and $L_{r,5}$ checks D_i against C_{i+1}. Technically, when merging both PSMs, we reach a state after the sequence

$$\mathtt{p}_3 \leftrightarrow \mathtt{p}_2 : \circ \cdot \mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \langle\!\langle$$

for which $L_{r,4}$ requires to have $\mathtt{p}_3 \leftrightarrow \mathtt{p}_4 : \circ$ next while $L_{r,5}$ requires to have a loop with

$$\mathtt{p}_2 \leftrightarrow \mathtt{p}_1 : \Delta \uplus Q \ .$$

It is not possible to let $\mathtt{p}_2$ send a message to distinguish both branches as the indistinguishability of both branches is necessary so that C_{i+1} can be compared to both D_i and D_{i+1}.

End Proof of Claim 1.

Claim 2: L_l is implementable.

Proof of Claim 2. By Theorem 8.14, the PSM for L_l can be represented as a global type with mixed choice. It is also 0-reachable, i.e. one can reach a final state from every state. For a 0-reachable global type G, we showed that implementations for $\mathcal{L}_{\text{fin}}(G)$ generalise to $\mathcal{L}_{\text{inf}}(G)$ (Lemma 4.18). Alur et al. [5] showed that a language L of finite words is implementable iff two closure conditions CC_2 and CC_3 hold.

CC_2: If w is FIFO-compliant, complete and for every participant $\mathtt{p} \in \mathcal{P}$, there is $v \in L$ with $w{\Downarrow}_{\Gamma_\mathtt{p}} = v{\Downarrow}_{\Gamma_\mathtt{p}}$, then $v \in L$.

CC_3: If w is FIFO-compliant and for every participant $\mathtt{p} \in \mathcal{P}$, there is $v \in \text{pref}(L)$ with $w{\Downarrow}_{\Gamma_\mathtt{p}} = v{\Downarrow}_{\Gamma_\mathtt{p}}$, then $v \in \text{pref}(L)$.

We show that L_l satisfies CC_2. The proof for CC_3 is analogous. Let w be a FIFO-compliant and complete word such that for every participant $\mathtt{p} \in \mathcal{P}$, there is $v \in L_l$ with $w\Downarrow_{\Gamma_\mathtt{p}} = v\Downarrow_{\Gamma_\mathtt{p}}$. Let us give the structure of $w\Downarrow_{\Gamma_{\mathtt{p}_i}}$ for $i \in \{1, 2, 3\}$.

$$
\begin{aligned}
w\Downarrow_{\Gamma_{\mathtt{p}_3}} &= (\mathtt{p}_3 \triangleright \mathtt{p}_2 ! \circ \cdot \mathtt{p}_3 \triangleleft \mathtt{p}_2 ? \circ \cdot \mathtt{p}_3 \triangleright \mathtt{p}_4 ! \circ \cdot \mathtt{p}_3 \triangleleft \mathtt{p}_4 ? \circ)^{k_3} \text{ for some } k_3 \\
w\Downarrow_{\Gamma_{\mathtt{p}_2}} &= \mathtt{p}_2 \triangleleft \mathtt{p}_3 ? \circ \cdot \mathtt{p}_2 \triangleright \mathtt{p}_1 ! \langle\!\langle \cdot \mathtt{p}_2 \triangleleft \mathtt{p}_1 ? \langle\!\langle \cdot \\
&\quad (\mathtt{p}_2 \triangleright \mathtt{p}_1 ! a_{1,1} \cdot \mathtt{p}_2 \triangleleft \mathtt{p}_1 ? a_{1,1} \cdot \ldots \mathtt{p}_2 \triangleright \mathtt{p}_1 ! a_{1,i_1} \cdot \mathtt{p}_2 \triangleleft \mathtt{p}_1 ? a_{1,i_1}) \cdot \\
&\quad \ldots \\
&\quad (\mathtt{p}_2 \triangleright \mathtt{p}_1 ! a_{k_2,1} \cdot \mathtt{p}_2 \triangleleft \mathtt{p}_1 ? a_{k_2,1} \cdot \ldots \mathtt{p}_2 \triangleright \mathtt{p}_1 ! a_{k_2,i_{k_2}} \cdot \mathtt{p}_2 \triangleleft \mathtt{p}_1 ? a_{k_2,i_{k_2}}) \\
&\quad \text{for some } k_2, i_1, \ldots, i_{k_2} \\
w\Downarrow_{\Gamma_{\mathtt{p}_2}} &= \mathtt{p}_1 \triangleleft \mathtt{p}_2 ? \langle\!\langle \cdot \mathtt{p}_1 \triangleright \mathtt{p}_2 ! \langle\!\langle \cdot \\
&\quad (\mathtt{p}_1 \triangleleft \mathtt{p}_2 ? b_{1,1} \cdot \mathtt{p}_1 \triangleright \mathtt{p}_2 ! b_{1,1} \cdot \ldots \mathtt{p}_1 \triangleleft \mathtt{p}_2 ? b_{1,j_1} \cdot \mathtt{p}_1 \triangleright \mathtt{p}_2 ! b_{1,j_1}) \cdot \\
&\quad \ldots \\
&\quad (\mathtt{p}_1 \triangleleft \mathtt{p}_2 ? b_{k_1,1} \cdot \mathtt{p}_1 \triangleright \mathtt{p}_2 ! b_{k_1,1} \cdot \ldots \mathtt{p}_1 \triangleleft \mathtt{p}_2 ? b_{k_1,j_{k_1}} \cdot \mathtt{p}_1 \triangleright \mathtt{p}_2 ! b_{k_1,j_{k_1}}) \\
&\quad \text{for some } k_1, j_1, \ldots, j_{k_1}
\end{aligned}
$$

(The projections for $\mathtt{p}_4$ and $\mathtt{p}_5$ are analogous to $\mathtt{p}_2$ and $\mathtt{p}_1$ and analogous reasoning applies.) By the fact that w is finite and complete, we know that $k_1 = k_2 = k_3$ and $i_l = j_l$ for every $1 \leq l \leq k$. By the fact that w is FIFO-compliant, the letters coincide, i.e. $a_{i,l} = b_{i,l}$ for every i and l. Therefore, it is straightforward that w can be obtained by reordering $w(C_1, D_1, \ldots, C_k, D_k)$ using $\sim$ and, thus, $w \in L_l$.

End Proof of Claim 2.

Claim 3: If TM has no accepting computation for w, then P_{TM} is implementable.

Proof of Claim 3. Recall that $\mathcal{L}(P_{TM}) = \{\mathtt{p}_2 \to \mathtt{p}_3 : \mathsf{l} \cdot w \mid w \in L'_l\} \uplus \{\mathtt{p}_2 \to \mathtt{p}_3 : \mathsf{r} \cdot w \mid w \in L'_r\}$. If w is not accepted, then L_l and L_r and, hence, L'_l and L'_r coincide by construction. Thus, it is irrelevant for $\mathtt{p}_1$, $\mathtt{p}_4$, and $\mathtt{p}_5$ which branch was taken. By Claim 2, L_l is implementable and so is P_{TM}.

End Proof of Claim 3.

Claim 4: If TM has an accepting computation for w, then P_{TM} is not implementable.

Proof of Claim 4. Let $(u_1, \ldots, u_m)$ be an accepting computation for w. Then, there is $w_u = w(u_1, u_1, \ldots, u_m, u_m)$ and, by construction, it holds that $w_u \notin L_r$. By definition of $\mathcal{L}(P_{TM})$, it holds that $\mathtt{p}_2 \to \mathtt{p}_3 : \mathsf{r} \cdot w_u \cdot w_{end} \notin \mathcal{S}(P_{TM})$. However, for every participant $\mathtt{p} \in \mathcal{P}$, there is $v \in \mathcal{L}(P_{TM})$ such that $w\Downarrow_{\Gamma_\mathtt{p}} = v\Downarrow_{\Gamma_\mathtt{p}}$. Together with the fact that the core language is a subset of the semantics, this contradicts closure condition CC_2, which is a necessary condition for implementability. Thus, P_{TM} is not implementable.

End Proof of Claim 4.

□

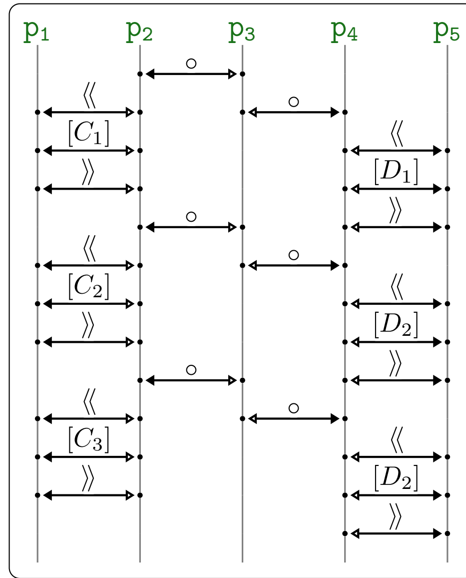


Figure 9.1: MSC representation $\mathrm{msc}(w(C_1, D_1, C_2, D_2, C_3, D_3))$ where the double-sided arrow abbreviates two message interactions of which the direction with the filled tip goes first and $[C_1]$ denotes a sequence of such message interactions.

For the proof of the previous theorem, we construct a sink-final mixed-choice $\Sigma 1$-PSM that is implementable if and only if a Turing Machine accepts a word. Using the same encoding, we show that the soft implementability problem is also undecidable in general.

Theorem 9.2. The soft implementability problem for sink-final mixed-choice $\Sigma 1$-PSMs is undecidable in general.

Proof. We use the same encoding P_{TM} and only highlight the differences.

With w_{end}, our encoding ensures that there is an implementation which is sink-final for every participant – if there is an implementation. For sink-final implementations, the notions of deadlocks and soft deadlocks coincide.

If TM has no accepting computation for w, P_{TM} is implementable and, thus, also softly implementable.

If TM has an accepting computation for w, P_{TM} is not implementable. We revisit *Claim 4* from the previous proof. There, CC_2 breaks. This means that any candidate implementation accepts a word that is not in $\mathcal{S}(P_{TM})$. This yields a contradiction to protocol fidelity, which is also required for soft implementability. □

With our results from Section 8.2.2, we can transform this PSM into a mixed-choice global type. This shows undecidability of the implementability and soft implementability problem for mixed-choice global types – both open problems to date.

Corollary 9.3. Both the implementability problem and the soft implementability problem for mixed-choice global types are undecidable in general.

Proof. From Theorems 9.1 and 9.2, we know that the implementability and soft implementability problem is undecidable for sink-final mixed-choice $\Sigma 1$-PSMs in general. From Theorem 8.14, we know that such PSMs can be transformed into a mixed-choice global type, which proves the claim. □

Remark 9.4 (Sender-driven non-determinism)**.** Our notion of sender-driven choice requires that, for every branching, there is a dedicated sender for every branch and the receiver-message-pairs should be distinct. One could think about weakening this restriction in a way that receiver-message-pairs are allowed not to be distinct but there is a still a dedicated sender. However, if one determinises such a PSM, it can easily yield mixed choice because this notion does not impose any restrictions on subsequent branches. Thus, this notion is not strong enough to regain decidability. In fact, the encoding for undecidability can easily be tweaked to satisfy this notion of sender-driven non-determinism.

Restrictions on choice have not been considered much for HMSCs and the implementability problem for sender-driven HMSCs is an open question, which is also the case for sender-driven PSMs. It is unclear if and how our techniques from Chapter 5 can be adapted to this setting. This more general setting comprises two main challenges: first, PSMs allow to have final states with outgoing transitions and, second, interactions are not specified jointly. While the first challenge seems more feasible but also less interesting, the second one seems significantly more challenging but also more interesting. We leave this investigation for future work.

Chapter 10

Related Work

10.1 High-level Message Sequence Charts

HMSCs were defined in an industry standard [127] as well as studied in academia [95, 54, 53, 51, 107]. Safe and weak realisability has been studied for HMSCs [55, 91, 6]. While safe realisability amounts to our notion of implementability, weak realisability solely requires protocol fidelity and, thus, allows the CSM to deadlock. It is well-known that, given an HMSC, there exists a canonical candidate implementation that implements the HMSC if some implementation exists [5, Thm. 13]. Still, checking HMSC implementability is undecidable in general [91]. Alur et al. [5] identified closure conditions for protocol languages which imply implementability. However, it is unclear how to check these for infinite languages as specified by HMSCs. In addition, the HMSC literature does only consider finite words while the semantics for session types comprise infinite words. We showed, though, with Lemma 4.18 that implementations for languages of finite words generalise to languages of infinite words for 0-reachable global types. This makes the results from HMSC literature applicable to the MST setting but it should also be straightforward to show that the same holds for languages specified as HMSCs. Several restrictions and conditions have been proposed to tame the implementability problem for HMSCs. We group them in choice and structural restrictions.

10.1.1 Choice Restrictions

The first definition of (non-)local choice for HMSCs by Ben-Abdallah et al. [15] suffers from severely restrictive assumptions and only yields finite-state systems.

Given an HMSC specification, research on *implied scenarios*, e.g. Muccini [99], investigates whether there are behaviours which, due to the asynchronous nature of communication, every implementation must allow in addition to the specified ones. In our setting, an implementable protocol specification must not have any implied scenarios. Mooij et al. [97] point out several contradictions of the observations on implied scenarios and non-local choice. Hence, they propose more variants of non-local choices but allow implied scenarios, waiving protocol fidelity.

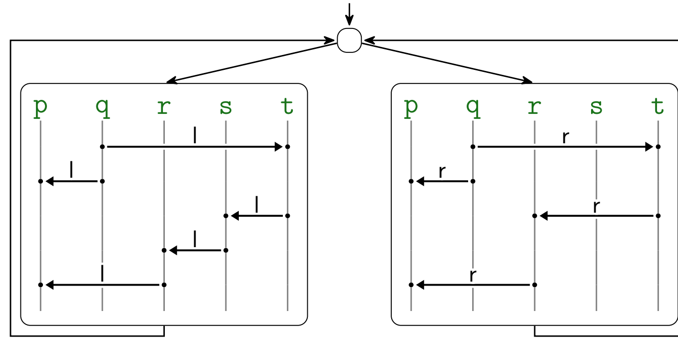


Figure 10.1: Reconstructible HMSC that is not implementable.

Similar to allowing implied scenarios of specifications, Hélouët [61] pointed out that non-local choice has been frequently misunderstood: it actually does not ensure implementability but less ambiguity. Hélouët and Jard proposed the notion of reconstructibility [62] for a quite restrictive setting: first, messages need to be unique in the protocol specification and, second, each node in an HMSC is implicitly final. Unfortunately, we show their results are flawed. Consider the HMSC in Fig. 10.1. (For simplicity, we use the same message identifier in each branch but one can easily index them for uniqueness.) The same protocol can be represented by the following global type:

$$\mu t\,.\ + \begin{cases} \mathtt{q}\to\mathtt{t}:\mathsf{l}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{l}\,.\,\mathtt{t}\to\mathtt{s}:\mathsf{l}\,.\,\mathtt{s}\to\mathtt{r}:\mathsf{l}\,.\,\mathtt{r}\to\mathtt{p}:\mathsf{l}\,.\,t \\ \mathtt{q}\to\mathtt{t}:\mathsf{r}\,.\,\mathtt{q}\to\mathtt{p}:\mathsf{r}\,.\,\mathtt{t}\to\mathtt{r}:\mathsf{r}\,.\,\mathtt{r}\to\mathtt{p}:\mathsf{r}\,.\,t \end{cases}$$

We refer to Fig. 10.1 for the corresponding HMSC. Because their notion of reconstructibility [62, Def. 12] only considers loop-free paths, they report that the HMSC is reconstructible. However, the HMSC is not implementable. Suppose that q first chooses to take the top (resp. left) and then the bottom (resp. right) branch. The message l from s to r can be delayed until after r received r from t. Therefore, r will first send r to p and then l which contradicts with the order of branches taken. This counterexample contradicts their result [62, Thm. 15] and shows that reconstructibility is not sufficient for implementability.

Genest et al. [55] introduced local HMSCs, which require unique minimal events for branches. For local HMSCs, only payload implementability was considered – where additional information can be added to existing messages. For more details on local HMSCs, we refer to Section 4.3.2.

The work by Dan et al. [43] centres around the idea of non-local choice. Intuitively, non-local choice yields scenarios that make it impossible to implement the language. In fact, if a language is not implementable, there is some non-local choice. Thus, checking implementability amounts to checking non-local choice freedom. For this definition, they showed insufficiency of Baker's condition [10] and reformulated the closure conditions for implementability by Alur et al. [5]. In particular, they provide a definition that is based on projected words of a language in contrast to explicit choice. While it is straightforward to check their definition for finite collections of k BMSCs

with n events in $O(k^2 \cdot |\mathcal{P}| + n \cdot |\mathcal{P}|)$, it is unclear how to check their condition for languages with infinitely many elements. The design of such a check is far from trivial as their definition does not give any insight about local behaviour and their algorithm heavily relies on the finite nature of finite collections of BMSCs.

10.1.2 Structural Restrictions

Globally-cooperative HMSCs were independently introduced by Morin [98], as c-HMSCs, and Genest et al. [55]. Lohrey [91] considered $\mathcal{I}$-closed HMSCs, for which checking implementability is PSPACE-complete, and showed that globally-cooperative HMSCs can be translated faithfully to an exponentially larger $\mathcal{I}$-closed HMSC. The reduction yields an EXPSPACE upper bound and Lohrey also proves an EXPSPACE lower bound. This shows that the two classes are equi-expressive but globally-cooperative HMSCs can be exponentially more succinct. In the context of weak realisability, Genest et al. [55] introduced locally-cooperative HMSCs for which weak realisability has linear time complexity. Every globally-cooperative HMSC is locally-cooperative. Thus, checking implementability of locally-cooperative HMSCs is at least EXPSPACE-hard if decidable at all. If one requires the communication graphs of loops to be strongly connected, the class of bounded/regular HMSCs [7, 100] is obtained. Historically, it was introduced before the class of globally-cooperative HMSCs and, after the latter has been introduced, implementability for bounded HMSCs was also shown to be EXPSPACE-complete [91]. This class was independently introduced as regular HMSCs by Muscholl and Peled [100]. Both terms are justified: the language generated by a *regular* HMSC is regular and every *bounded* HMSC can be implemented with universally bounded channels. In fact, a HMSC is bounded if and only if it is a globally-cooperative and has universally bounded channels [55, Prop. 4].

