Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

A Calculus for Modeling and Analyzing Conversations in Service-Oriented Computing

Hugo Filipe Mendes Torres Vieira

Dissertação apresentada para a obtenção do Grau de Doutor em Informática pela Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Lisboa, 2010

This dissertation was prepared under the supervision of
Professor Luís Caires,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To my family.

Acknowledgements

This dissertation is the result of the work developed over several years, during which I learnt, was encouraged, and received helpful input from many people to whom I owe a great deal. For what it's worth, I write their names here as a small symbol of my gratitude.

First of all, I would like to thank Luís Caires from whom I have learnt so much, who has been a friend in difficult times, who encouraged me to keep going, who made this work flourish. Luís, thank you.

I would also like to thank my coauthors, Emilio Tuosto, João Costa Seco and Carla Ferreira, to whom I owe more than a line in my résumé. Also, I thank other people from the Programming Languages and Models group of my department, for their interest in the work and for their support, namely Luís Monteiro and José Pacheco. Thanks also to Luísa Lourenço for her interest and follow-up work. My thanks to my office mates Anabela, Cristóvão, David, Guida, Paula and Vasco who contributed to a great work environment.

I would like to express my gratitude to Nobuko Yoshida and Mariangiola Dezani-Ciancaglini for their interest and availability to discuss this work, in particular the conversation type system. I thank all the members of the jury of my PhD defense for their suggestions to improve the thesis, in particular Rosario Pugliese and Nobuko Yoshida. I also thank my colleagues from the Sensoria project for all the interesting discussions, and all the anonymous reviewers that helped us improve our papers, even by rejecting them.

My thanks to Rui for the companionship during the period of the writing, to Étienne, Paolo and Graham for their encouragement. To my friends João, Miguel, Renata and Ricardo, my deepest thanks for always being there.

My sincere thanks to my "in-law" family Maurice, Nadia, Paolo and Rita for their support and encouragement. My deepest thanks to my sister Cristiana and her husband Jorge, for their pure friendship. To my parents, António and Conceição who made it all happen, thank you. Special thanks to my son Tiago for being the joy of my life. To Francesca, for her unquantifiable support, continuous encouragement and unconditional love: thank you.

This work was supported by the PhD Scholarship SFRH/BD/23760/2005 from the Fundação para a Ciência e Tecnologia.

Abstract

The service-oriented computing paradigm has motivated a large research effort in the past few years. On the one hand, the wide dissemination of Web-Service technology urged for the development of standards, tools and formal techniques that contributed for the design of more reliable systems. On the other hand, many of the problems presented in the study of service-oriented applications find an existing work basis in well-established research fields, as is the case of the study of interaction models that has been an active field of research in the last couple of decades. However, there are many new problems raised by the service-oriented computing paradigm in particular that call for new concepts, dedicated models and specialized formal analysis techniques. The work presented in this dissertation is inserted in such effort, with particular focus on the challenges involved in governing interaction in service-oriented applications.

One of the main innovations introduced by the work presented here is the way in which multiparty interaction is handled. One reference field of research that addresses the specification and analysis of interaction of communication-centric systems is based on the notion of *session*. Essentially, a session characterizes the interaction between two parties, a client and a server, that exchange messages between them in a sequential and dual way. The notion of session is thus particularly adequate to model the client/server paradigm, however it fails to cope with interaction between several participants, a scenario frequently found in real service-oriented applications. The approach described in this dissertation improves on the state of the art as it allows to model and analyze systems where several parties interact, while retaining the fundamental flavor of session-based approaches, by relying on a novel notion of *conversation*: a simple extension of the notion of session that allows for several parties to interact in a single medium of communication in a disciplined way, via labeled message passing.

The contributions of the work presented in this dissertation address the modeling and analysis of service-oriented applications in a rigorous way: First, we propose and study a formal model for service-oriented computing, the Conversation Calculus, which, building on the abstract notion of conversation, allows to capture the interactions between several parties that are relative to the same service task using a single medium of communication. Second, we introduce formal analysis techniques, namely the conversation type system and progress proof system that can be used to ensure, in a provably correct way and at static verification time (before deploying such applications), that systems enjoy good properties such as "the prescribed protocols will be followed at runtime by all conversation participants" (*conversation fidelity*) and "the system will never run into a stuck state" (*progress*).