10.2 Session Types

MSTs stem from process algebra and they have been proposed for typing communication channels. We refer to Section 1.4 for details on their evolution from binary [65] to multiparty [67] session types. The latter uses a projection operator with plain merging but multiple ones with full merging have been proposed. Unfortunately, numerous of these turned out to be flawed. We refer to [109, Sec. 8.2] for details. The connection between local types and CSMs has also been studied [46]. Here, we gave conditions for CSMs which make both equi-expressive.

Session types have been incorporated to a number of programming languages [8, 75, 90, 84, 110, 101, 36]. They have also been applied to various other domains like operating systems [48], web services [131], distributed algorithms [81], timed systems [17], cyber-physical systems [94], and smart contracts [44]. Our results could potentially extend the expressivity of the types involved in these applications.

There are first mechanisations of MST frameworks [118, 33, 74, 73, 120]. Tirore et al. [120] also propose a projection operator which is complete with respect to a coinductive projection operator. They define two kinds of global and types local types: inductive and coinductive global types. Projection operators for inductive global types are computable but restricted in presence of recursion and branching. Recent work on projection operators for coinductive global types provide a more general approach but are not computable [56]. Tirore et al. [120] specify a relation to relate inductive and coinductive global types but also their local counterparts. They propose a projection operator for inductive global types that is as powerful as its coinductive counterpart, showing that one does not need to sacrifice computability for what the coinductive projection operator can achieve [120, Thm. 14]. The work in this thesis is different in three regards. First, our notion of completeness is different: any global type for which the subset projection is undefined cannot be (softly) implemented by any CSM. Regarding this, we also showed that, for soft implementations of sender-driven global types, CSMs are not more expressive than local types and can be transformed into such. Second, Tirore et al. [120] use what is equivalent to what we call plain merge (for both the inductive and coinductive projection operator): only the two participants in a choice can learn about it. This is a severe restriction of applicability. We refer to Definition 3.7 and Section 3.1.6 for details on different merge operators and their brittleness. Here, we present an example that is inspired by Example 3.11. A buyer a decides whether to buy an item or not and the seller s propagates this information to the payment service p:

$$+\begin{cases} \mathtt{a}\to\mathtt{s}:\mathsf{buy}\,.\,\mathtt{s}\to\mathtt{p}:\mathsf{buy}\,.\,0 \\ \mathtt{a}\to\mathtt{s}:\mathsf{no}\,.\,\mathtt{s}\to\mathtt{p}:\mathsf{no}\,.\,0 \end{cases} .$$

It is easy to imagine that such behaviour can happen in a protocol. However, the plain merge operator prohibits that p has different continuations after the choice. Third, they consider a directed choice setting and we have shown that extending to the sender-driven choice setting comes with its own challenges.

10.2.1 Generalising Restrictions on Choice

The work by Castagna et al. [29] is the only one to present a projection that aims for completeness. Their semantic conditions, however, are not effectively computable and their notion of completeness is “less demanding than the classical ones”[29]. They consider multiple implementations, generating different sets of traces, to be sound and complete with regard to a single global type [29, Sec. 5.3]. In addition, the algorithmic version of their conditions does not use global information as our message availability analysis does. For instance, our projection operators can project the following example [29, p. 19] but their algorithmic version cannot:

$$\big(\mathtt{p}\to\mathtt{r}:\mathsf{a}\,.\,\mathtt{r}\to\mathtt{p}:\mathsf{a}\,.\,\mathtt{p}\to\mathtt{q}:\mathsf{a}\,.\,\mathtt{q}\to\mathtt{r}:\mathsf{b}\,.\,0\big) + \big(\mathtt{p}\to\mathtt{q}:\mathsf{a}\,.\,\mathtt{q}\to\mathtt{r}:\mathsf{b}\,.\,0\big) \ .$$

Hu and Yoshida [70] syntactically allow a sender to send to different receivers in global and local types as well as a receiver to receive from different senders in local types. However, their projection is only defined if a receiver receives messages from a single participant. From our evaluation, all the examples that need the generalised projection are rejected by their projection. Recently, Castellani et al. [31] investigated ways to allow local types to specify receptions from multiple senders for reversible computations but only in the synchronous setting. Similarly, for synchronous communication only, Jongmans and Yoshida [76] discuss generalising choice in MSTs. Because their calculus has an explicit parallel composition, they can emulate some asynchronous communication but their channels have bag semantics instead of FIFO queues. The correctness of the projection also computes causality among messages, as we do for both presented projection operators, and shares the idea of annotating local types with the generalised projection operator.

Scalas et al. [109] start from a given concrete implementation for each participant and, through typing, they are shown to simulate local types. Then, the behaviour of the parallel composition of the local types is model-checked against correctness properties, but, not given a global type, do not consider protocol fidelity. For asynchronous systems, model checking is undecidable and, thus, their approach is incomplete. Dagnino et al. [42] and Castellani et al. [30] have a setup where the parallel composition of concrete participant implementations is type-checked against a so-called deconfined global type, which describes a finite-control language of send and receive events. These deconfined global types need to be checked against correctness properties, which is shown to be undecidable.

10.2.2 MST-based Works

Choreography Automata [11] are syntactically similar to Σ1-PSMs. However, their semantics is considered literally as the core language of the automaton, i.e. not employing the closure under $\sim$. Therefore, such reorderings need to be explicitly represented, preventing finite state representations for common communication patterns. For the asynchronous setting, their conditions for implementability do not acknowledge the partial order for messages from different senders, leading to unsoundness. Consider the choreography automaton in Fig. 10.2. It can also be represented as a global type:

$$+\begin{cases} \mathtt{p}\!\to\!\mathtt{s}:\mathtt{m}_1\,.\,\mathtt{p}\!\to\!\mathtt{t}:\mathtt{m}\,.\,\mathtt{p}\!\to\!\mathtt{s}:\mathtt{m}\,.\,\mathtt{s}\!\to\!\mathtt{t}:\mathtt{m}\,.\,\mathtt{t}\!\to\!\mathtt{p}:\mathtt{m}_1\,.\,0 \\ \mathtt{p}\!\to\!\mathtt{s}:\mathtt{m}_2\,.\,\mathtt{s}\!\to\!\mathtt{t}:\mathtt{m}\,.\,\mathtt{s}\!\to\!\mathtt{p}:\mathtt{m}\,.\,\mathtt{p}\!\to\!\mathtt{s}:\mathtt{m}\,.\,\mathtt{p}\!\to\!\mathtt{t}:\mathtt{m}\,.\,\mathtt{t}\!\to\!\mathtt{p}:\mathtt{m}_2\,.\,0 \end{cases}$$

It is well-formed according to their conditions. However, $\mathtt{t}$ cannot determine which branch was chosen since the messages $\mathtt{m}$ from $\mathtt{p}$ and $\mathtt{s}$ are not ordered when sent asynchronously. As a result, it can send $\mathtt{m}_2$ in the top branch which is not specified as well as $\mathtt{m}_1$ in the bottom branch.

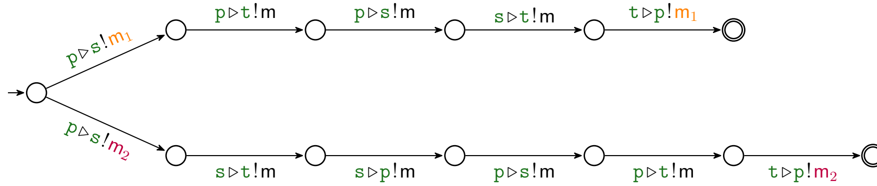


Figure 10.2: Conditions for choreography automata are unsound in the asynchronous setting.

Lange et al. [85] have shown how to obtain graphical choreographies from CSM executions. Unfortunately, they cannot fully handle unbounded FIFO channels as their method internally uses Petri nets. Still, their branching property [85, Def. 3.5] consists of similar – even though more restrictive – conditions as our MST framework: a single participant chooses at each branch but participants have to learn with the first received message or do not commit any action until the branches merge back. In our developments, we allow a participant to learn later.

10.2.3 Subtyping

For the generalised projection operator, we use local types directly as implementations for participants, while, for the subset projection operator, we directly construct CSMs. Our type system uses these CSMs for type-checking. Intuitively, subtyping investigates ways to give implementation freedom while preserving the desired correctness properties. There are two notions of subtyping: *synchronous* and *asynchronous* subtyping. The terminology is a bit unfortunate as both notions of subtyping can apply in an asynchronous setting, as we consider. It is, though, the case that asynchronous subtyping can only be used in an asynchronous setting. Intuitively, synchronous subtyping allows a participant to implement fewer sends and more receives while asynchronous subtyping allows a participant to reorder its local actions.

Synchronous subtyping, as established for the directed choice setting, cannot simply be applied to the sender-driven choice setting. Intuitively, adding receives can lead to unspecified behaviour if the message is indeed available through to the partial order of messages by different senders (cf. Section 6.5) – with directed choice, this cannot be the case. Inspired by the conditions in Chapter 5, we developed sound and complete conditions for the synchronous subtyping problem for CSMs, in the context of sender-driven global types, which can be checked in polynomial time [88]. In addition, we gave a variant of these conditions which allows to check if a CSM implements a sender-driven global type.

Asynchronous subtyping generalises what we allow with the relaxed indistinguishability relation $\approx$. While $\approx$ only allows to reorder receives, asynchronous subtyping investigates which events of a participant can be reordered. For $\approx$-implementability, we can still relate language generated by the implementation and the one specified

by the global protocol. For asynchronous subtyping however, this is not the case, completely waiving protocol fidelity as property of interest. In contrast, in the context of synchronous subtyping, the updated implementation should most likely still implement a subset of the specified language.

For further details on subtyping, we refer to work by Lange and Yoshida [86], Bravetti et al. [24], Chen et al. [38, 37], and Li et al. [88].

10.2.4 Extensions

A number of extensions for MSTs have been considered. These include but are not limited to parametrised session types [35, 47], dependent session types [121, 47, 123], gradual session types [72], and delegation [66, 67, 32]. We do consider a restricted form of delegation with our type system, which requires a strict partial order between global types, indicating which global type can delegate local behaviour from which session. Bejleri et al. [14] survey and uniformly present a number of advanced features. Recently, fault-tolerant MSTs [128, 13] allow to (partially) waive the strong assumptions on reliable channels. Context-free session types [119, 77] allow to specify binary sessions that are not representable with finite-control. Many of these extensions are interesting directions for future work, extending the applicability of our novel techniques.

10.3 Communicating State Machines and Channels

While we consider finite state machines as model for processes, research has also been conducted on communicating systems where processes are given more computational power, e.g. pushdown automata [63, 125, 4]. However, as noted before, our setting is already Turing-powerful. In Section 7.1, we surveyed how channel restrictions can yield decidability. Incomplete approaches consider subclasses which enable the effective computation of symbolic representations (of channel contents) for reachable states [18, 78]. Other approaches change the semantics of channels, e.g. by making them lossy [3, 2, 78], input-bounded [19], or by restricting the communication topology [103, 124]. The concept of existential boundedness [52], which is the basis of our definition of PSMs, was initially defined for CSMs and yields decidability of control state reachability. The same holds for synchronisability [22, 59]. We consider the most recent definitions of synchronisability [59] and we refer to the work by Finkel and Lozes [49] and Bouajjani et al. [22] for earlier work on synchronisability. Bollig et al. [21] studied the connection of different notions of synchronisability for MSCs and MSO logic which yields interesting decidability results. We refer to their work for more details but briefly point to the slightly different use of terminology: k-synchronisability is called weak (k-)synchronisability by Bollig where the omission of k indicates a system is synchronisable for some k; while strong (k-)synchronisability does solely apply to the mailbox setting, i.e. where each participant has a single channel for incoming messages.

10.4 Choreographic Programming

In this thesis, we consider a top-down approach for the design of communication in a system. In general, there are two main top-down uses of a Global Protocol Specification (GPS).

The first takes the view that the GPS is a description of the entire evolution of the closed system. Choreographic programming [40, 57, 64] takes this view and, as a logical consequence, the endpoint projection (EPP) returns the full local implementation of each thread. To do so, the GPS needs to include information about the manipulations of local states of the parties. Sometimes MST work also adopts this view, with the difference that local behaviour is only required to describe the communication skeleton of the computations of the threads, while local computation is implemented separately by processes that are shown to adhere to the communication skeleton by means of type checking.

The second view is that the GPS is a description of the evolution of a system when ignoring everything but a single instance of the protocol. This means that the concrete system might include – additionally to what is modelled in the GPS – local computation, and arbitrary initiation of new sessions of the same global type. In this approach, the local types are again only the skeleton of communication for a single instance of the protocol, and a type system can be used to link that description to the actual global process implementation. In this context, the properties of the protocol like deadlock-freedom might not be reflected as properties of the final implementation "for free" (unless suitably weakened), and so some MST systems build on top of the single-session deadlock-freedom guarantee some additional checks to guarantee global deadlock-freedom.

Our work starts from the second view: that the GPS is an abstract view of the communication structure of a single instance of a protocol. In this setting, it is natural to consider (asynchronous) finite-control protocols, i.e. protocols where the communication structure can be described using finitely many local states per participant. Note that the behaviour of asynchronous finite-control protocols is infinite-state and actually Turing-complete. For this class, it is not a priori clear whether implementability can be made sound, complete and cheap. We show that it is for global types with sender-driven choice.

In choreographies, since one typically works with non-finite-control-state GPSs, we know theoretically that the hopes to have a complete and decidable EPP are slim, justifying giving up on completeness. Our work is still potentially useful for choreographies, however: if a choreography can be represented as sender-driven global type, the EPP can be computed with our results. The big advantage of completeness is that any failure of the EPP is entirely explainable as a flaw in the design of the GPS itself (and not just a limitation of the tool). We can produce examples that cannot be projected using EPPs from the literature but can be projected using our method.

A simple example is the following (also using choreography syntax and omitting the payloads as they are irrelevant):

$$\textit{if}\ \mathtt{p}.\star\ \ \textit{then}\ (\mathtt{p}\rightarrow\mathtt{q}:_\,.\,\mathtt{q}\rightarrow\mathtt{s}:_)\ \textit{else}\ (\mathtt{p}\rightarrow\mathtt{s}:_\,.\,\mathtt{s}\rightarrow\mathtt{q}:_)\ \ .$$

(Here we use $\mathtt{p}.\star$ to denote a non-deterministic choice of $\mathtt{p}$ but this could be replaced by some non-trivial guard.) The example is syntactically valid in [41] and easily encoded as a global type with sender-driven choice. However, their EPP would be undefined for $\mathtt{q}$ and $\mathtt{s}$. Consider these two branches that need to be merged when projecting onto $\mathtt{q}$:

$$B_1 := \mathtt{q}\triangleleft\mathtt{p}?_\,.\,\mathtt{q}\triangleright\mathtt{s}!_\,.\,0$$
$$B_2 := \mathtt{q}\triangleleft\mathtt{s}?_\,.\,0$$

The EPP of [41] uses the merge from [28], which can only merge same sender receives, and would fail to merge $\mathtt{q}\triangleleft\mathtt{p}?_$ and $\mathtt{q}\triangleleft\mathtt{s}?_$. Our results would instead produce the desired projection.

The notion of *unique point of choice* [83] has also been considered for choreographies. This restriction and sender-driven choice seem to be incomparable conditions. Sender-driven choice does not insist on the participants coinciding for the branches; unique point of choice does not seem to insist on same sender-receiver branches differing in the message label. In addition, it is unclear how unique point of choice would interact with the presence of loops, which are not considered in [83].

For future work, one could aim at designing an EPP for choreographies which uses our subset projection operator on GPSs that are essentially fitting our class and soundly but incompletely covers the rest.

Chapter 11

Conclusion

In this thesis, we investigated the implementability problem for asynchronous communication protocols. We did this from the joint perspective of Multiparty Session Types (MSTs) and High-level Message Sequence Charts (HMSCs). Most notably, we showed that restrictions on choice have tremendous impact on the decidability of the implementability problem for global types from MSTs. In the beginning, we presented a generalised projection operator that uses novel message causality analysis to make the efficient MST verification techniques applicable to sender-driven global types, necessary for many common communication patterns from distributed computing. Still, we showed that the classical projection approach does not tolerate minor implementability-preserving changes, exemplifying the incompleteness of this approach. Subsequently, we proved decidability of the implementability problem for global types with sender-driven choice, using the first formal encoding of global types as HMSCs. We also elaborated how earlier work from the HMSC literature becomes applicable with this encoding. While our earlier results answered an open question and are thus of theoretical interest, they do not lend themselves to an efficient implementation. This is why we developed the first direct and complete projection operator for global types with sender-driven choice. With this, we provided the first upper bound for the respective implementability problem and proved it to be in PSPACE. Despite, we showed, using a prototype implementation, that local specifications for sender-driven global types can be obtained fairly quickly. If they cannot be constructed, we know, from our completeness result, that the protocol is not implementable. This was not the case for any previous approach. Our approach allows to pinpoint the flaw in the design of the protocol. More broadly, session types have been applied to various different domains. Now that we overcame their brittleness with an efficient and complete approach for a more general setting, namely sender-driven choice, we believe session types can even have wider impact and applicability. With our subset projection operator, we proved communicating state machines advantageous for projection. With our type system, we showed that they are also useful for type-checking, allowing delegation between sessions. In a nutshell, the use of communicating state machines as interface between projection and type-checking improves generality without loosing efficiency. Notably, our setting with

sender-driven choice makes subtyping more challenging and we hope that our automata-based techniques can provide insights to obtain meaningful results. The computational power of communicating state machines, the standard model for distributed settings, hinges on the capabilities of its channels. Hence, we considered various channel restrictions from the literature and compared them for protocol specifications as well as general communicating state machines. Inspired by existential boundedness, one of these channel restrictions, we proposed protocol state machines (PSMs), which are basically automata where transitions are labelled with send and receive events, that is more expressive than both global types from MSTs and HMSCs. This is why one might want to apply certain sanity checks to PSMs, which are always satisfied by global types and HMSCs by construction. Here, we gave a number of such sanity checks but expect that this can be generalised to a class of sanity checks, using Büchi automata. We hope that PSMs can be a building block for a unifying theory of communication protocol design. As a first step, we investigated global types, in relation to PSMs, and found that many of their syntactic restrictions are not restrictive in terms of which protocols can be specified. This helps to distinguish between restrictions that change the class of protocols one can define and the ones that do not and, therefore, shall help to identify interesting directions for future work. We proved the implementability problem for PSMs with mixed choice to be undecidable in general and, with our results on expressivity, this settled an open problem: the implementability problem for global types with mixed choice is undecidable in general. Thus, if one hopes for complete algorithmics to check implementability, one will need to design and assume further restrictions in addition to mixed choice. The implementability problem for HMSCs with directed and sender-driven choice, and hence the one for PSMs, is still open and it will be interesting to see if and how our techniques can be generalised to solve these problems. The result of such investigations could be enhanced by generalising the semantics of infinite protocol executions using Büchi automata.

Bibliography

[1] Martín Abadi and Leslie Lamport. "The Existence of Refinement Mappings". In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988.*

[2] Parosh Aziz Abdulla, C. Aiswarya, and Mohamed Faouzi Atig. "Data Communicating Processes with Unreliable Channels". In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016.* Ed. by Martin Grohe, Eric Koskinen, and Natarajan Shankar. ACM, 2016, pp. 166–175. DOI: 10.1145/2933575.2934535.

[3] Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. "On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels". In: *Computer Aided Verification, 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings.* Ed. by Alan J. Hu and Moshe Y. Vardi. Vol. 1427. Lecture Notes in Computer Science. Springer, 1998, pp. 305–318. DOI: 10.1007/BFb0028754.

[4] C. Aiswarya, Paul Gastin, and K. Narayan Kumar. "Verifying Communicating Multi-pushdown Systems via Split-Width". In: *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings.* Ed. by Franck Cassez and Jean-François Raskin. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 1–17. DOI: 10.1007/978-3-319-11936-6_1.

[5] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. "Inference of Message Sequence Charts". In: *IEEE Trans. Software Eng.* 29.7 (2003), pp. 623–633. DOI: 10.1109/TSE.2003.1214326.

[6] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. "Realizability and verification of MSC graphs". In: *Theor. Comput. Sci.* 331.1 (2005), pp. 97–114. DOI: 10.1016/j.tcs.2004.09.034.

[7] Rajeev Alur and Mihalis Yannakakis. "Model Checking of Message Sequence Charts". In: *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings.* Ed. by Jos C. M. Baeten and Sjouke Mauw. Vol. 1664. Lecture Notes in Computer Science. Springer, 1999, pp. 114–129. DOI: 10.1007/3-540-48320-9_10.

[8] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. "Behavioral Types in Programming Languages". In: *Found. Trends Program. Lang.* 3.2-3 (2016), pp. 95–230. DOI: 10.1561/2500000031.

[9] Dean N. Arden. "Delayed-logic and finite-state machines". In: *2nd Annual Symposium on Switching Circuit Theory and Logical Design, Detroit, Michigan, USA, October 17-20, 1961.* IEEE Computer Society, 1961, pp. 133–151. DOI: 10.1109/FOCS.1961.13.

[10] Paul Baker, Paul Bristow, Clive Jervis, David J. King, Robert Thomson, Bill Mitchell, and Simon Burton. "Detecting and resolving semantic pathologies in UML sequence diagrams". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005.* Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 50–59. DOI: 10.1145/1081706.1081716.

[11] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. "Choreography Automata". In: *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings.* Ed. by Simon Bliudze and Laura Bocchi. Vol. 12134. Lecture Notes in Computer Science. Springer, 2020, pp. 86–106. DOI: 10.1007/978-3-030-50029-0_6.

[12] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics.* Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[13] Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. "Generalised Multiparty Session Types with Crash-Stop Failures". In: *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland.* Ed. by Bartek Klin, Slawomir Lasota, and Anca Muscholl. Vol. 243. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 35:1–35:25. DOI: 10.4230/LIPIcs.CONCUR.2022.35.

[14] Andi Bejleri, Elton Domnori, Malte Viering, Patrick Eugster, and Mira Mezini. "Comprehensive Multiparty Session Types". In: *Art Sci. Eng. Program.* 3.3 (2019), p. 6. DOI: 10.22152/programming-journal.org/2019/3/6.

[15] Hanêne Ben-Abdallah and Stefan Leue. “Syntactic Detection of Process Divergence and Non-local Choice inMessage Sequence Charts”. In: *Tools and Algorithms for Construction and Analysis of Systems, Third International Workshop, TACAS ’97, Enschede, The Netherlands, April 2-4, 1997, Proceedings.* Ed. by Ed Brinksma. Vol. 1217. Lecture Notes in Computer Science. Springer, 1997, pp. 259–274. DOI: 10.1007/BFb0035393.

[16] Laura Bocchi, Julien Lange, and Nobuko Yoshida. “Meeting Deadlines Together”. In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015.* Ed. by Luca Aceto and David de Frutos-Escrig. Vol. 42. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 283–296. DOI: 10.4230/LIPIcs.CONCUR.2015.283.

[17] Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. “Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes”. In: *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings.* Ed. by Luís Caires. Vol. 11423. Lecture Notes in Computer Science. Springer, 2019, pp. 583–610. DOI: 10.1007/978-3-030-17184-1_21.

[18] Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. “The Power of QDDs (Extended Abstract)”. In: *Static Analysis, 4th International Symposium, SAS ’97, Paris, France, September 8-10, 1997, Proceedings.* Ed. by Pascal Van Hentenryck. Vol. 1302. Lecture Notes in Computer Science. Springer, 1997, pp. 172–186. DOI: 10.1007/BFb0032741.

[19] Benedikt Bollig, Alain Finkel, and Amrita Suresh. “Bounded Reachability Problems Are Decidable in FIFO Machines”. In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference).* Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 49:1–49:17. DOI: 10.4230/LIPIcs.CONCUR.2020.49.

[20] Benedikt Bollig and Paul Gastin. “Non-Sequential Theory of Distributed Systems”. In: *CoRR* abs/1904.06942 (2019). DOI: 10.48550/arXiv.1904.06942. arXiv: 1904.06942.

[21] Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes, and Amrita Suresh. “A Unifying Framework for Deciding Synchronizability”. In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference.* Ed. by Serge Haddad and Daniele Varacca. Vol. 203. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 14:1–14:18. DOI: 10.4230/LIPIcs.CONCUR.2021.14.

[22] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. "On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II.* Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 372–391. DOI: 10.1007/978-3-319-96142-2_23.

[23] Daniel Brand and Pitro Zafiropulo. "On Communicating Finite-State Machines". In: *J. ACM* 30.2 (1983), pp. 323–342. DOI: 10.1145/322374.322380.

[24] Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. "On the boundary between decidability and undecidability of asynchronous session subtyping". In: *Theor. Comput. Sci.* 722 (2018), pp. 19–51. DOI: 10.1016/j.tcs.2018.02.010.

[25] Anne Brüggemann-Klein. "Regular Expressions into Finite Automata". In: *LATIN '92, 1st Latin American Symposium on Theoretical Informatics, São Paulo, Brazil, April 6-10, 1992, Proceedings.* Ed. by Imre Simon. Vol. 583. Lecture Notes in Computer Science. Springer, 1992, pp. 87–98. DOI: 10.1007/BFb0023820.

[26] Anne Brüggemann-Klein and Derick Wood. "One-Unambiguous Regular Languages". In: *Inf. Comput.* 142.2 (1998), pp. 182–206. DOI: 10.1006/inco.1997.2695.

[27] Janusz A. Brzozowski. "Derivatives of Regular Expressions". In: *J. ACM* 11.4 (1964), pp. 481–494. DOI: 10.1145/321239.321249.

[28] Marco Carbone, Kohei Honda, and Nobuko Yoshida. "Structured Communication-Centered Programming for Web Services". In: *ACM Trans. Program. Lang. Syst.* 34.2 (2012), 8:1–8:78. DOI: 10.1145/2220365.2220367.

[29] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. "On Global Types and Multi-Party Session". In: *Log. Methods Comput. Sci.* 8.1 (2012). DOI: 10.2168/LMCS-8(1:24)2012.

[30] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. "Asynchronous Sessions with Input Races". In: *Proceedings of the 13th International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@ETAPS 2022, Munich, Germany, 3rd April 2022.* Ed. by Marco Carbone and Rumyana Neykova. Vol. 356. EPTCS. 2022, pp. 12–23. DOI: 10.4204/EPTCS.356.2.

[31] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. "Reversible sessions with flexible choices". In: *Acta Informatica* 56.7-8 (2019), pp. 553–583. DOI: 10.1007/s00236-019-00332-y.

[32] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. "Global types with internal delegation". In: *Theor. Comput. Sci.* 807 (2020), pp. 128–153. DOI: 10.1016/j.tcs.2019.09.027.

[33] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. “Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 237–251. DOI: 10.1145/3453483.3454041.

[34] Gérard Cécé and Alain Finkel. “Verification of programs with half-duplex communication”. In: *Inf. Comput.* 202.2 (2005), pp. 166–190. DOI: 10.1016/j.ic.2005.05.006.

[35] Minas Charalambides, Peter Dinges, and Gul A. Agha. “Parameterized, concurrent session types for asynchronous multi-actor interactions”. In: *Sci. Comput. Program.* 115-116 (2016), pp. 100–126. DOI: 10.1016/j.scico.2015.10.006.

[36] Ruofei Chen, Stephanie Balzer, and Bernardo Toninho. “Ferrite: A Judgmental Embedding of Session Types in Rust”. In: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany.* Ed. by Karim Ali and Jan Vitek. Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 22:1–22:28. DOI: 10.4230/LIPIcs.ECOOP.2022.22.

[37] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. “On the Preciseness of Subtyping in Session Types”. In: *Log. Methods Comput. Sci.* 13.2 (2017). DOI: 10.23638/LMCS-13(2:12)2017.

[38] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. “On the Preciseness of Subtyping in Session Types”. In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming, Kent, Canterbury, United Kingdom, September 8-10, 2014.* Ed. by Olaf Chitil, Andy King, and Olivier Danvy. ACM, 2014, pp. 135–146. DOI: 10.1145/2643135.2643138.

[39] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. “A Gentle Introduction to Multiparty Asynchronous Session Types”. In: *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures.* Ed. by Marco Bernardo and Einar Broch Johnsen. Vol. 9104. Lecture Notes in Computer Science. Springer, 2015, pp. 146–178. DOI: 10.1007/978-3-319-18941-3_4.

[40] Luís Cruz-Filipe and Fabrizio Montesi. “A core model for choreographic programming”. In: *Theor. Comput. Sci.* 802 (2020), pp. 38–66. DOI: 10.1016/j.tcs.2019.07.005.

[41] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. “Communications in choreographies, revisited”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018.* Ed. by Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir. ACM, 2018, pp. 1248–1255. DOI: 10.1145/3167132.3167267.

[42] Francesco Dagnino, Paola Giannini, and Mariangiola Dezani-Ciancaglini. “Deconfined Global Types for Asynchronous Sessions”. In: *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings.* Ed. by Ferruccio Damiani and Ornela Dardha. Vol. 12717. Lecture Notes in Computer Science. Springer, 2021, pp. 41–60. DOI: 10.1007/978-3-030-78142-2_3.

[43] Haitao Dan, Robert M. Hierons, and Steve Counsell. “Non-local Choice and Implied Scenarios”. In: *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010, Pisa, Italy, 13-18 September 2010.* Ed. by José Luiz Fiadeiro, Stefania Gnesi, and Andrea Maggiolo-Schettini. IEEE Computer Society, 2010, pp. 53–62. DOI: 10.1109/SEFM.2010.14.

[44] Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. “Resource-Aware Session Types for Digital Contracts”. In: *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021.* IEEE, 2021, pp. 1–16. DOI: 10.1109/CSF51468.2021.00004.

[45] Pierre-Malo Deniélou and Nobuko Yoshida. “Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types”. In: *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II.* Ed. by Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg. Vol. 7966. Lecture Notes in Computer Science. Springer, 2013, pp. 174–186. DOI: 10.1007/978-3-642-39212-2_18.

[46] Pierre-Malo Deniélou and Nobuko Yoshida. “Multiparty Session Types Meet Communicating Automata”. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings.* Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 194–213. DOI: 10.1007/978-3-642-28869-2_10.

[47] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. “Parameterised Multiparty Session Types”. In: *Log. Methods Comput. Sci.* 8.4 (2012). DOI: 10.2168/LMCS-8(4:6)2012.

[48] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. "Language support for fast and reliable message-based communication in singularity OS". In: *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*. Ed. by Yolande Berbers and Willy Zwaenepoel. ACM, 2006, pp. 177–190. DOI: 10.1145/1217935.1217953.

[49] Alain Finkel and Étienne Lozes. "Synchronizability of Communicating Finite State Machines is not Decidable". In: *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*. Ed. by Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl. Vol. 80. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 122:1–122:14. DOI: 10.4230/LIPIcs.ICALP.2017.122.

[50] Paul Gastin. "Infinite Traces". In: *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*. Ed. by Irène Guessarian. Vol. 469. Lecture Notes in Computer Science. Springer, 1990, pp. 277–308. DOI: 10.1007/3-540-53479-2_12.

[51] Thomas Gazagnaire, Blaise Genest, Loïc Hélouët, P. S. Thiagarajan, and Shaofa Yang. "Causal Message Sequence Charts". In: *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. Lecture Notes in Computer Science. Springer, 2007, pp. 166–180. DOI: 10.1007/978-3-540-74407-8_12.

[52] Blaise Genest, Dietrich Kuske, and Anca Muscholl. "On Communicating Automata with Bounded Channels". In: *Fundam. Inform.* 80.1-3 (2007), pp. 147–167. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09.

[53] Blaise Genest and Anca Muscholl. "Message Sequence Charts: A Survey". In: *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005), 6-9 June 2005, St. Malo, France*. IEEE Computer Society, 2005, pp. 2–4. DOI: 10.1109/ACSD.2005.25.

[54] Blaise Genest, Anca Muscholl, and Doron A. Peled. "Message Sequence Charts". In: *Lectures on Concurrency and Petri Nets, Advances in Petri Nets [This tutorial volume originates from the 4th Advanced Course on Petri Nets, ACPN 2003, held in Eichstätt, Germany in September 2003. In addition to lectures given at ACPN 2003, additional chapters have been commissioned]*. Ed. by Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg. Vol. 3098. Lecture Notes in Computer Science. Springer, 2003, pp. 537–558. DOI: 10.1007/978-3-540-27755-2_15.

[55] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. "Infinite-state high-level MSCs: Model-checking and realizability". In: *J. Comput. Syst. Sci.* 72.4 (2006), pp. 617–647. DOI: 10.1016/j.jcss.2005.09.007.

[56] Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. "Precise subtyping for synchronous multiparty sessions". In: *J. Log. Algebraic Methods Program.* 104 (2019), pp. 127–173. DOI: 10.1016/j.jlamp.2018.12.002.

[57] Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. "Multiparty Languages: The Choreographic and Multitier Cases (Pearl)". In: *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference).* Ed. by Anders Møller and Manu Sridharan. Vol. 194. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 22:1–22:27. DOI: 10.4230/LIPIcs.ECOOP.2021.22.

[58] Jean-Yves Girard. "Linear Logic". In: *Theor. Comput. Sci.* 50 (1987), pp. 1–102. DOI: 10.1016/0304-3975(87)90045-4.

[59] Cinzia Di Giusto, Laetitia Laversa, and Étienne Lozes. "On the k-synchronizability of Systems". In: *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* Ed. by Jean Goubault-Larrecq and Barbara König. Vol. 12077. Lecture Notes in Computer Science. Springer, 2020, pp. 157–176. DOI: 10.1007/978-3-030-45231-5_9.

[60] Rob van Glabbeek, Peter Höfner, and Ross Horne. "Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* IEEE, 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470531.

[61] Loïc Hélouët. "Some Pathological Message Sequence Charts, and How to Detect Them". In: *SDL 2001: Meeting UML, 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings.* Ed. by Rick Reed and Jeanne Reed. Vol. 2078. Lecture Notes in Computer Science. Springer, 2001, pp. 348–364. DOI: 10.1007/3-540-48213-X_22.

[62] Loïc Hélouët and Claude Jard. "Conditions for synthesis of communicating automata from HMSCs". In: *In 5th International Workshop on Formal Methods for Industrial Critical Systems (FMICS).* 2000.

[63] Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. "Reachability Analysis of Communicating Pushdown Systems". In: *Log. Methods Comput. Sci.* 8.3 (2012). DOI: 10.2168/LMCS-8(3:23)2012.

[64] Andrew K. Hirsch and Deepak Garg. "Pirouette: higher-order typed functional choreographies". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–27. DOI: 10.1145/3498684.

[65] Kohei Honda. "Types for Dyadic Interaction". In: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings.* Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 509–523. DOI: 10.1007/3-540-57208-2_35.

[66] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings.* Ed. by Chris Hankin. Vol. 1381. Lecture Notes in Computer Science. Springer, 1998, pp. 122–138. DOI: 10.1007/BFb0053567.

[67] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty asynchronous session types". In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 273–284. DOI: 10.1145/1328438.1328472.

[68] Kohei Honda, Nobuko Yoshida, and Marco Carbone. "Multiparty Asynchronous Session Types". In: *J. ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[69] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. ISBN: 0-201-02988-X.

[70] Raymond Hu and Nobuko Yoshida. "Explicit Connection Actions in Multiparty Session Types". In: *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings.* Ed. by Marieke Huisman and Julia Rubin. Vol. 10202. Lecture Notes in Computer Science. Springer, 2017, pp. 116–133. DOI: 10.1007/978-3-662-54494-5_7.

[71] Raymond Hu and Nobuko Yoshida. "Hybrid Session Verification Through Endpoint API Generation". In: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* Ed. by Perdita Stevens and Andrzej Wasowski. Vol. 9633. Lecture Notes in Computer Science. Springer, 2016, pp. 401–418. DOI: 10.1007/978-3-662-49665-7_24.

[72] Atsushi Igarashi, Peter Thiemann, Yuya Tsuda, Vasco T. Vasconcelos, and Philip Wadler. “Gradual session types”. In: *J. Funct. Program.* 29 (2019), e17. DOI: 10.1017/S0956796819000169.

[73] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. “Connectivity graphs: a method for proving deadlock freedom based on separation logic”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–33. DOI: 10.1145/3498662.

[74] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. “Multiparty GV: functional multiparty session types with certified deadlock freedom”. In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 466–495. DOI: 10.1145/3547638.

[75] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. “Session types for Rust”. In: *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015.* Ed. by Patrick Bahr and Sebastian Erdweg. ACM, 2015, pp. 13–22. DOI: 10.1145/2808098.2808100.

[76] Sung-Shik Jongmans and Nobuko Yoshida. “Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types”. In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 251–279. DOI: 10.1007/978-3-030-44914-8_10.

[77] Alex C. Keizer, Henning Basold, and Jorge A. Pérez. “Session Coalgebras: A Coalgebraic View on Regular and Context-free Session Types”. In: *ACM Trans. Program. Lang. Syst.* 44.3 (2022), 18:1–18:45. DOI: 10.1145/3527633.