We give substantial evidence that our approach is already effective enough to model and type sophisticated service-based systems, at a fairly high level of abstraction. Examples of such systems include challenging scenarios involving simultaneous multiparty conversations, with

concurrency and access to local resources, and conversations with a dynamically changing and unanticipated number of participants, that fall out of scope of previous approaches.

Sumário

O paradigma da Computação Orientada a Serviços suscitou um imenso esforço de investigação nos últimos anos. Por um lado, a grande disseminação do uso da tecnologia de Serviços Web requeria o desenvolvimento de standards, ferramentas e técnicas formais que contribuíssem para o desenho de sistemas com menos erros de execução. Por outro lado, muitos dos desafios que se apresentam no estudo de aplicações orientadas a serviços encontram bases de trabalho muito fortes no seio de diversas comunidades científicas, como é o caso do estudo de modelos de interacção que tem sido uma área de investigação muito activa nas últimas décadas. Contudo, existem muitos problemas específicos ao paradigma da computação orientada a serviços que requerem novos conceitos, modelos específicos e técnicas de verificação especializadas. O trabalho apresentado nesta dissertação insere-se nesse esforço, endereçando, em particular, as problemáticas associadas à interacção entre participantes numa aplicação orientada a serviços.

Um dos aspectos inovadores do trabalho apresentado aqui prende-se com a forma como são tratadas as interacções entre vários participantes. Uma das abordagens tradicionalmente propostas para modelar a interacção de sistemas orientados a serviços é baseada na noção de *sessão*. Essencialmente, uma sessão caracteriza a interacção entre dois participantes, um cliente e um servidor, que trocam mensagens entre si de uma forma sequencial e dual. A noção de sessão é então particularmente adequada ao paradigma cliente/servidor, contudo trata-se de uma abordagem que fica aquém quando se trata de modelar interacção entre vários participantes, um cenário frequente em sistemas orientados a serviços reais. A abordagem aqui descrita melhora o estado da arte pois permite modelar interacção entre vários participantes de uma forma igualmente canónica baseado numa inovadora noção de *conversação*: uma extensão simples da noção de sessão que permite que múltiplos participantes interajam no mesmo meio de comunicação de uma forma disciplinada, através de mensagens etiquetadas.

As contribuições do trabalho apresentado nesta dissertação endereçam a modelação e análise de aplicações orientadas a serviços de uma forma matematicamente precisa: Em primeiro lugar, propomos e estudamos um modelo formal para computação orientada a serviços, o "Conversation Calculus" (cálculo das conversações) que, baseando-se na noção abstracta de conversação, permite especificar todas as interacções entre várias partes que dizem respeito à mesma tarefa de serviço usando um único meio de comunicação. Em segundo lugar, introduzimos técnicas formais, nomeadamente o "conversation type system" (sistema de tipos de conversações) e o "progress proof system" (sistema de prova de progresso) que permitem garantir boas propriedades dos sistemas, de forma matematicamente demonstrável e em tempo estático de verificação (antes que as aplicações sejam executadas), tais como "os protocolos previstos vão ser seguidos pelos intervenientes" ("*conversation fidelity*") e "o sistema nunca irá entrar num estado bloqueado" ("*progress*").

Mostramos, de forma substancial, que na nossa abordagem é desde já possível modelar e tipificar aplicações orientadas a serviços bastante sofisticadas, a um nível de abstracção bastante alto. Exemplos de tais sistemas incluem cenários que envolvem várias conversações simultâneas entre múltiplos participantes, com concorrência e acesso a recursos locais, e conversações com um número dinâmico e imprevisto de participantes, cenários esses que estão fora do alcance de abordagens prévias.