[78] Chris Köcher. “Reachability Problems on Reliable and Lossy Queue Automata”. In: *Theory Comput. Syst.* 65.8 (2021), pp. 1211–1242. DOI: 10.1007/s00224-021-10031-2.

[79] Denes König. *Theory of finite and infinite graphs.* Birkhäuser, 1990. ISBN: 978-3-7643-3389-8.

[80] Denes König. *Über eine Schlussweise aus dem Endlichen ins Unendliche.* Acta Sci. Math. (Szeged), 1927. URL: https://acta.bibl.u-szeged.hu/13338/1/math_003_121-130.pdf.

[81] Dimitrios Kouzapas, Ramunas Gutkovas, A. Laura Voinea, and Simon J. Gay. “A Session Type System for Asynchronous Unreliable Broadcast Communication”. In: *CoRR* abs/1902.01353 (2019). arXiv: 1902.01353. URL: http://arxiv.org/abs/1902.01353.

[82] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[83] Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. "Bridging the Gap between Interaction- and Process-Oriented Choreographies". In: *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*. Ed. by Antonio Cerone and Stefan Gruner. IEEE Computer Society, 2008, pp. 323–332. DOI: 10.1109/SEFM.2008.11.

[84] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. "A static verification framework for message passing in Go using behavioural types". In: *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman. ACM, 2018, pp. 1137–1148. DOI: 10.1145/3180155.3180157.

[85] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. "From Communicating Machines to Graphical Choreographies". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programm. Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 221–232. DOI: 10.1145/2676726.2676964.

[86] Julien Lange and Nobuko Yoshida. "On the Undecidability of Asynchronous Session Subtyping". In: *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Javier Esparza and Andrzej S. Murawski. Vol. 10203. Lecture Notes in Computer Science. 2017, pp. 441–457. DOI: 10.1007/978-3-662-54458-7_26.

[87] Julien Lange and Nobuko Yoshida. "Verifying Asynchronous Interactions via Communicating Session Automata". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 97–117. DOI: 10.1007/978-3-030-25540-4_6.

[88] Elaine Li, Felix Stutz, and Thomas Wies. "Deciding Subtyping for Asynchronous Multiparty Sessions". In: *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 9 - April 11, 2024. Proceedings*. Ed. by Stephanie Weirich. Vol. 14576. Lecture Notes in Computer Science. Springer, 2024, pp. 176–205. DOI: 10.1007/978-3-031-57262-3_8.

[89] Elaine Li, Felix Stutz, Thomas Wies, and Damien Zufferey. "Complete Multiparty Session Type Projection with Automata". In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings,*

Part III. Ed. by Constantin Enea and Akash Lal. Vol. 13966. Lecture Notes in Computer Science. Springer, 2023, pp. 350–373. DOI: 10.1007/978-3-031-37709-9_17.

[90] Sam Lindley and J. Garrett Morris. "Embedding session types in Haskell". In: *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. Ed. by Geoffrey Mainland. ACM, 2016, pp. 133–145. DOI: 10.1145/2976002.2976018.

[91] Markus Lohrey. "Realizability of high-level message sequence charts: closing the gaps". In: *Theor. Comput. Sci.* 309.1-3 (2003), pp. 529–554. DOI: 10.1016/j.tcs.2003.08.002.

[92] P. Madhusudan. "Reasoning about Sequential and Branching Behaviours of Message Sequence Graphs". In: *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, July 8-12, 2001, Proceedings*. Ed. by Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen. Vol. 2076. Lecture Notes in Computer Science. Springer, 2001, pp. 809–820. DOI: 10.1007/3-540-48224-5_66.

[93] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. "Generalising Projection in Asynchronous Multiparty Session Types". In: *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference*. Ed. by Serge Haddad and Daniele Varacca. Vol. 203. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 35:1–35:24. DOI: 10.4230/LIPIcs.CONCUR.2021.35.

[94] Rupak Majumdar, Marcus Pirron, Nobuko Yoshida, and Damien Zufferey. "Motion Session Types for Robotic Interactions (Brave New Idea Paper)". In: *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*. Ed. by Alastair F. Donaldson. Vol. 134. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 28:1–28:27. DOI: 10.4230/LIPIcs.ECOOP.2019.28.

[95] Sjouke Mauw and Michel A. Reniers. "High-level message sequence charts". In: *SDL '97 Time for Testing, SDL, MSC and Trends - 8th International SDL Forum, Evry, France, 23-29 September 1997, Proceedings*. Ed. by Ana R. Cavalli and Amardeo Sarma. Elsevier, 1997, pp. 291–306.

[96] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999. ISBN: 978-0-521-65869-0.

[97] Arjan J. Mooij, Nicolae Goga, and Judi Romijn. "Non-local Choice and Beyond: Intricacies of MSC Choice Nodes". In: *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Maura Cerioli. Vol. 3442. Lecture Notes

in Computer Science. Springer, 2005, pp. 273–288. DOI: 10.1007/978-3-540-31984-9_21.

[98] Rémi Morin. "Recognizable Sets of Message Sequence Charts". In: *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings*. Ed. by Helmut Alt and Afonso Ferreira. Vol. 2285. Lecture Notes in Computer Science. Springer, 2002, pp. 523–534. DOI: 10.1007/3-540-45841-7_43.

[99] Henry Muccini. "Detecting Implied Scenarios Analyzing Non-local Branching Choices". In: *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. Ed. by Mauro Pezzè. Vol. 2621. Lecture Notes in Computer Science. Springer, 2003, pp. 372–386. DOI: 10.1007/3-540-36578-8_26.

[100] Anca Muscholl and Doron A. Peled. "Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces". In: *Mathematical Foundations of Computer Science 1999, 24th International Symposium, MFCS'99, Szklarska Poreba, Poland, September 6-10, 1999, Proceedings*. Ed. by Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki. Vol. 1672. Lecture Notes in Computer Science. Springer, 1999, pp. 81–91. DOI: 10.1007/3-540-48340-3_8.

[101] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. "A session type provider: compile-time API generation of distributed protocols with refinements in F#". In: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. Ed. by Christophe Dubach and Jingling Xue. ACM, 2018, pp. 128–138. DOI: 10.1145/3178372.3179495.

[102] Catuscia Palamidessi. "Comparing The Expressive Power Of The Synchronous And Asynchronous Pi-Calculi". In: *Math. Struct. Comput. Sci.* 13.5 (2003), pp. 685–719. DOI: 10.1017/S0960129503004043.

[103] Wuxu Peng and S. Purushothaman. "Analysis of a Class of Communicating Finite State Machines". In: *Acta Informatica* 29.6/7 (1992), pp. 499–522. DOI: 10.1007/BF01185558.

[104] C. A. Petri. "Fundamentals of a Theory of Asynchronous Information Flow". In: *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*. North-Holland, 1962, pp. 386–390.

[105] Emil L. Post. "A variant of a recursively unsolvable problem". In: *Bulletin of the American Mathematical Society* 52 (1946), pp. 264–268.

[106] *Prototype Implementation of Subset Projection for Multiparty Session Types*. https://gitlab.mpi-sws.org/fstutz/async-mpst-gen-choice/. (Visited on 04/15/2024).

[107] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. "Symbolic Message Sequence Charts". In: *ACM Trans. Softw. Eng. Methodol.* 21.2 (2012), 12:1–12:44. DOI: 10.1145/2089116.2089122.

[108] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. "A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain.* Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 24:1–24:31. DOI: 10.4230/LIPIcs.ECOOP.2017.24.

[109] Alceste Scalas and Nobuko Yoshida. "Less is more: multiparty session types revisited". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 30:1–30:29. DOI: 10.1145/3290343.

[110] Alceste Scalas and Nobuko Yoshida. "Lightweight Session Programming in Scala". In: *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy.* Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 21:1–21:28. DOI: 10.4230/LIPIcs.ECOOP.2016.21.

[111] Alceste Scalas and Nobuko Yoshida. *Mpstk: The Multiparty Session Types Toolkit.* 2018. URL: https://doi.org/10.1145/3291638.

[112] Michael Sipser. *Introduction to the theory of computation.* PWS Publishing Company, 1997. ISBN: 978-0-534-94728-6.

[113] "Spring and Hibernate Transaction in Java". In: https://www.uml-diagrams.org/examples/spring-hibernate-transaction-sequence-diagram-example.html. (Visited on 01/23/2021).

[114] Felix Stutz. *Artifact for "Complete Multiparty Session Type Projection with Automata".* Apr. 2024. DOI: 10.5281/zenodo.10972978.

[115] Felix Stutz. "Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts". In: *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States.* Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 32:1–32:31. DOI: 10.4230/LIPIcs.ECOOP.2023.32.

[117] Felix Stutz and Damien Zufferey. "Comparing Channel Restrictions of Communicating State Machines, High-level Message Sequence Charts, and Multiparty Session Types". In: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022.* Ed. by Pierre Ganty and Dario Della Monica. Vol. 370. EPTCS. 2022, pp. 194–212. DOI: 10.4204/EPTCS.370.13.

[118] Peter Thiemann. "Intrinsically-Typed Mechanized Semantics for Session Types". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 19:1–19:15. DOI: 10.1145/3354166.3354184.

[119] Peter Thiemann and Vasco T. Vasconcelos. "Context-free session types". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 462–475. DOI: 10.1145/2951913.2951926.

[120] Dawit Legesse Tirore, Jesper Bengtson, and Marco Carbone. "A Sound and Complete Projection for Global Types". In: *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*. Ed. by Adam Naumowicz and René Thiemann. Vol. 268. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 28:1–28:19. DOI: 10.4230/LIPIcs.ITP.2023.28.

[121] Bernardo Toninho, Luís Caires, and Frank Pfenning. "Dependent session types via intuitionistic linear type theory". In: *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*. Ed. by Peter Schneider-Kamp and Michael Hanus. ACM, 2011, pp. 161–172. DOI: 10.1145/2003476.2003499.

[122] Bernardo Toninho and Nobuko Yoshida. "Certifying data in multiparty session types". In: *J. Log. Algebraic Methods Program.* 90 (2017), pp. 61–83. DOI: 10.1016/j.jlamp.2016.11.005.

[123] Bernardo Toninho and Nobuko Yoshida. "Depending on Session-Typed Processes". In: *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Ed. by Christel Baier and Ugo Dal Lago. Vol. 10803. Lecture Notes in Computer Science. Springer, 2018, pp. 128–145. DOI: 10.1007/978-3-319-89366-2_7.

[124] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. "Context-Bounded Analysis of Concurrent Queue Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 299–314. DOI: 10.1007/978-3-540-78800-3_21.

[125] Tayssir Touili and Mohamed Faouzi Atig. "Verifying parallel programs with dynamic communication structures". In: *Theor. Comput. Sci.* 411.38-39 (2010), pp. 3460–3468. DOI: 10.1016/j.tcs.2010.05.028.

[126] Alan M. Turing. "On computable numbers, with an application to the Entscheidungsproblem". In: *Proc. London Math. Soc.* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230.

[127] International Telecommunication Union. *Z.120: Message Sequence Chart.* Tech. rep. International Telecommunication Union, 1996. URL: https://www.itu.int/rec/T-REC-Z.120.

[128] Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. "A multiparty session typing discipline for fault-tolerant event-driven distributed programming". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30. DOI: 10.1145/3485501.

[129] Michael Wehar. "On the complexity of intersection non-emptiness problems". PhD thesis. University of Buffalo, 2016.

[130] Nobuko Yoshida and Lorenzo Gheri. "A Very Gentle Introduction to Multiparty Session Types". In: *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings.* Ed. by Dang Van Hung and Meenakshi D'Souza. Vol. 11969. Lecture Notes in Computer Science. Springer, 2020, pp. 73–93. DOI: 10.1007/978-3-030-36987-3_5.

[131] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. "The Scribble Protocol Language". In: *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers.* Ed. by Martín Abadi and Alberto Lluch-Lafuente. Vol. 8358. Lecture Notes in Computer Science. Springer, 2013, pp. 22–41. DOI: 10.1007/978-3-319-05119-2_3.

Appendix A

Appendix for Chapter 5

A.1 Additional Material for Section 5.1

Lemma 5.4. Let G be a global type, $\mathtt{r}$ be a participant, and $\mathscr{C}(G, \mathtt{r})$ be its *subset construction*. If w is a trace of $\mathsf{GAut}(G)$, $w{\Downarrow_{\Gamma_\mathtt{r}}}$ is a trace of $\mathscr{C}(G, \mathtt{r})$. If u is a trace of $\mathscr{C}(G, \mathtt{r})$, there is a trace w of $\mathsf{GAut}(G)$ such that $w{\Downarrow_{\Gamma_\mathtt{r}}} = u$. Then, it holds that $\mathcal{L}(G){\Downarrow_{\Gamma_\mathtt{r}}} = \mathcal{L}(\mathscr{C}(G, \mathtt{r}))$.

Proof. All claims are rather straightforward from the definitions and constructions and the proofs exploit the connection to the projection by erasure. We still spell them out to familiarise the reader with these.

We prove the first claim first. By construction, for every run ρ in $\mathsf{GAut}(G)$, there exists a run ρ' in the projection by erasure $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$. Let ρ be the run for trace w in $\mathsf{GAut}(G)$. Then, ρ is also a run in $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$ with trace $w{\Downarrow_{\Gamma_\mathtt{p}}}$. Since $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$ might be non-deterministic, we apply the subset construction from Definition 5.3. For the reachable states, this is equivalent to the definition by Sipser [112, Thm. 1.39]. Thus, the constructed deterministic finite state machine can mimic any run (which is initial by definition) in $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$: for every run ρ' in $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$ with trace w', there is a run ρ'' in $\mathscr{C}(G, \mathtt{r})$ with trace w'.

For the second claim, we consider a trace u of $\mathscr{C}(G, \mathtt{r})$. Because of the subset construction, it holds that for every run ρ' in $\mathscr{C}(G, \mathtt{r})$ with trace w', there is a run ρ'' in the projection by erasure $\mathsf{GAut}(G){\downarrow_\mathtt{r}}$ with trace w'. By definition of the projection by erasure, a run ρ''' in $\mathsf{GAut}(G)$ exists with the same sequence of syntactic subterms as ρ'' and $\mathrm{trace}(\rho'''){\Downarrow_{\Gamma_\mathtt{p}}} = w'$.

From this, it easily follows that $\mathcal{L}(\mathscr{C}(G, \mathtt{r})) = \mathcal{L}(\mathsf{GAut}(G){\downarrow_\mathtt{r}})$ and, therefore, $\mathcal{L}(\mathscr{C}(G, \mathtt{r})) = \mathcal{L}(G){\Downarrow_{\Gamma_\mathtt{r}}}$. ☐

Lemma 5.5. For all global types G, it holds that $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}})$.

Proof. Given that $\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}}$ is deterministic, to prove language inclusion it suffices to prove the inclusion of the respective prefix sets:

$$\mathrm{pref}(\mathcal{L}(G)) \subseteq \mathrm{pref}(\mathcal{L}\{\!\{\mathscr{C}(G, \mathtt{p})\}\!\}_{\mathtt{p} \in \mathcal{P}})$$

We prove this via structural induction on w. The base case, $w = \varepsilon$, is trivial. For the inductive step, let $wx \in \text{pref}(\mathcal{L}(G))$. From the induction hypothesis, $w \in \text{pref}(\mathcal{L}\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$. It suffices to show that the transition labeled with x is enabled for the active participant in x. Let $(\vec{s},\xi)$ denote the $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ configuration reached on w. In the case that $x \in \Gamma_!$, let $x = \mathtt{p}\triangleright\mathtt{q}!\mathtt{m}$. The existence of an outgoing transition $\xrightarrow{\mathtt{p}\triangleright\mathtt{q}!\mathtt{m}}$ from $\vec{s}_\mathtt{p}$ follows from the fact that $\mathcal{L}(\mathscr{C}(G,\mathtt{p})) = \mathcal{L}(G)\Downarrow_{\Gamma_\mathtt{p}}$ (Lemma 5.4). The fact that $wx \in \text{pref}(\mathcal{L}\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$ follows immediately from this and the fact that send transitions in a CSM are always enabled. In the case that $x \in \Gamma_?$, let $x = \mathtt{q}\triangleleft\mathtt{p}?\mathtt{m}$. We obtain an outgoing transition $\xrightarrow{\mathtt{q}\triangleleft\mathtt{p}?\mathtt{m}}$ from $\vec{s}_\mathtt{p}$ analogously. We additionally need to show that $\xi(\mathtt{q},\mathtt{p})$ contains m at the head. This follows from Lemma 2.5 and the induction hypothesis. This concludes our proof of prefix set inclusion.

Let w be a word in $\mathcal{L}(G)$. Let $(\vec{s},\xi)$ denote the $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ configuration reached on w. In the case that w is finite, all states in $\vec{s}$ are final from Lemma 5.4 and all channels in ξ are empty from the fact that all send events in w contain matching receive events. In the case that w is infinite, we show that w has an infinite run in $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ using König's Lemma. We construct an infinite graph $\mathcal{G}_w(V,E)$ with $V := \{v_\rho \mid \text{trace}(\rho) \leq w\}$ and

$$E := \{(v_{\rho_1}, v_{\rho_2}) \mid \exists\, x \in \Gamma.\ \text{trace}(\rho_2) = \text{trace}(\rho_1)\cdot x\} \enspace .$$

Because $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ is deterministic, $\mathcal{G}_w$ is a tree rooted at v_ε, the vertex corresponding to the empty run. By König's Lemma, every infinite tree contains either a vertex of infinite degree or an infinite path. Because $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ consists of a finite number of communicating state machines, the last configuration of any run has a finite number of next configurations, and $\mathcal{G}_w$ is finitely branching. Therefore, there must exist an infinite path in $\mathcal{G}_w$ representing an infinite run for w, and thus $w \in \mathcal{L}(\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}})$. □

A.2 Additional Material for Section 5.3

Corollary A.1 (Intersection sets are invariant under $\sim$)**.** Let G be a global type. Let $w, w' \in \Gamma^*$ and $w \sim w'$. Then, $I(w) = I(w')$.

Proof. It follows immediately from $w \sim w'$ that

$$\forall \mathtt{p} \in \mathcal{P}.\, w\Downarrow_{\Gamma_\mathtt{p}} = w'\Downarrow_{\Gamma_\mathtt{p}}$$

By the definition of I,

$$\forall \rho.\, \rho \in I(w) \Leftrightarrow \forall \mathtt{p} \in \mathcal{P}.\, w\Downarrow_{\Gamma_\mathtt{p}} \leq \text{trace}(\rho)\Downarrow_{\Gamma_\mathtt{p}}$$

Let ρ be a run in $\mathsf{GAut}(G)$. Then,

$$\begin{aligned}\rho \in I(w) &\Leftrightarrow \forall \mathtt{p} \in \mathcal{P}.\, w{\Downarrow}_{\Gamma_\mathtt{p}} \leq \mathrm{trace}(\rho){\Downarrow}_{\Gamma_\mathtt{p}} \\ &\Leftrightarrow \forall \mathtt{p} \in \mathcal{P}.\, w'{\Downarrow}_{\Gamma_\mathtt{p}} \leq \mathrm{trace}(\rho){\Downarrow}_{\Gamma_\mathtt{p}} \\ &\Leftrightarrow \rho \in I(w')\end{aligned}$$

□

The definition of available messages immeditately yields the following properties.

Proposition A.2 (Structural properties of $\mathrm{msgs}^{\mathcal{B}}_{(G\ldots)}$)**.** Let G be a global type, $\mathcal{B} \subseteq \mathcal{P}$ be a set of participants, G' be a syntactic subterm of G, and $\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \in \mathrm{msgs}^{\mathcal{B}}_{(G\ldots)}$. Then, it holds that:

(1) $\mathrm{msgs}^{\mathcal{B}}_{(G'\ldots)}$ does not contain any events whose sender is blocked: $\forall\, \mathtt{p} \in \mathcal{B}, \mathtt{q}\triangleleft\mathtt{r}?_ \in \mathrm{msgs}^{\mathcal{B}}_{(G'\ldots)} \,.\, \mathtt{r} \neq \mathtt{p}$

(2) There exists a run suffix β such that:
   i. $G' \cdot \beta$ is the suffix of a maximal run in $\mathsf{GAut}(G)$,
   ii. $\xrightarrow{\mathtt{q}\to\mathtt{p}:\mathsf{m}}$ occurs in β, and
   iii. $\mathcal{B}$ monotonically increases during the computation of $\mathrm{msgs}^{\mathcal{B}}_{(G\ldots)}$

Lemma A.3 (Correspondence between unique splittings and local states)**.** Let G be a global type, and $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be the subset construction for each participant. Let w be a trace of $\{\!\{\mathscr{C}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$, and $(\vec{s},\xi)$ be the CSM configuration reached on w. Let $\mathtt{p}$ be a participant. Then, it holds that:

$$\bigcup_{\rho\in \mathrm{R}^G_\mathtt{p}(w)} \{G' \mid \alpha \cdot G' \xrightarrow{l} G'' \cdot \beta \text{ is the unique splitting of } \rho \text{ matching } w\} \subseteq \vec{s}_\mathtt{p}$$

Proof. Let ρ be a run in $\mathrm{R}^G_\mathtt{p}(w)$, and let $\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ be its unique splitting for $\mathtt{p}$ matching w. It follows from the definition of unique splitting that $\mathrm{trace}(\alpha \cdot G''){\Downarrow}_{\Gamma_\mathtt{p}} = w{\Downarrow}_{\Gamma_\mathtt{p}}$. From the subset construction, there exists a run $s_1, \ldots, s_n$ in $\mathscr{C}(G,\mathtt{p})$ such that $s_1 = s_{0,\mathtt{p}}$ and $s_1 \xrightarrow{w{\Downarrow}_{\Gamma_\mathtt{p}}}{}^* s_n$. By the definition of $Q_\mathtt{p}$, we know that s_n contains all global syntactic subterms in G that are reachable via $q_{0,G} \xrightarrow{w{\Downarrow}_{\Gamma_\mathtt{p}}} \xrightarrow{\varepsilon}{}^*$ in $\mathsf{GAut}(G)_{\downarrow}$, of which G' is one. Hence, $G' \in s_n$. By assumption, $\mathscr{C}(G,\mathtt{p})$ reached state $\vec{s}_\mathtt{p}$ on $w{\Downarrow}_{\Gamma_\mathtt{p}}$. Because subset constructions are deterministic, it follows that $s_n = \vec{s}_\mathtt{p}$. We conclude that $G' \in \vec{s}_\mathtt{p}$. □

Lemma A.4 (No send transitions from final states in subset projection)**.** Let G be a global type, and $\mathscr{P}(G,\mathtt{p})$ be the subset projection for $\mathtt{p}$. Let $s \in F_\mathtt{p}$, and $s \xrightarrow{x} s' \in \delta_\mathtt{p}$. Then, $x \in \Gamma_{\mathtt{p},?}$.

Proof. Assume by contradiction that $x \in \Gamma_{p,!}$. We instantiate Send Validity with $s \xrightarrow{x} s'$ to obtain:

$$x \in \Gamma_{p,!} \implies \text{tr-orig}(s \xrightarrow{x} s') = s$$

By the definition of tr-orig(-), for every syntactic subterm $G' \in \text{tr-orig}(s \xrightarrow{x} s')$:

$$\exists G'' \in s'. G' \xrightarrow{x}{}^{*} G'' \in \delta\restriction_{\Gamma_p}$$

Because s is a final state in $\mathscr{P}(G, p)$, by definition it must contain a syntactic subterm that is a final state in $\mathsf{GAut}(G)\restriction_p$. Because $\mathsf{GAut}(G)\restriction_p$ and $\mathsf{GAut}(G)$ share the same set of final states (Definition 5.2), s must contain a syntactic subterm that is a final state in $\mathsf{GAut}(G)$. Let G_0 denote this final state. By the structure of $\mathsf{GAut}(G)$, there exists no outgoing transition from G_0. Therefore, G'' does not exist. We reach a contradiction. □

Definition A.5 (G-complete words of $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$)**.** Let G be a global type, $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$ be a CSM, and w be a trace of $\{\!\{A_p\}\!\}_{p\in\mathcal{P}}$. We say w is *G-complete* if for all participants p and for all runs $\rho \in \bigcap_{p\in\mathcal{P}} \mathrm{R}^G_p(w)$, it holds that $w\Downarrow_{\Gamma_p} = \big(\text{trace}(\rho)\big)\Downarrow_{\Gamma_p}$.

Lemma 5.18. Let G be a global type and $\{\!\{\mathscr{P}(G, p)\}\!\}_{p\in\mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\!\{\mathscr{P}(G, p)\}\!\}_{p\in\mathcal{P}}$ such that $x \in \Gamma_?$. Then, $I(w) = I(wx)$.

Proof. Let $x = p \triangleleft q?m$. Because wx is a trace of $\{\!\{\mathscr{P}(G, p)\}\!\}_{p\in\mathcal{P}}$, there exists a run $(\vec{s}_0, \xi_0) \xrightarrow{w}{}^{*} (\vec{s}, \xi) \xrightarrow{x} (\vec{s}', \xi')$ such that m is at the head of channel $\xi(p, q)$.

We assume that $I(w)$ is non-empty; if $I(w)$ is empty then $I(wx)$ is trivially empty. To show $I(w) = I(wx)$, it suffices to show the following claim.

Claim I: It holds that $\mathrm{R}^G_p(w) \cap \mathrm{R}^G_q(w) = \mathrm{R}^G_p(wx) \cap \mathrm{R}^G_q(wx)$.

We first show Claim I's sufficiency for $I(w) = I(wx)$: By definition, $I(w) = \bigcap_{r\in\mathcal{P}} \mathrm{R}^G_r(w) \subseteq \mathrm{R}^G_q(w) \cap \mathrm{R}^G_p(w)$. With *Claim I*, it holds that $I(w) \subseteq \mathrm{R}^G_q(wx) \cap \mathrm{R}^G_p(wx)$. From this, it follows that $I(w) \subseteq \mathrm{R}^G_p(wx)$ (H1). Since p is the active participant for the receive event x, i.e. $x \in \Gamma_p$, it holds for any $r \neq p$, that $(wx)\Downarrow_{\Gamma_r} = w\Downarrow_{\Gamma_r}$ and $\mathrm{R}^G_r(w) = \mathrm{R}^G_r(wx)$ (H2). Again, by definition of $\mathrm{R}^G_-(\text{-})$, we have $\mathrm{R}^G_p(wx) \subseteq \mathrm{R}^G_p(w)$ (H3). We apply the observations to $I(wx)$:

$$\begin{aligned}
I(wx) &= \bigcap_{r\in\mathcal{P}} \mathrm{R}^G_r(wx) \\
&\overset{(H2)}{=} \mathrm{R}^G_p(wx) \cap \bigcap_{r\in\mathcal{P}\wedge r\neq p} \mathrm{R}^G_r(w) \\
&\overset{(H3)}{=} \mathrm{R}^G_p(wx) \cap \mathrm{R}^G_p(w) \cap \bigcap_{r\in\mathcal{P}\wedge r\neq p} \mathrm{R}^G_r(w) \\
&= \mathrm{R}^G_p(wx) \cap I(w) \\
&\overset{(H1)}{=} I(w) \enspace .
\end{aligned}$$

This concludes our reasoning that *Claim I* is sufficient.

Proof of Claim I. We instantiate (H2) for participant q, which yields $\mathrm{R}^G_{\mathtt{q}}(wx) = \mathrm{R}^G_{\mathtt{q}}(w)$. The proof of Claim I therefore amounts to showing:

$$\mathrm{R}^G_{\mathtt{p}}(w) \cap \mathrm{R}^G_{\mathtt{q}}(w) = \mathrm{R}^G_{\mathtt{p}}(wx) \cap \mathrm{R}^G_{\mathtt{q}}(w) \enspace .$$

The right direction, i.e. $\mathrm{R}^G_{\mathtt{p}}(wx) \subseteq \mathrm{R}^G_{\mathtt{p}}(w)$, follows from (H3). For the left direction, i.e. $\mathrm{R}^G_{\mathtt{p}}(w) \cap \mathrm{R}^G_{\mathtt{q}}(w) \subseteq \mathrm{R}^G_{\mathtt{p}}(wx)$, assume by contradiction that there exists a run ρ_0 such that

$$\rho_0 \in \mathrm{R}^G_{\mathtt{p}}(w) \ \wedge \ \rho_0 \in \mathrm{R}^G_{\mathtt{q}}(w) \ \wedge \ \rho_0 \notin \mathrm{R}^G_{\mathtt{p}}(wx) \enspace .$$

Let ρ' be a run in $\mathrm{R}^G_{\mathtt{p}}(w) \setminus \mathrm{R}^G_{\mathtt{p}}(wx)$. Let $\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \cdot \beta'$ be the unique splitting of ρ' for p matching w.

Let $\rho'_{\mathtt{p}}$ denote the largest consistent prefix of ρ' for p; it is clear that $\rho'_{\mathtt{p}} = \alpha' \cdot G'_{pre}$. Formally,

$$\rho'_{\mathtt{p}} = \max\{\rho \mid \rho \leq \rho' \ \wedge \ \big(\mathrm{trace}(\rho)\big)\Downarrow_{\Gamma_{\mathtt{p}}} \leq w\Downarrow_{\Gamma_{\mathtt{p}}}\} \enspace .$$

Let $\rho'_{\mathtt{q}}$ be defined analogously.

We claim that q is ahead of p in ρ', i.e. $\rho'_{\mathtt{p}} < \rho'_{\mathtt{q}}$. Intuitively, this claim follows from the half-duplex property of CSMs and the fact that q is the sender. Formally, Lemma 2.5 implies $\xi(\mathtt{q},\mathtt{p}) = u$ where $\mathcal{V}(w\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}) \cdot u$. Because $\xi(\mathtt{q},\mathtt{p})$ contains at least m by assumption, $|\mathcal{V}(w\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_})| > |\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_})|$. Because $\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}) < \mathcal{V}(w\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_})$ and traces of CSMs are FIFO-compliant (Lemma 2.5), it holds that $\rho'_{\mathtt{q}}$ contains all $|\mathcal{V}(w\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_})|$ transition labels of the form $\mathtt{q}\to\mathtt{p}:_$ that are contained in $\rho'_{\mathtt{q}}$, plus at least one more of the form $\mathtt{q} \to \mathtt{p} : m$. Because both $\rho'_{\mathtt{p}}$ and $\rho'_{\mathtt{q}}$ are prefixes of ρ', it must be the case that $\rho'_{\mathtt{p}} < \rho'_{\mathtt{q}}$.

End Proof of Claim I.

By assumption, $\rho' \notin \mathrm{R}^G_{\mathtt{p}}(wx)$ and therefore $l' \neq \mathtt{q} \to \mathtt{p} : m$. By the definition of unique splittings, p must be the active participant in l'; by Corollary 5.13, p must be the receiving participant in l'. In other words, l' must be of the form $\mathtt{r} \to \mathtt{p} : \mathsf{m}'$, where either $\mathtt{r} \neq \mathtt{q}$ or $\mathsf{m}' \neq \mathsf{m}$.

Case: $\mathtt{r} = \mathtt{q}$ and $\mathsf{m}' \neq \mathsf{m}$.

We discharge this case by showing a contradiction to the assumption that m is at the head of the channel between q and p.

Because $\alpha' \cdot G'_{pre} \leq \rho'_{\mathtt{p}}$ and $\rho'_{\mathtt{p}} < \rho'_{\mathtt{q}}$ from the claim above, it must be the case that $\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \leq \rho'_{\mathtt{q}}$ and $\mathtt{q}\triangleright\mathtt{p}!\mathsf{m}'$ is in $w\Downarrow_{\Gamma_{\mathtt{q}}}$. From Lemma 2.5, it follows that $\mathcal{V}(w\Downarrow_{\mathtt{q}\triangleright\mathtt{p}!_}) = \mathcal{V}(w\Downarrow_{\mathtt{p}\triangleleft\mathtt{q}?_}).\mathsf{m}'.u'$ and $\xi(\mathtt{q},\mathtt{p}) = \mathsf{m}'.u'$, i.e. m' is at the head of the channel between q and p. This contradicts the assumption that m is at the head of $\xi(\mathtt{p},\mathtt{q})$.

Case: $\mathtt{r} \neq \mathtt{q}$.

We discharge this case by showing a contradiction to Receive Validity. We instantiate Receive Validity with $\vec{s}_{\mathtt{p}} \xrightarrow{x} \vec{s}'_{\mathtt{p}}$ to obtain

$$\forall\, \vec{s}_{\mathtt{p}} \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}_2?\mathsf{m}_2} s_2 \in \delta_{\mathtt{p}}.\, \mathtt{q} \neq \mathtt{q}_2 \implies \forall\, G_2 \in \text{tr-dest}(\vec{s}_{\mathtt{p}} \xrightarrow{\mathtt{p}\triangleleft\mathtt{q}_2?\mathsf{m}_2} s_2).\, \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \notin \text{msgs}^{\mathtt{p}}_{(G_2\ldots)} \quad .$$

We prove the negation, stated as follows:

$$\mathtt{q} \neq \mathtt{r} \wedge \exists\, s_2 \in Q_{\mathtt{p}}, G_2 \in \text{tr-dest}(\vec{s}_{\mathtt{p}} \xrightarrow{\mathtt{p}\triangleleft\mathtt{r}?\mathsf{m}'} s_2).\, \mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \in \text{msgs}^{\mathtt{p}}_{(G_2\ldots)} \quad .$$

The left conjunct follows immediately. From the existence of ρ' and Lemma 5.4, there exists an s_2 such that $\vec{s}_{\mathtt{p}} \xrightarrow{\mathtt{p}\triangleleft\mathtt{r}?\mathsf{m}'} s_2 \in \delta_{\mathtt{p}}$. The fact that $G'_{post} \in \text{tr-dest}(\vec{s}_{\mathtt{p}} \xrightarrow{\mathtt{p}\triangleleft\mathtt{r}?\mathsf{m}'} s_2)$ is trivial from the unique splitting of ρ' for $\mathtt{p}$ matching w:

$$\rho' = \alpha' \cdot G'_{pre} \xrightarrow{\mathtt{r}\to\mathtt{p}:\mathsf{m}'} G'_{post} \cdot \beta' \quad .$$

Therefore, all that remains is to show that $\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \in \text{msgs}^{\mathtt{p}}_{(G'_{post}\ldots)}$. Because $\rho' \in \mathrm{R}^{G}_{\mathtt{q}}(w)$ and $\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \leq \rho'_{\mathtt{q}}$, where $\mathtt{q}$ is not the active participant in l', there must exist a transition labelled $\mathtt{q} \to \mathtt{p} : m$ that occurs in the suffix $G'_{post} \cdot \beta'$ of ρ'. Let $G_0 \xrightarrow{\mathtt{q}\to\mathtt{p}:m} G'_0$ be the earliest occurrence of such a transition in the suffix, then:

$$\rho'_{\mathtt{q}} = \alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \ldots G_0 \xrightarrow{\mathtt{q}\to\mathtt{p}:m} G'_0 \ldots \quad .$$

Note that G_0 must be a syntactic subterm of G'_{post}. In order for $\mathtt{p}\triangleleft\mathtt{q}?\mathsf{m} \in \text{msgs}^{\mathtt{p}}_{(G'_{post}\ldots)}$ to hold, it suffices to show that $\mathtt{q} \notin \mathcal{B}$ in the recursive call to $\text{msgs}^{\mathcal{B}}_{(G_0\ldots)}$.

We argue this from the definition of msgs^{-}_{-} and the fact that $\rho'_{\mathtt{p}} = \alpha' \cdot G'_{pre}$. Suppose for the sake of contradiction that $\mathtt{q} \in \mathcal{B}$. Because msgs^{-}_{-} only adds receivers of already blocked senders to $\mathcal{B}$ and $\text{msgs}^{\mathtt{p}}_{(G'_{post}\ldots)}$ starts with $\mathcal{B} = \{\mathtt{p}\}$, there must exist a chain of message exchanges $\mathtt{s}_{i+1} \to \mathtt{s}_i : \mathsf{m}_i$ in G'_{post} with $1 \leq i < n$, $\mathtt{p} = \mathtt{s}_n$, and $\mathtt{q} = \mathtt{s}_1$. That is, $G'_{post} \cdot \beta'$ must be of the form

$$G'_{post} \ldots G_{n-1} \xrightarrow{\mathtt{p}\to\mathtt{s}_{n-1}:\mathsf{m}_{n-1}} G'_{n-1} \ldots G_1 \xrightarrow{\mathtt{s}_2\to\mathtt{q}:\mathsf{m}_1} G'_1 \ldots G_0 \xrightarrow{\mathtt{q}\to\mathtt{p}:m} G'_0 \ldots \quad .$$

Let $\mathsf{m}_0 = \mathsf{m}$ and $\mathtt{s}_0 = \mathtt{p}$. We show by induction over i that for all $i \in [1, n]$:

$$\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \ldots G_i \xrightarrow{\mathtt{s}_i\to\mathtt{s}_{i-1}:\mathsf{m}_{i-1}} G'_i \;\leq\; \rho'_{\mathtt{s}_i} \quad .$$

We then obtain the desired contradiction with the fact that $\rho'_{\mathtt{s}_n} = \rho'_{\mathtt{p}} = \alpha' \cdot G'_{pre}$.