Notation

Names	$a, b, c, a_i, \ldots$
List of names	$\tilde{a}$
Set of names	Λ
Variables	$x, y, z, x_i, \ldots$
List of variables	$\tilde{x}$
Set of variables	$\mathcal{V}$
Identifiers	$n, m, o, u, v, n_i, \ldots$
List of identifiers	$\tilde{n}$
Labels	$l, s, l', l_i, \ldots$
Set of labels	$\mathcal{L}$
Set of shared labels	$\mathcal{L}_\star$
Set of plain labels	$\mathcal{L}_p$
Recursion variables	$\mathcal{X}, \mathcal{Y}$
Set of recursion variables	χ
Processes	$P, Q, R, P_i, P', \ldots$
Set disjointness	$\#$
Unitary identifier substitution	$\{n/m\}$
α-equivalence relation	$\equiv_\alpha$
Internal action	τ
Transition labels	$\lambda, \lambda', \lambda_i, \ldots$
Set of transition labels	$\mathcal{T}$
Transition relation	$\xrightarrow{\lambda}$
Reduction relation	$\rightarrow$
Output polarity	$\mathtt{!}$
Input polarity	$\mathtt{?}$
Internal action polarity	$\mathtt{\tau}$
Polarity type	p
Behavioral types	$B, B', B_i, \ldots$
Message types	$M, M', M_i, \ldots$
Conversation types	$C, C', C_i, \ldots$
Located types	$L, L', L_i, \ldots$
Process types	$T, T', T_i, \ldots$
Replicated message type	$\star M$

Exponential output message types	$\star M^!, \star M^!_i, \ldots$
Exponential output behavioral types	$\star B^!, \star B^!_i, \ldots$
Exponential output located types	$\star L^!, \star L^!_i, \ldots$
Conformance relation	$\asymp$
Apartness predicate	$\#$
Subtyping relation	$<:$
Type equivalence	$\equiv$
Unitary message type substitution	$\{M_1/M_2\}$
Merge relation	$\bowtie$
Typing judgment	$P :: L$
Static process context	$\mathcal{C}[\cdot]$
Action prefix	$c \cdot l -$
Here direction	$\downarrow$
Up direction	$\uparrow$
Directions	$d, d', \ldots$
Action prefixes	$\alpha, \alpha_i, \ldots$
Action prefix summation	$\Sigma_{i \in I}\ \alpha_i$
Here-up direction	$\downarrow\uparrow$
Extended directions	d_e
Action labels	$\sigma, \sigma', \sigma_i, \ldots$
Undefined synchronization	$\circ$
Synchronization function	$\bullet$
Strong bisimilarity	$\sim$
Active processes	$U, V, U_i, \ldots$
Structural congruence	$\equiv$
Weak transition relation	$\stackrel{\lambda}{\Longrightarrow}$
Weak bisimilarity	$\approx$
Recursive type	$B\langle \mathcal{X} \rangle$
Typing judgment	$P :: T$
Action prefix	$l^d -$
Conversation identifiers	$\mathsf{n}, \mathsf{m}, \mathsf{n}', \mathsf{n}_i, \ldots$
Events	$e, e', e_i, \ldots$
Event ordering	Γ
Recursion Environment	Δ
Locator	ℓ
Progress proof system judgment	$\Gamma \vdash_\ell P$
Primitive action	a
Atomic actions	$A, B, \ldots$
Basic programs	$P, Q, \ldots$
Compensable programs	$R, T, \ldots$

Contents

1 Introduction **1**
1.1 Background . . . 2
1.1.1 Service-Oriented Software Development . . . 2
1.1.2 Models of Interaction . . . 3
1.2 Modeling and Analyzing Service-Oriented Systems . . . 6
1.2.1 From Binary to Multiparty Interaction . . . 6
1.2.2 Key Aspects of Service-Oriented Computing . . . 8
1.2.3 The Conversation Calculus . . . 10
1.2.4 Programming in the Conversation Calculus . . . 13
1.2.5 Analyzing Conversations . . . 19
1.3 Contributions and Structure of the Dissertation . . . 25
1.3.1 Contributions . . . 25
1.3.2 Structure of the Dissertation . . . 26

2 Preliminaries **29**
2.1 π_{lab}-Calculus: π-Calculus with Label Indexed Channels . . . 29
2.2 π_{lab}-Calculus Operational Semantics . . . 33
2.2.1 Proving a Transition . . . 36
2.2.2 Programming the Purchase Scenario in the π_{lab}-Calculus . . . 37
2.3 Remarks . . . 39