The base case of the induction follows immediately from the construction. For the induction step, assume that

$$\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \ldots G_i \xrightarrow{\mathtt{s}_i\to\mathtt{s}_{i-1}:\mathsf{m}_{i-1}} G'_i \;\leq\; \rho'_{\mathtt{s}_i} \quad .$$

From the definition of $\rho'_{\mathtt{s}_i}$ and the fact that $\mathtt{s}_i$ is the active participant in $\mathtt{s}_i \triangleleft \mathtt{s}_{i+1}?\mathsf{m}_i$, it follows that $\mathtt{s}_i \triangleleft \mathtt{s}_{i+1}?\mathsf{m}_i \in w$. Hence, we must also have $\mathtt{s}_{i+1} \triangleright \mathtt{s}_i!\mathsf{m}_i \in w$. Since $\mathtt{s}_{i+1}$ is the active participant in $\mathtt{s}_{i+1} \triangleright \mathtt{s}_i!\mathsf{m}_i$, we can conclude

$$\alpha' \cdot G'_{pre} \xrightarrow{l'} G'_{post} \ldots G_i \xrightarrow{\mathtt{s}_{i+1} \to \mathtt{s}_i : \mathsf{m}_i} G'_{i+1} \leq \rho'_{\mathtt{s}_{i+1}} .$$

□

Lemma 5.20. Let G be a global type and $\{\!\{\mathscr{P}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ be the subset projection. Let wx be a trace of $\{\!\{\mathscr{P}(G,\mathtt{p})\}\!\}_{\mathtt{p}\in\mathcal{P}}$ such that $x \in \Gamma_! \cap \Gamma_{procA}$ for some $\mathtt{p} \in \mathcal{P}$. Let ρ be a run in $I(w)$, and $\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ be the unique splitting of ρ for $\mathtt{p}$ with respect to w. Then, there exists a run ρ' in $I(wx)$ such that $\alpha \cdot G' \leq \rho'$.

Proof. Let $x = \mathtt{p} \triangleright \mathtt{q}!\mathsf{m}$. We prove the claim by induction on the length of w.
Base Case: $w = \varepsilon$. By definition, $I(\varepsilon)$ contains all maximal runs in $\mathsf{GAut}(G)$, and the unique splitting prefix of any run $\rho \in I(\varepsilon)$ for $\mathtt{p}$ with respect to ε is ε. Because ε is a prefix of any run, we need only show the non-emptiness of $I(x)$. By Lemma 5.4, $\mathcal{L}(G){\Downarrow}_{\Gamma_\mathtt{p}} = \mathcal{L}(\mathscr{P}(G,\mathtt{p}))$. Because x is the prefix of a word in $\mathcal{L}(\mathscr{P}(G,\mathtt{p}))$, there exists $w' \in \mathcal{L}(G)$ such that $x \leq w'{\Downarrow}_{\Gamma_\mathtt{p}}$. By the semantics of $\mathcal{L}(G)$, there exists a run $\rho' \in \mathsf{GAut}(G)$ such that x is the first symbol in $\mathrm{trace}(\rho'){\Downarrow}_{\Gamma_\mathtt{p}}$, and therefore $\rho' \in I(x)$.
Induction Step: let wx be an extension of w by $x \in \Gamma_!$.

Let ρ be a run in $I(w)$, and let $\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta$ be the unique splitting of ρ for participant $\mathtt{p}$ with respect to w. To re-establish the induction hypothesis, we need to show the existence of a run $\bar{\rho}$ in $I(wx)$ such that $\alpha \cdot G' \leq \bar{\rho}$. Since $\mathtt{p}$ is the active participant in x, it holds for any $\mathtt{r} \neq \mathtt{p}$ that $\mathrm{R}^G_\mathtt{r}(w) = \mathrm{R}^G_\mathtt{r}(wx)$. Therefore, to prove the existential claim, it suffices to construct a run $\bar{\rho}$ that satisfies:

(1) $\bar{\rho} \in \mathrm{R}^G_\mathtt{p}(wx)$,
(2) $\bar{\rho} \in I(w)$, and
(3) $\alpha \cdot G' \leq \bar{\rho}$.

In the case that $l{\Downarrow}_{\Gamma_\mathtt{p}} = x$, we are done: (2) and (3) hold by construction, and (1) holds by the definition of possible run sets. In the case that $l{\Downarrow}_{\Gamma_\mathtt{p}} \neq x$, we show the existence of a transition label and state $\xrightarrow{\bar{l}} \bar{G}''$, and a maximal suffix $\bar{\beta}$ such that $\alpha \cdot G' \xrightarrow{\bar{l}} \bar{G}'' \cdot \bar{\beta}$ satisfies all three conditions.

Let $(\vec{s}_w, \xi_w)$ denote the CSM configuration reached on w: $(\vec{s}_0, \xi_0) \xrightarrow{w}{}^{*} (\vec{s}_w, \xi_w)$. Send Validity states that every transition in $\vec{s}_{w,\mathtt{p}}$ originates in all global states in $\vec{s}_{w,\mathtt{p}}$. By assumption, $\mathtt{p} \triangleright \mathtt{q}!\mathsf{m}$ is a transition in $\vec{s}_{w,\mathtt{p}}$. By Proposition A.3, $\rho \in I \subseteq \mathrm{R}^G_\mathtt{p}(w)$, and therefore $G' \in \vec{s}_{w,\mathtt{p}}$. Therefore, Send Validity gives the existence of some $\bar{G}'' \in Q_{\mathsf{GAut}(G)}$ such that $G' \xrightarrow{\mathtt{p} \to \mathtt{q}:m} \bar{G}'' \in \delta_{\mathsf{GAut}(G)}$. Because $\alpha \cdot G'$ is a run in $\mathsf{GAut}(G)$ and $G' \xrightarrow{\mathtt{p} \to \mathtt{q}:m} \bar{G}''$ is a transition in $\mathsf{GAut}(G)$, $\alpha \cdot G' \xrightarrow{\mathtt{p} \to \mathtt{q}:m} \bar{G}''$ is a run in $\mathsf{GAut}(G)$.

The construction thus far satisfies (1) and (3) regardless of our choice of maximal suffix: for all choices of $\bar{\beta}$ such that $\alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}$ is a maximal run, $wx{\Downarrow_{\Gamma_\mathtt{p}}} \leq \mathrm{trace}(\alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}){\Downarrow_{\Gamma_\mathtt{p}}}$ and $\alpha \cdot G' \leq \alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}$.

Property (2), however, requires that the projection of w onto each participant is consistent with $\bar{\rho}$, and this cannot be ensured by the prefix alone.

We construct the remainder of $\bar{\rho}$ by picking an arbitrary maximal suffix to form a candidate run, and iteratively performing suffix replacements on the candidate run until it lands in I. Let $\bar{\beta}$ be a run suffix such that $\alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}$ is a maximal run in $\mathsf{GAut}(G)$. Let ρ_c denote our candidate run $\alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}$. If $\rho_c \in I$, we are done. Otherwise, $\rho_c \notin I$ and there exists a non-empty set of processes $\mathcal{S} \subseteq \mathcal{P}$ such that for each $\mathtt{r} \in \mathcal{S}$,

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \not\leq \mathrm{trace}(\rho_c){\Downarrow_{\Gamma_\mathtt{r}}} \ . \tag{A.1}$$

By the fact that $\rho \in I$,

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \leq \mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathtt{r}}} \ . \tag{A.2}$$

We can rewrite (A.1) and (A.2) above as:

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \not\leq \mathrm{trace}(\alpha \cdot G' \xrightarrow{\mathtt{p}\to\mathtt{q}:m} \bar{G}'' \cdot \bar{\beta}){\Downarrow_{\Gamma_\mathtt{r}}} \tag{A.3}$$

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \leq \mathrm{trace}(\alpha \cdot G' \xrightarrow{l} G'' \cdot \beta){\Downarrow_{\Gamma_\mathtt{r}}} \ . \tag{A.4}$$

By the definition of unique splitting, $\mathtt{p}$ is the active participant in l. By Lemma 5.4, $\mathcal{L}(G){\Downarrow_{\Gamma_\mathtt{p}}} = \mathcal{L}(\mathscr{P}(G,\mathtt{p}))$, and because $\mathrm{trace}(\rho) \in \mathcal{L}(G)$, it holds that $\mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathtt{p}}} \in \mathcal{L}(G){\Downarrow_{\Gamma_\mathtt{p}}}$, and $\mathrm{trace}(\rho){\Downarrow_{\Gamma_\mathtt{p}}} \in \mathcal{L}(\mathscr{P}(G,\mathtt{p}))$. By assumption, $\mathscr{P}(G,\mathtt{p})$ is in state $\vec{s}_{w,\mathtt{p}}$ upon consuming $w{\Downarrow_{\Gamma_\mathtt{p}}}$. Then, there must exist an outgoing transition from $\vec{s}_{w,\mathtt{p}}$ labelled with $l{\Downarrow_{\Gamma_\mathtt{p}}}$. By Corollary 5.13, all outgoing transitions from $\vec{s}_{w,\mathtt{p}}$ must be send actions. Therefore, l must be of the form $\mathtt{p}\to\mathtt{q}':\mathsf{m}'$. By assumption, $\mathtt{q}' \neq \mathtt{q}$ or $\mathsf{m}' \neq \mathsf{m}$.

We can further rewrite (A.3) and (A.4) to make explicit their shared prefix:

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \not\leq (\mathrm{trace}(\alpha \cdot G').\, \mathtt{p}\triangleright\mathtt{q}!\mathsf{m}.\, \mathtt{q}\triangleleft\mathtt{p}?\mathsf{m}.\ \mathrm{trace}(\bar{\beta})){\Downarrow_{\Gamma_\mathtt{r}}} \tag{A.5}$$

$$w{\Downarrow_{\Gamma_\mathtt{r}}} \leq (\mathrm{trace}(\alpha \cdot G').\, \mathtt{p}\triangleright\mathtt{q}'!\mathsf{m}'.\, \mathtt{q}'\triangleleft\mathtt{p}?\mathsf{m}'.\ \mathrm{trace}(\beta)){\Downarrow_{\Gamma_\mathtt{r}}} \tag{A.6}$$

It is clear that in order for both (A.5) and (A.6) to hold, it must be the case that $\mathrm{trace}(\alpha \cdot G'){\Downarrow_{\Gamma_\mathtt{r}}} \leq w{\Downarrow_{\Gamma_\mathtt{r}}}$.

We formalise the point of disagreement between $w{\Downarrow_{\Gamma_\mathtt{r}}}$ and ρ_c using an index $i_\mathtt{r}$ representing the position of the first disagreeing transition label in $\mathrm{trace}(\rho_c)$:

$$i_\mathtt{r} := \max\{i \mid \mathrm{trace}(\rho_c[0..i-1]){\Downarrow_{\Gamma_\mathtt{r}}} \leq w{\Downarrow_{\Gamma_\mathtt{r}}}\} \ .$$

Then, $\mathrm{trace}(\rho_c[i_{\mathtt{r}}])\Downarrow_{\Gamma_{\mathtt{r}}} \neq \varepsilon$ and from (A.5) and (A.6) we know that

$$i_{\mathtt{r}} > 2 \cdot |\mathrm{trace}(\alpha \cdot G')| \ .$$

We identify the participant in $\mathcal{S}$ with the *earliest disagreement* in ρ_c: let $\bar{\mathtt{r}}$ be the participant with the smallest $i_{\bar{\mathtt{r}}}$ in $\mathcal{S}$. Let $y_{\bar{\mathtt{r}}}$ denote $\mathrm{trace}(\rho_c[i_{\bar{\mathtt{r}}}])\Downarrow_{\Gamma_{\bar{\mathtt{r}}}}$.

Claim I: $y_{\bar{\mathtt{r}}}$ must be a send event.

Proof of Claim I. Assume by contradiction that $y_{\bar{\mathtt{r}}}$ is a receive event. We identify the symbol in w that disagrees with $y_{\bar{\mathtt{r}}}$: let w' be the largest prefix of w such that $w'\Downarrow_{\Gamma_{\bar{\mathtt{r}}}} \leq \mathrm{trace}(\rho_c)$. By definition, $w'\Downarrow_{\Gamma_{\bar{\mathtt{r}}}} = \mathrm{trace}(\rho_c[0..i_{\bar{\mathtt{r}}} - 1])\Downarrow_{\Gamma_{\bar{\mathtt{r}}}}$. Let z be the next symbol following w' in w; then $z \in \Gamma_{\bar{\mathtt{r}}}$ and $z \neq y_{\bar{\mathtt{r}}}$. Furthermore, by Corollary 5.13, we have that $z \in \Gamma_?$.

By assumption, $w'z \not\leq \mathrm{trace}(\rho_c[0..i_{\bar{\mathtt{r}}}])$. Therefore, any run that begins with $\rho_c[0..i_{\bar{\mathtt{r}}}]$ cannot be contained in $\mathrm{R}^G_{\bar{\mathtt{r}}}(w'z)$, or consequently in $I(w'z)$. We show however, that $I(w'z)$ must contain some runs that begin with $\rho_c[0..i_{\bar{\mathtt{r}}}]$. With Lemma 5.18 for traces w' and $w'z$, we obtain that $I(w') = I(w'z)$. Therefore, it suffices to show that $I(w')$ contains runs that begin with $\rho_c[0..i_{\bar{\mathtt{r}}}]$.

Claim II: For all $w'' \leq w'$, $I(w'')$ contains runs that begin with $\rho_c[0..i_{\bar{\mathtt{r}}}]$.

Proof of Claim II. We prove the claim via induction on w'.

The base case is trivial from the fact that $I(\varepsilon)$ contains all maximal runs.

For the inductive step, let $w''y \leq w'$.

In case that $y \in \Gamma_?$, we have $I(w''y) = I(w'')$ from Lemma 5.18 and the witness from $I(w'')$ can be reused.

In case that $y \in \Gamma_!$, let $\mathtt{s}$ be the active participant of y and let ρ' be a run in $I(w'')$ beginning with $\rho_c[0..i_{\bar{\mathtt{r}}}]$ given by the inner induction hypothesis. Let $\alpha' \cdot G'' \xrightarrow{l'} G''' \cdot \beta'$ be the unique splitting of ρ' for $\mathtt{s}$ with respect to w''. If $l'\Downarrow_{\Gamma_{\mathtt{s}}} = y$, then ρ' can be used as the witness. Otherwise, $l'\Downarrow_{\Gamma_{\mathtt{s}}} \neq y$, and $\rho' \notin \mathrm{R}^G_{\mathtt{s}}(w''y)$.

The outer induction hypothesis holds for all prefixes of w: we instantiate it with w'' and y to obtain:

$$\exists\, \rho'' \in I(w''y).\, \alpha' \cdot G'' \leq \rho'' \ .$$

Let $i_{\mathtt{s}}$ be defined as before; it follows that $\rho'[i_{\mathtt{s}}] = G''$. It must be the case that $i_{\mathtt{s}} > i_{\bar{\mathtt{r}}}$: if $i_{\mathtt{s}} \leq i_{\bar{\mathtt{r}}}$, because ρ_c and ρ' share a prefix $\rho_c[0..i_{\bar{\mathtt{r}}}]$ and $w''y \leq w$, $\mathtt{s}$ would be the earliest disagreeing participant instead of $\bar{\mathtt{r}}$.

Because $i_{\mathtt{s}} > i_{\bar{\mathtt{r}}}$, $\rho_c[0..i_{\bar{\mathtt{r}}}] = \rho'[0..i_{\bar{\mathtt{r}}}] \leq \rho'[0..i_{\mathtt{s}}]$. Because $\rho'[0..i_{\mathtt{s}}] = \alpha' \cdot G'' \leq \rho''$, it follows from prefix transitivity that $\rho_c[0..i_{\bar{\mathtt{r}}}] \leq \rho''$, thus re-establishing the induction hypothesis for $w''y$ with ρ'' as a witness run that begins with $\rho_c[0..i_{\bar{\mathtt{r}}}]$.

This concludes our proof that $I(w')$ contains runs that begin with $\rho_c[0..i_{\bar{\mathtt{r}}}]$, and in turn our proof by contradiction that $y_{\bar{\mathtt{r}}}$ must be a receive event.

End Proof of Claim I & II.

We can rewrite candidate run ρ_c as follows:

$$\rho_c = G_0 \xrightarrow{l_0} G_1 \dots G_{i_{\bar{r}}} \xrightarrow{l_{i_{\bar{r}}}} G_{i_{\bar{r}}+1} \dots \quad .$$

We have established that $l_{i_{\bar{r}}}$ must be a send event for $\bar{r}$. We can reason from Send Validity similarly to our construction of $\bar{\rho}$'s prefix above, and conclude that there exists a transition label and maximal suffix from $G_{i_{\bar{r}}}$ such that the resulting run no longer disagrees with $w{\Downarrow}_{\Gamma_{\bar{r}}}$. We update our candidate run ρ_c with the correct transition label and maximal suffix, update the set of states $\mathcal{S} \in \mathcal{P}$ to the new set of participants that disagree with the new candidate run, and repeat the construction above on the new candidate run until $\mathcal{S}$ is empty.

Termination is guaranteed in at most $|w|$ steps by the fact that the number of symbols in w that agree with the candidate run up to $i_{\bar{r}}$ must increase.

Upon termination, the resulting $\bar{\rho}$ satisfies the final remaining (3): $\bar{\rho} \in I$. This concludes the proof of the inductive step, and consequently the proof of the prefix-preservation of send transitions. □

Lemma 5.16. Let G be a global type and $\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ be the subset projection. Let w be a trace of $\{\!\{\mathscr{P}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$. It holds that $I(w)$ is non-empty.

Proof. We prove the claim by induction on the length of w.
Base Case: $w = \varepsilon$. The trace $w = \varepsilon$ is trivially consistent with all maximal runs, and $I(w)$ therefore contains all maximal runs. By definition of G, language $\mathcal{L}(G)$ is non-empty and at least one maximal run exists. Thus, $I(w)$ is non-empty.
Induction Step: Let wx be an extension of w by $x \in \Gamma$.

The induction hypothesis states that $I(w) \neq \emptyset$. To re-establish the induction hypothesis, we need to show $I(wx) \neq \emptyset$. We proceed by case analysis on whether x is a receive or send event.

Receive Case: let $x = \mathsf{p}\triangleleft\mathsf{q}?\mathsf{m}$. By Lemma 5.18, $I(wx) = I(w)$. $I(wx) \neq \emptyset$ follows trivially from the induction hypothesis and this equality.

Send Case: let $x = \mathsf{p}\triangleright\mathsf{q}!\mathsf{m}$. By Lemma 5.20, there exists a run in $I(wx)$ that shares a prefix with a run in $I(w)$. $I(wx) \neq \emptyset$ again follows trivially. □

Lemma A.6. Let G be a global type and $\{\!\{\mathscr{C}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ be the subset construction. Let w be a trace of $\{\!\{\mathscr{C}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$. If w is terminated, then w is G-complete.

Proof. We prove the claim by contraposition and assume that w is not G-complete. Then, there exists a run $\rho \in I(w)$ and a non-empty set of participants $\mathcal{S}$ such that for every $\mathsf{r} \in \mathcal{S}$, it holds that $w{\Downarrow}_{\Gamma_{\mathsf{r}}} \neq \big(\mathrm{trace}(\rho)\big){\Downarrow}_{\Gamma_{\mathsf{r}}}$ (*). Since w is a trace, we know there exists a run $(\vec{s}_0, \xi_0) \xrightarrow{w_0} \dots \xrightarrow{w_{n-1}} (\vec{s}_n, \xi_n)$ of $\{\!\{\mathscr{C}(G, \mathsf{p})\}\!\}_{\mathsf{p}\in\mathcal{P}}$ such that $w = w_0 \dots w_{n-1}$. We need to show that there exists $(\vec{s}_{n+1}, \xi_{n+1})$ with $(\vec{s}_n, \xi_n) \xrightarrow{w_n} (\vec{s}_{n+1}, \xi_{n+1})$ for some w_n. Given some participant p, let ρ_{p} denote the largest prefix of ρ that contains p's local view of w. Formally,

$$\rho_{\mathsf{p}} = \max\{\rho' \mid \rho' \leq \rho \,\wedge\, \mathrm{trace}(\rho'){\Downarrow}_{\Gamma_{\mathsf{p}}} = w{\Downarrow}_{\Gamma_{\mathsf{p}}}\} \quad .$$

Note that due to maximality, the next transition in ρ after ρ_{p} must have p as its active participant. Let q be the participant in $\mathcal{S}$ for whom ρ_{q} is the smallest. From Lemma 5.4 and (*), it follows that $\vec{s}_{n,\mathsf{q}}$ has outgoing transitions. If $\vec{s}_{n,\mathsf{q}}$ has outgoing send transitions, then $(\vec{s}_{n+1}, \xi_{n+1})$ exists trivially. If $\vec{s}_{n,\mathsf{q}}$ has outgoing receive transitions, it must be the case that the next transition in ρ after ρ_{q} is of the form $\mathsf{p} \to \mathsf{q} : \mathsf{m}$ for some p and m. From the fact that q is the participant with the smallest ρ_{q}, we know that $\rho_{\mathsf{q}} < \rho_{\mathsf{p}}$, and from the FIFO property of CSM channels it follows that m is in $\xi_n(\mathsf{p}, \mathsf{q})$. Then, the receive transition is enabled for q, and there exists $(\vec{s}_{n+1}, \xi_{n+1})$ with $(\vec{s}_n, \xi_n) \xrightarrow{\mathsf{p} \triangleleft \mathsf{p}?\mathsf{m}} (\vec{s}_{n+1}, \xi_{n+1})$. This shows that $w = w_1 \ldots w_{n-1}$ is not terminated and concludes the proof. □

Lemma A.7. Let G be a global type and $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ be the subset projection. Let w be a trace of $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$. If w is G-complete, then $w \in \mathcal{L}(G)$.

Proof. By definition of w being G-complete,

$$\forall \mathsf{p} \in \mathcal{P},\ \rho \in I(w).\ w \Downarrow_{\Gamma_{\mathsf{p}}} = \big(\mathrm{trace}(\rho)\big) \Downarrow_{\Gamma_{\mathsf{p}}} \ .$$

From Lemma 5.16, $I(w)$ is non-empty. Let ρ be a run in $I(w)$, and let $w' = \mathrm{trace}(\rho) \in \mathcal{L}(G)$. By the semantics of $\mathcal{L}(G)$, $\mathcal{L}(G)$ is closed under the $\sim$ relation, and thus it suffices to show that $w \sim w'$. Lemma 2.9 states that if w is FIFO-compliant, then $w \sim w'$ iff w' is FIFO-compliant and forall $\mathsf{p} \in \mathcal{P}$, $w \Downarrow_{\Gamma_{\mathsf{p}}} = w' \Downarrow_{\Gamma_{\mathsf{p}}}$. The fact that w is FIFO-compliant follows from Lemma 2.5 and w being a CSM trace; w' is FIFO-compliant by construction, and the last condition is satisfied by assumption that w is G-complete and by definition of w'. Thus, we conclude that $w \sim w'$. □

Theorem 5.14. Let G be a global type and $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ be the subset projection. Then, $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ implements G.

Proof. First, we show that $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ is deadlock-free, namely, that every finite trace extends to a maximal trace. Let w be a trace of $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$. Let w' denote the extension of w. If $w' \in \Gamma^\omega$, then w' is maximal and we are done. Otherwise, we have $w' \in \Gamma^*$. Let $(\vec{s}', \xi')$ denote the $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ configuration reached on w'. By definition of w' being the largest extension, w' is a terminated trace, and there exists no configuration reachable from $(\vec{s}', \xi')$. By Lemma A.6, w' is G-complete. By Lemma A.7, $w' \in \mathcal{L}(G)$. Therefore, all states in $\vec{s}'$ are final and all channels in ξ are empty, and w' is a maximal trace in $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$.

This concludes our proof that $\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}$ is deadlock-free.

Next, we show that $\mathcal{L}(\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}}) = \mathcal{L}(G)$. The backward direction, $\mathcal{L}(G) \subseteq \mathcal{L}(\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}})$, is given by Lemma 5.5. For the forward direction, let $w \in \mathcal{L}(\{\!\{ \mathscr{P}(G, \mathsf{p}) \}\!\}_{\mathsf{p} \in \mathcal{P}})$, and let $(\vec{s}, \xi)$ denote the configuration reached on w. We proceed by case analysis on whether w is a finite or infinite maximal trace.

Case: $w \in \Gamma^*$. We show a stronger property: w is a terminated trace. Then, we use Lemma A.6 and Lemma A.7 as above to obtain $w \in \mathcal{L}(G)$. By definition of $(\vec{s}, \xi)$ being final, all states in $\vec{s}$ are final and all channels in ξ are empty. We argue there does not exist a configuration reachable from $(\vec{s}, \xi)$. From Lemma A.4, all outgoing states from states in $\vec{s}$ must be receive transitions. However, no receive transitions are enabled because all channels in ξ are empty. Therefore, $(\vec{s}, \xi)$ is a terminated configuration and w is a terminated trace.
Case: $w \in \Gamma^\omega$. By the semantics of $\mathcal{L}(G)$, to show $w \in \mathcal{L}(G)$ it suffices to show:

$$\exists w' \in \Gamma^\omega.\, w' \in \mathcal{L}(\mathsf{GAut}(G)) \wedge w \preceq^\omega_\sim w' \ .$$

Claim I: $\bigcap_{u \leq w} I(u)$ contains an infinite run.

Proof of Claim I. First, we show that there exists an infinite run in $\mathsf{GAut}(G)$. We apply König's Lemma to an infinite tree where each vertex corresponds to a finite run. We obtain the vertex set from the intersection sets of w's prefixes; each prefix "contributes" a set of finite runs. Formally, for each prefix $u \leq w$, let V_u be defined as:

$$V_u := \bigcup_{\rho_u \in I(u)} \min\{\rho' \mid \rho' \leq \rho_u \wedge \forall \mathtt{p} \in \mathcal{P}.\, u\Downarrow_{\Gamma_{\mathtt{p}}} \leq \operatorname{trace}(\rho')\Downarrow_{\Gamma_{\mathtt{p}}}\} \ .$$

By Lemma 5.16, V_u is guaranteed to be non-empty. We construct a tree $\mathcal{T}_w(V, E)$ with $V := \bigcup_{u \leq w} V_u$ and $E := \{(\rho_1, \rho_2) \mid \rho_1 \leq \rho_2\}$. The tree is rooted in the empty run, which is included V by V_ε. V is infinite because there are infinitely many prefixes of w. $\mathcal{T}_w$ is finitely branching due to the finiteness of δ_G and the fact that each vertex represents a finite run. Therefore, there must exist a ray in $\mathcal{T}_w$ representing an infinite run in $\mathsf{GAut}(G)$.

Let ρ' be such an infinite run. We now show that $\rho' \in \bigcap_{u \leq w} I(u)$. Let v be a prefix of w. To show that $\rho' \in I(v)$, it suffices to show that one of the vertices in V_v lies on ρ'. In other words,

$$V_v \cap \{v \mid v \in \rho'\} \neq \emptyset \ .$$

Assume by contradiction that ρ' passes through none of the vertices in V_v. Then, for any $u' \geq u$, because intersection sets are monotonically decreasing, it must be the case that ρ' passes through none of the vertices in V'_u. Therefore, ρ' can only pass through vertices in V''_u, where $u'' \leq u$. However, the set $\bigcup_{u'' \leq u} V''_u$ has finite cardinality. We reach a contradiction, concluding our proof of the above claim.

End Proof of Claim I.

Let $\rho' \in \bigcap_{u \leq w} I(u)$, and let $w' = \operatorname{trace}(\rho')$. It is clear that $w' \in \Gamma^\omega$ and $w' \in \mathcal{L}(\mathsf{GAut}(G))$. It remains to show that $w \preceq^\omega_\sim w'$. By the definition of $\preceq^\omega_\sim$, it further suffices to show that:

$$\forall u \leq w,\ \exists u' \leq w', v \in \Gamma^*.\, uv \sim u' \ .$$

Let u be an arbitrary prefix of w. Because by definition $\rho' \in I(u)$, it holds that $u{\Downarrow_{\Gamma_p}} \leq \text{trace}(\rho'){\Downarrow_{\Gamma_p}}$.

For each participant $p \in \mathcal{P}$, let ρ'_p be defined as the largest prefix of ρ' such that $\text{trace}(\rho'_p){\Downarrow_{\Gamma_p}} = u{\Downarrow_{\Gamma_p}}$. Such a run is well-defined by the fact that u is a prefix of an infinite word w, and there exists a longer prefix v such that $u \leq v$ and $v{\Downarrow_{\Gamma_p}} \leq \text{trace}(\rho'){\Downarrow_{\Gamma_p}}$.

Let s be the participant with the maximum $|\rho'_s|$ in $\mathcal{P}$. Let $u' = \text{trace}(\rho'_s)$. Clearly, $u' \leq w'$. Because u' is $\text{trace}(\rho'_s)$ for the participant with the longest ρ'_s, it holds for all participants $p \in \mathcal{P}$ that $u{\Downarrow_{\Gamma_p}} \leq u'{\Downarrow_{\Gamma_p}}$. Then, there must exist $y_p \in \Gamma_p^*$ such that

$$u{\Downarrow_{\Gamma_p}} \cdot y_p = u'{\Downarrow_{\Gamma_p}} \quad .$$

Let y_p be defined in this way for each participant. We construct $v \in \Gamma^*$ such that $uv \sim u'$. Let v be initialised with ε. If there exists some participant in $\mathcal{P}$ such that $y_p[0] \in \Gamma_{p,!}$, append y_p to v and update y_p. If not, for all participants $p \in \mathcal{P}$, $y_p[0] \in \Gamma_{p,?}$. Each symbol $y_p[0]$ for all participants appears in u'. Let i_p denote for each participant the index in u' such that $u'[i] = y_p[0]$. Let r be the participant with the minimum index i_r. Append y_r to v and update y_r. Termination is guaranteed by the strictly decreasing measure of $\sum_{p \in \mathcal{P}} |y_p|$.

We argue that uv satisfies the inductive invariant of channel compliancy. In the case where v is extended with a send action, channel compliancy is trivially re-established. In the receive case, channel compliancy is re-established by the fact that the append order for receive actions follows that in u', which is FIFO-compliant by construction. We conclude that $uv \sim u'$ by applying Lemma 2.10 ☐

A.3 Additional Material for Section 5.4

Lemma A.8. Let p be a participant, G be a global type, G' be a syntactic subterm of G, and $\{\!\{\mathscr{C}(G, p)\}\!\}_{p \in \mathcal{P}}$ be its subset construction. Let s be some state in Q_p with $G' \in s$. Then, there is a run ρ_G in $\mathsf{GAut}(G)$ ending in state q'_G, i.e.

$$\rho_G = q_{0,G} \xrightarrow{\text{trace}(\rho_G)}{}^{*} q'_G,$$

such that $\mathscr{C}(G, p)$ will reach s on the projected trace, i.e.

$$\rho_p = s_{0,p} \xrightarrow{\text{trace}(\rho_G){\Downarrow_{\Gamma_p}}}{}^{*} s.$$

Proof. Recall that the set of states for $\mathscr{C}(G, p)$ is defined as a least fixed point:

$$Q_p := \text{lfp}^{\subseteq}_{\{s_{0,p}\}} \lambda Q.\, Q \cup \{\delta(s, a) \mid s \in Q \wedge a \in \Gamma_p\} \setminus \{\emptyset\}$$

where $\delta(s, a)$ is an intermediate transition relation that is defined for all subsets $s \subseteq Q_G$ and every event $a \in \Gamma_{\mathrm{p}}$ as follows:

$$\delta(s, a) := \{q' \in Q_G \mid \exists q \in s, q \xrightarrow{a}\xrightarrow{\varepsilon}{}^{*} q' \in \delta_{\downarrow}\}$$

From the definition of Q_{p}, there exists a sequence of states $s_1, \dots, s_n$ with $s_1 = s_{0,\mathrm{p}}$, $s_n = s$ and for every $i \in \{1, \dots, n-1\}$, it holds that

$$\exists a \in \Gamma_{\mathrm{p}}.\, \delta(s_i, a) = s_{i+1}$$

Let a_i denote the existential witness for each i. From the definition of $\delta(s, a)$, for every $i \in \{1, \dots, n-1\}$, it follows that

$$\forall q' \in s_{i+1}.\, \exists q \in s_i.\, q \xrightarrow{a_i}\xrightarrow{\varepsilon}{}^{*} q' \in \delta_{\downarrow} \ .$$

By assumption, $G' \in s$. There then exists a sequence of global syntactic subterms $G_1, \dots, G_n$ such that $G_1 = G$, $G_n = G'$ and for every $i \in \{1, \dots, n-1\}$, it holds that

$$G_i \in s_i \wedge G_{i+1} \in s_{i+1} \wedge G_i \xrightarrow{a_i}\xrightarrow{\varepsilon}{}^{*} G_{i+1} \in \delta_{\downarrow}$$

We can expand ε^*: for every $i \in \{1, \dots, n-1\}$, there exists $k_i \geq 0$ and a sequence of syntactic subterms $G_{i,0}, \dots, G_{i,k_i}$ such that $G_{i,0} = G_i$ and $G_{i,k_i} = G_{i+1}$ and

$$G_{i,0} \xrightarrow{a} G_{i,1} \in \delta_{\downarrow} \text{ and } G_{i,j} \xrightarrow{\varepsilon} G_{i,j+1} \in \delta_{\downarrow} \text{ for every } j \in \{1, k_i - 1\}.$$

This expansion yields a run $\rho_{\downarrow}$ in the projection by erasure $\mathsf{GAut}(G)\downarrow_{\mathrm{p}}$. Because of recursion terms, the expansion might not be unique, but we can pick the smallest k_i possible for every i. With the definition of $\delta_{\downarrow}$, it is trivial to translate this run in $\mathsf{GAut}(G)\downarrow_{\mathrm{p}}$ to a run ρ_G in $\mathsf{GAut}(G)$: the events $a \in \Gamma_{\mathrm{p}}$ become $a' \in \Sigma$ such that $a' \Downarrow_{\Gamma_{\mathrm{p}}} = a$ and ε becomes $b \in \Sigma$ such that $b \Downarrow_{\Gamma_{\mathrm{p}}} = \varepsilon$.

It is clear by construction that $s_1, \dots, s_n$ (with its corresponding transitions) serves as a witness for ρ_{p}, while $G_{1,0}, \dots, G_{1,k_1}, \dots, G_n$ (with its respective transitions) serves as a witness for ρ_G. □

Lemma A.9. Let G be a global type and let $\{\!\{B_{\mathrm{p}}\}\!\}_{\mathrm{p} \in \mathcal{P}}$ implement G with $B_{\mathrm{p}} = (Q_{B,\mathrm{p}}, \delta_{B,\mathrm{p}}, s_{B,0,\mathrm{p}}, F_{B,0,\mathrm{p}})$ for participant p. Let $s \in Q_{\mathrm{p}}$, $x \in \Gamma_{\mathrm{p}}$ and $s \xrightarrow{x} t \in \delta_{\mathrm{p}}$ from the subset construction $\mathscr{C}(G, \mathrm{p})$. Let $u \in \Gamma_{\mathrm{p}}^*$ such that $s_{0,\mathrm{p}} \xrightarrow{u}{}^{*} s$. Then, there exists $s', t' \in Q_{B,\mathrm{p}}$ such that $s_{B,0,\mathrm{p}} \xrightarrow{u}{}^{*} s'$ and $s' \xrightarrow{x} t' \in \delta_{B,\mathrm{p}}$.

Proof. Because $\{\!\{B_{\mathrm{p}}\}\!\}_{\mathrm{p} \in \mathcal{P}}$ implements G, it must hold that $\mathcal{L}(G)\Downarrow_{\Gamma_{\mathrm{p}}} \subseteq \mathcal{L}(B_{\mathrm{p}})$ since B_{p} must produce at least the behaviors as specified by G for its participant. It follows that $\mathrm{pref}(\mathcal{L}(G)\Downarrow_{\Gamma_{\mathrm{p}}}) \subseteq \mathrm{pref}(\mathcal{L}(B_{\mathrm{p}}))$. From Lemma 5.4, we know that $\mathcal{L}(G)\Downarrow_{\Gamma_{\mathrm{p}}} = \mathscr{C}(G, \mathrm{p})$. By construction of $\mathscr{C}(G, \mathrm{p})$, if ux is reachable from the initial state in $\mathscr{C}(G, \mathrm{p})$ then ux is the prefix of some word in $\mathcal{L}(G)\Downarrow_{\Gamma_{\mathrm{p}}}$. Therefore, it holds that $ux \in \mathrm{pref}(\mathcal{L}(G)\Downarrow_{\Gamma_{\mathrm{p}}})$

and consequently $ux \in \text{pref}(\mathcal{L}(B_p))$. Thus, there exists t'' such that B_p reaches t'' from the initial state on ux. It might be that B_p can also reach some other state s'' with u, from which it reaches t' with x then. This is fine. However, there also needs to be a transition with label x from s'. If there was not, we can reach a deadlock, contradicting that $\{\!\{B_p\}\!\}_{p \in \mathcal{P}}$ implements G. Thus, there is t' which can be reached with x from s', concluding our proof. □

Theorem 5.21 (Completeness)**.** If G is implementable, then the subset projection $\{\!\{\mathscr{P}(G, p)\}\!\}_{p \in \mathcal{P}}$ is defined.