3 Introducing Conversation Types **41**
3.1 Analysis of Dynamic Conversations . . . 41
3.2 Type Language . . . 43
3.2.1 Linear and Shared Types . . . 44
3.2.2 Apartness . . . 45
3.2.3 Subtyping . . . 46
3.2.4 Merge Relation . . . 48
3.3 Type System . . . 55
3.3.1 Type Safety . . . 57
3.3.2 Typing the Purchase Example . . . 60
3.4 Remarks . . . 63

4 The Conversation Calculus **65**
4.1 Modeling Service-Oriented Computation . . . 65
4.2 The Core Conversation Calculus . . . 67
4.3 Operational Semantics . . . 74
4.4 Behavioral Semantics . . . 80
4.4.1 Normal Form . . . 84
4.4.2 Structural Congruence Based Operational Semantics . . . 86
4.4.3 Weak Bisimilarity . . . 88
4.5 Idioms for Service-Oriented Computing . . . 91
4.5.1 Proving Interaction in the Purchase Scenario . . . 93

4.5.2 Programming a Finance Portal 95
4.6 Remarks 98

5 Typing Conversations **101**
5.1 Analysis of Dynamic Conversations 101
5.2 Type Language 104
5.2.1 Projecting Types 106
5.2.2 Typing Recursive Behavior 107
5.2.3 Apartness 108
5.2.4 Subtyping 110
5.2.5 Merge Relation 112
5.3 Type System 117
5.3.1 Type Safety 121
5.3.2 Derived Typings for Service Idioms 124
5.3.3 Typing Examples 126
5.4 Remarks 134

6 Proving Progress of Conversations **137**
6.1 Progress of Dynamic Conversations 137
6.2 Event Ordering 139
6.2.1 Event Ordering Operators 140
6.2.2 Ordering Recursive Behavior 141
6.2.3 Locating Events 143
6.3 Progress Proof System 143
6.3.1 Progress Property 145
6.3.2 Proving Progress in the Purchase Example 148
6.4 Proving Progress in the π_{lab}-Calculus 150
6.5 Remarks 152

7 Exception Handling **155**
7.1 Recovering from Error 155
7.2 The Exception Handling Conversation Calculus (CC_{exc}) 157
7.3 Operational Semantics 158
7.4 Behavioral Semantics 161
7.4.1 Weak Bisimilarity 163
7.5 Implementing Compensations in the CC_{exc} 164
7.5.1 The cCSP Language 165
7.5.2 Encoding cCSP in CC_{exc} 166
7.5.3 Compensations in a Service-Oriented Application 169
7.6 Remarks 172

8 Conclusions **173**
8.1 Summary of Contributions 173
8.2 Evolutions and Future Work 175

Bibliography **178**

A Proofs **187**
A.1 Chapter 3 187
A.2 Chapter 4 198
A.3 Chapter 5 220
A.4 Chapter 6 255
A.5 Chapter 7 290

List of Figures

1.1 Buyer Seller Interaction. 7
1.2 Buyer Seller Shipper Interaction. 8
1.3 Credit Request Interaction. 17

2.1 The π_{lab}-Calculus Syntax. 30
2.2 Transition Rules. 35

3.1 Basic Conversation Types Syntax. 43
3.2 Behavioral Types Subtyping Rules. 48
3.3 Located Types Subtyping Rules. 49
3.4 Behavioral Type Merge Relation Rules. 52
3.5 Located Type Merge Relation Rules. 55
3.6 Typing Rules. 57
3.7 Located Types Reduction Rules. 58

4.1 The Core Conversation Calculus Syntax. 67
4.2 Transition Rules for Basic Operators (π-Calculus). 80
4.3 Transition Rules for Conversation Operators. 81
4.4 Basic Structural Congruence Rules (π-Calculus). 86
4.5 Conversation Structural Congruence Rules. 86
4.6 Reduction Rules. 87
4.7 Service Idioms. 92