Proof. From the fact that G is implementable, we know there exists a CSM $\{\!\{B_p\}\!\}_{p \in \mathcal{P}}$ that implements G. Showing that the subset projection is defined amounts to showing that Send and Receive Validity (Definitions 5.8 and 5.11) hold for the subset construction. Without loss of generality, we assume that B_p is deterministic for every p. If it was not, we can determinise every FSM. (One could also argue that any non-determinism should not change possible subsequent behaviours, all of which would either yield contradictions to protocol fidelity or deadlock freedom.)

We proceed by contradiction and assume the negation of Send and Receive Validity in turn, and in each case derive a contradiction to the fact that $\{\!\{B_p\}\!\}_{p \in \mathcal{P}}$ implements G. Specifically, we contradict protocol fidelity and show that $\mathcal{L}(G) \neq \mathcal{L}(\{\!\{B_p\}\!\}_{p \in \mathcal{P}})$.

To prove the inequality of the two languages, it suffices to prove the inequality of their respective prefix sets, i.e.

$$\{u \mid u \leq w \wedge w \in \mathcal{L}(G)\} \neq \{u \mid u \leq w \wedge w \in \mathcal{L}(\{\!\{B_p\}\!\}_{p \in \mathcal{P}})\}$$

Specifically, we show there is $v \in \Gamma^*$ such that

$$\begin{aligned} & v \in \{u \mid u \leq w \wedge w \in \mathcal{L}(\{\!\{B_p\}\!\}_{p \in \mathcal{P}})\} \wedge \\ & v \notin \{u \mid u \leq w \wedge w \in \mathcal{L}(G)\} \ . \end{aligned}$$

Because $\{\!\{B_p\}\!\}_{p \in \mathcal{P}}$ is deadlock-free by assumption, every trace either can be extended to end in a final configuration or to be infinite. Therefore, any word $v \in \Gamma^*$ that is a trace of $\{\!\{B_p\}\!\}_{p \in \mathcal{P}}$ is a member of the prefix set, i.e.

$$\exists\, (\vec{s}, \xi).\, (\vec{s}_0, \xi_0) \xrightarrow{v}{}^{*} (\vec{s}, \xi) \implies v \in \{u \mid u \leq w \wedge w \in \mathcal{L}(\{\!\{B_p\}\!\}_{p \in \mathcal{P}})\} \ .$$

By the semantics of $\mathcal{L}(G)$, for any $w \in \mathcal{L}(G)$, there exists $w' \in \mathcal{L}(\mathsf{GAut}(G))$ with $w \sim w'$. For any $w' \in \mathcal{L}(\mathsf{GAut}(G))$, it is straightforward that $I(w') \neq \emptyset$. Because intersection sets are closed under the indistinguishability relation (Corollary A.1), it holds that $I(w) \neq \emptyset$. Because $I(\text{-})$ is monotonically decreasing, if $I(w)$ is non-empty then for any $v \leq w$, $I(v)$ is non-empty. By the following, to show that a word v is not a member of the prefix set of $\mathcal{L}(G)$ it suffices to show that $I(v)$ is empty:

$$\forall v \in \Gamma^*.\, I(v) = \emptyset \implies \forall w.\, v \leq w \implies w \notin \mathcal{L}(G) \ .$$

Therefore, under the assumption of the negation of Send or Receive Validity respectively, we explicitly construct a witness v_0 satisfying:

(a) v_0 is a trace of $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$, and
(b) $I(v_0) = \emptyset$.

Send Validity (Definition 5.8). Assume that Send Validity does not hold for some participant $p \in \mathcal{P}$. Let $s \in Q_p$ be a state and $s \xrightarrow{p\triangleright q!m} s' \in \delta_p$ a transition in the subset construction $\mathscr{C}(G, p)$ such that

$$\text{tr-orig}(s \xrightarrow{p\triangleright q!m} s') \neq s \enspace .$$

Let D denote $s \setminus \text{tr-orig}(s \xrightarrow{p\triangleright q!m} s')$. By the negation of Send Validity, D is non-empty. Let G' be a syntactic subterm in D.

Because $G' \in s$, it follows from Lemma A.8 that there exists α such that $\alpha \cdot G'$ is a run in $\mathsf{GAut}(G)$. Let $\bar{w}$ be $\text{trace}(\alpha \cdot G')$. Because $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$ implements G, there exists a configuration $(\vec{t}, \xi)$ of $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$ such that $(\vec{t_0}, \xi_0) \xrightarrow{\bar{w}}{}^* (\vec{t}, \xi)$. Instantiating Lemma A.9 with s, $s \xrightarrow{p\triangleright q!m} s'$ and $\text{trace}(\alpha \cdot G')\Downarrow_{\Gamma_p}$, it follows that $\vec{t}_p$ has an outgoing transition labelled $p\triangleright q!m$. Let $\vec{t}_p \xrightarrow{p\triangleright q!m} t''$ be this transition.

The send transitions of any local machine in a CSM are always enabled. Formally, for all $w \in \Gamma^*$, $x \in \Gamma_!$, and $r \in \mathcal{P}$, if w is a trace of $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$ and $\vec{t}_{w,r} \xrightarrow{x} \vec{t}'_{w,r} \in \delta_r$, then wx is a trace of $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$. Instantiating this fact with $\bar{w}$ and $\vec{t}_p \xrightarrow{p\triangleright q!m} t''$, we obtain that $\bar{w} \cdot p\triangleright q!m$ is a trace of $\{\!\{B_p\}\!\}_{p\in\mathcal{P}}$.

Let $\bar{w} \cdot p\triangleright q!m$ be our witness v_0; it then follows that v_0 satisfies (a). It remains to show that v_0 satisfies (b), namely $I(\bar{w} \cdot p\triangleright q!m) = \emptyset$.

Claim I: All runs in $I(\bar{w})$ begin with $\alpha \cdot G'$.

Proof of Claim I. Recall that $\bar{w}$ is defined as $\text{trace}(\alpha \cdot G')$. Assume by contradiction that $\rho' \in I(\bar{w})$ and ρ' does not begin with $\alpha \cdot G'$. Due to the syntactic structure of global runs, the first divergence between two runs must correspond to a syntactic subterm of the form $\sum_{i\in I} p' \to q'_i : m'_i.G'_i$. Let p' be the sender in the first divergence between ρ' and $\alpha \cdot G'$, and let the two runs respectively contain the subterms G'_i and G'_j. Because ρ' is in $\mathrm{R}^G_{p'}(\bar{w})$, it holds that $\bar{w}\Downarrow_{\Gamma_{p'}} \leq \text{trace}(\rho')\Downarrow_{\Gamma_{p'}}$. Because $\bar{w} = \text{trace}(\alpha \cdot G')$, we can rewrite the inequality as $\text{trace}(\alpha \cdot G')\Downarrow_{\Gamma_{p'}} \leq \text{trace}(\rho')\Downarrow_{\Gamma_{p'}}$. We know that $\text{trace}(\alpha \cdot G')\Downarrow_{\Gamma_{p'}}$ and $\text{trace}(\rho')\Downarrow_{\Gamma_{p'}}$ share a common prefix, followed by different send actions from p', i.e. they are respectively of the form $x' \cdot p'\triangleright q_j!m'_j \cdot y'$ and $x' \cdot p'\triangleright q_i!m'_i \cdot z'$. We arrive at a contradiction.

End Proof of Claim I.

Recall that $G' \in D$ and $D = s \setminus \text{tr-orig}(s \xrightarrow{p\triangleright q!m} s')$. By the definition of $\text{tr-orig}(\text{-})$ (Definition 5.3), there does not exist a global syntactic subterm G'' with $G' \xrightarrow{l'}{}^* G'' \in \delta_G$ such that $l'\Downarrow_{\Gamma_p} = p\triangleright q!m$. Therefore, there does not exist a maximal run in $\mathrm{R}^G_p(\bar{w} \cdot p\triangleright q!m)$, and $I(\bar{w} \cdot p\triangleright q!m) = \emptyset$ follows.

Our witness $v_0 = \bar{w} \cdot \mathsf{p} \triangleright \mathsf{q} ! \mathsf{m}$ thus satisfies both conditions (a) and (b) required for a contradiction. This concludes our proof that Send Validity is required to hold.

Receive Validity (Definition 5.11). Assume that Receive Validity does not hold for some participant $\mathsf{p} \in \mathcal{P}$. In other words, there exists $s \in Q_\mathsf{p}$ with two transitions $s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1} s_1$, $s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2 \in \delta_\mathsf{p}$ and $G_2 \in \text{tr-dest}(s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2)$ such that

$$\mathsf{q}_1 \neq \mathsf{q}_2 \wedge \mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1 \in \text{msgs}^{\mathsf{p}}_{(G_2 \ldots)} \quad .$$

Claim II: There exists $u \in \Gamma^*$ such that both $u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1$ and $u \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2$ are traces of $\{\!\{ B_\mathsf{p} \}\!\}_{\mathsf{p} \in \mathcal{P}}$.

Proof of Claim II. By the negation of Receive Validity, we have

$$G_2 \in \text{tr-dest}(s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2) \subseteq s_2 \quad .$$

From Lemma A.8 for s_2 and $G_2 \in s_2$, there exists ρ' such that ρ' ends in G_2 and is a run in $\mathsf{GAut}(G)$. Because $\text{trace}(\rho')$ is a prefix in $\mathcal{L}(G)$ and by assumption $\{\!\{ B_\mathsf{p} \}\!\}_{\mathsf{p} \in \mathcal{P}}$ implements G, there exists a $\{\!\{ B_\mathsf{p} \}\!\}_{\mathsf{p} \in \mathcal{P}}$ configuration $(\vec{t}, \xi)$ such that $(\vec{t_0}, \xi_0) \xrightarrow{\text{trace}(\rho')}{}^* (\vec{t}, \xi)$. By the subset construction, it holds that $\mathscr{C}(G, \mathsf{p})$ reaches s on $\text{trace}(\rho')$. Instantiating Lemma A.9 twice with $s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1} s_1$, $s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2$ and $\text{trace}(\rho') \Downarrow_{\Gamma_\mathsf{p}}$, we obtain $t_1 \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1} t'_1$ and $t_2 \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} t'_2$. From the determinacy of B_p, it holds that $t_1 = t_2$. Therefore, it holds that $\vec{t}_\mathsf{p} = t_1$ and there exist two outgoing transitions from $\vec{t}_\mathsf{p}$ labelled with $\mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1$ and $\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2$.

From the fact that $s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2 \in \delta_\mathsf{p}$, there exist

$$G_1 \in s \text{ and } G'_2 \in \text{tr-dest}(s \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ? \mathsf{m}_2} s_2) \subseteq s_2$$

such that $G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p} : \mathsf{m}_2} G'_2 \in \delta_G$. Either $G_2 = G'_2$, or G_2 is reachable from G'_2 via ε-transitions for p. Without loss of generality, assume that $G_2 = G'_2$; if $G'_2 \neq G_2$ then G'_2 can also be picked as the witness from the definition of msgs^{-}_{-}. We rewrite ρ' as follows:

$$\rho' := \alpha \cdot G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p} : \mathsf{m}_2} G_2$$

From the negation of Receive Validity, we know that

$$\mathsf{p} \triangleleft \mathsf{q}_1 ? \mathsf{m}_1 \in \text{msgs}^{\mathsf{p}}_{(G_2 \ldots)}$$

Then, there exists some suffix β such that the transition $\xrightarrow{\mathsf{q}_1 \to \mathsf{p} : \mathsf{m}_1}$ occurs in β and $\alpha \cdot G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p} : \mathsf{m}_2} G_2 \cdot \beta$ is a maximal run. Let ρ denote this maximal run. Let

$G_3 \xrightarrow{\mathsf{q}_1 \to \mathsf{p}:\mathsf{m}_1} G_4$ be the earliest occurrence of $\xrightarrow{\mathsf{q}_1 \to \mathsf{p}:\mathsf{m}_1}$ in β. We rewrite the suffix β in ρ to reflect the existence of G_3 and G_4:

$$\rho := \alpha \cdot G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p}:\mathsf{m}_2} G_2 \cdot \beta_1 \cdot G_3 \xrightarrow{\mathsf{q}_1 \to \mathsf{p}:\mathsf{m}_1} G_4 \cdot \beta_2$$

Note that β_1 does not contain any transitions of the form $\xrightarrow{\mathsf{q}_1 \to \mathsf{p}:\mathsf{m}_1}$.

Let $\bar{w}$ denote $\mathrm{trace}(\alpha)$, and $\bar{v}$ denote $\mathrm{trace}(\beta_1)$. To produce a witness for u, we show that $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$ is a trace of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$, and in the resulting CSM configuration $(\vec{s}', \xi')$, $\vec{s}'_\mathsf{p}$ has two outgoing transitions labelled $\mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$ and $\mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$. Moreover, we show that the channels $\xi'(\mathsf{q}_1, \mathsf{p})$ and $\xi'(\mathsf{q}_2, \mathsf{p})$ respectively contain the messages m_1 and m_2 at the head.

First, we show that $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$ is a trace of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$.

By assumption that $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ implements G, both $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2$ and $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$ are traces of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$. Let $(\vec{s}'', \xi'')$ and $(\vec{s}''', \xi''')$ respectively denote configurations of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ such that

$$(\vec{s}_0, \xi_0) \xrightarrow{\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2} (\vec{s}'', \xi'') \xrightarrow{\mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2 \cdot \bar{v}} (\vec{s}''', \xi''') \ .$$

Because send actions are always enabled in a CSM, it suffices to show that $\vec{s}''_{\mathsf{q}_1}$ has an outgoing transition label $\mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$. We do so by showing that $\vec{s}''_{\mathsf{q}_1} = \vec{s}'''_{\mathsf{q}_1}$: it is clear from the fact that $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2 \cdot \bar{v} \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$ is a trace of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ that $\vec{s}'''_{\mathsf{q}_1}$ has an outgoing transition label $\mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$.

Due to the determinacy of subset construction, it suffices to show that

$$(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)\Downarrow_{\Gamma_{\mathsf{q}_1}} = (\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2 \cdot \bar{v})\Downarrow_{\Gamma_{\mathsf{q}_1}} \ .$$

This equality follows from the definition of msgs^{-}_{-} and the fact that $\mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1 \in \mathrm{msgs}^{\mathsf{p}}_{(G_2 \ldots)}$: because the blocked set of participants in msgs^{-}_{-} monotonically increases, and for any $G', \mathcal{B}$, no actions in a run suffix starting with G' involving participants in $\mathcal{B}'$ are included in $\mathrm{msgs}^{\mathcal{B}'}_{G'}$, we know that $\mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$ must be the lexicographically earliest action involving q_1 in $\bar{v} \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1 \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$. In other words, $\bar{v}\Downarrow_{\Gamma_{\mathsf{q}_1}} = \varepsilon$.

This concludes the reasoning that $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$ is a trace of $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$.

Recall that $(\vec{s}', \xi')$ is the $\{\!\{B_\mathsf{p}\}\!\}_{\mathsf{p}\in\mathcal{P}}$ configuration reached on $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$. We showed above that $\vec{s}_\mathsf{p}$ has two outgoing transitions labelled $\mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$ and $\mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$. It follows from the equality below that $\vec{s}'_\mathsf{p}$ likewise has two outgoing transitions labelled $\mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$ and $\mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$:

$$(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)\Downarrow_{\Gamma_\mathsf{p}} = (\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1)\Downarrow_{\Gamma_\mathsf{p}} \ .$$

We now show that the channels $\xi'(\mathsf{q}_1, \mathsf{p})$ and $\xi'(\mathsf{q}_2, \mathsf{p})$ respectively contain the messages m_1 and m_2 at the head. Recall that $\bar{w}$ is defined as $\mathrm{trace}(\alpha)$; this from the fact that $\xi_{\bar{w}}$ is uniquely determined by $\bar{w}$ and all channels in $\xi_{\bar{w}}$ are empty.

Let $u := \bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{q}_1 \triangleright \mathsf{p}!\mathsf{m}_1$. This concludes our proof that both $u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$ and $u \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$ are traces of $\{\!\{ B_\mathsf{p} \}\!\}_{\mathsf{p} \in \mathcal{P}}$.

End Proof of Claim II.

The next claim establishes that our witness $u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$ satisfies (b).

Claim III: It holds that $I(u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1) = \emptyset$.

Proof of Claim III. This claim follows trivially from the observation that every run in $I(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)$ must begin with $\alpha \cdot G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p}:\mathsf{m}_2} G_2$. Because $I(u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1) \subseteq I(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)$, and the $\mathrm{trace}(\text{-})$ of every run in $I(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)$ starts with $\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2 \cdot \mathsf{p} \triangleleft \mathsf{q}_2 ?\mathsf{m}_2$, therefore $I(u \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1)$ is empty.

End Proof of Claim III.

From here, the reasoning that every run in $I(\bar{w} \cdot \mathsf{q}_2 \triangleright \mathsf{p}!\mathsf{m}_2)$ must begin with $\alpha \cdot G_1 \xrightarrow{\mathsf{q}_2 \to \mathsf{p}:\mathsf{m}_2} G_2$ is identical to the reasoning for the analogous claim in the Send Validity case, and thus omitted.

By choosing $v_0 := \bar{u} \cdot \mathsf{p} \triangleleft \mathsf{q}_1 ?\mathsf{m}_1$, we thus establish both conditions (a) and (b) required for a contradiction. This concludes our proof that Receive Validity is required to hold. □

Curriculum Vitae

Research Interests

Concurrent and Message-passing Systems, Software Verification,
Formal Methods, and Security Protocols

Education and Employment

2024 – present	Research (and Development) Specialist, University of Luxembourg.
2019 – 2024	Doctoral Researcher, Max Planck Institute for Software Systems, Kaiserslautern, Germany.
2017 – 2019	M.Sc., Computer Science, University of Saarland, Germany.
2014 – 2017	B.Sc., Computer Science, University of Kaiserslautern, Germany.