5.1 Conversation Types Syntax. 104
5.2 Process Types Subtyping Rules. 112
5.3 Behavioral Type Merge Relation Rules. 116
5.4 Process Types Merge Relation Rules. 117
5.5 Typing Rules. 120
5.6 Process Types Reduction Rules. 121
5.7 Service Idioms Derived Typing Rules. 126
5.8 Credit Request Example Typing. 133
5.9 The Newsfeed Conversation CC Code. 133
5.10 The Newsfeed System Typing. 134

6.1 Progress Proof Rules. 146
6.2 Code of the Purchase Service Collaboration Example. 148
6.3 Progress Proof Rules (π_{lab}-Calculus). 151

7.1 The Exception Handling Conversation Calculus ($\mathrm{CC_{exc}}$) Syntax. 158
7.2 Transition Rules for Basic Operators (π-Calculus). 160
7.3 Transition Rules for Conversation Operators. 161
7.4 Transition Rules for Exception Handling Operators. 161
7.5 Compensating CSP (cCSP) Syntax. 165
7.6 Encoding of cCSP Basic Programs in the $\mathrm{CC_{exc}}$. 167
7.7 Encoding of cCSP Compensable Programs in the $\mathrm{CC_{exc}}$. 169

Chapter 1

Introduction

The paradigm of Service-Oriented Computing has motivated much research effort in the past few years. On the one hand, the wide dissemination of Web-Service technology urged for the development of standards, tools and formal techniques that would contribute to the design of more reliable systems — for example, systems that do not exhibit runtime errors and that never run into deadlocked states. On the other hand, many of the problems presented in the study of service-oriented applications find an existing work basis in well-established research fields, as is the case of the study of interaction models that has been a very active field of research in the last couple of decades. However, there are many new problems raised by the service-oriented paradigm in particular that call for new concepts, dedicated models and specialized formal analysis techniques. The work presented in this dissertation is inserted in such effort, with particular focus on the challenges involved in governing interaction in service-oriented applications.

This Introduction is divided in three parts and is structured as follows:

- The first part (Section 1.1) sets the background of the work and starts with a brief description of some of the state of the art industry standards for service-oriented technology, so as to provide the reader with some context and identify some of the problems that motivate our development (Section 1.1.1); then we present in general lines the research stream in which our work is inserted in, and discuss some open problems — that fall out of the scope of related approaches — so as to motivate our work (Section 1.1.2).

- The second part (Section 1.2) introduces our main technical developments: to pinpoint our contributions we discuss how our approach differs from previous works in order to cope with the previously mentioned open problems (Section 1.2.1). We proceed by identifying the key features of service-oriented computing that are at the basis of the design principles of our development (Section 1.2.2). Then we present our proposed model for service-oriented computing — the Conversation Calculus — by first describing the primitives (Section 1.2.3) and by programming some examples (Section 1.2.4). Finally we introduce the key ideas behind our main contributions at the level of analysis techniques — the conversation type system and the progress proof system (Section 1.2.5).

- The third and last part (Section 1.3) lists the contributions of the work and describes the structure of the dissertation.

1.1 Background

1.1.1 Service-Oriented Software Development

The goal of the service-oriented computing paradigm is to allow to build large systems from smaller applications that are distributed throughout the network. The motivation is a recurring argument in software development: (re)use software functionalities that are already available in order to build more complex applications, instead of building everything from scratch. Furthermore, it is well known that implementing a complex application by separating the basic functionalities in dedicated modules can contribute to the reliability of the developed software, as such modules can be separately tested and analyzed. Moreover, such modules can be replaced just as long as the external interfaces are kept intact, allowing for easier maintenance and evolution of the applications. If we are to add to this classical set of arguments the fact that services run remotely and therefore use remote resources to carry out their tasks, then we obtain a highly appealing paradigm for software development.

A service-oriented application is built from the *collaboration* of several partners, each one contributing by implementing a part of the overall functionality. Such collaborations are typically established dynamically and without centralized control. Proposed industry standards for Web Services technology such as UDDI [7] or WS-Discovery [6] give support for the dynamic discovery of services, while SOAP [50] defines a protocol to support the decentralized information exchange between Web-Services. A central concern in the development of service-oriented systems is thus the design of interaction protocols that allow for the decentralized and dynamic collaboration between several parties in a reliable way.

The Web Services Choreography Description Language (WS-CDL) [60] is one of the proposals for languages that support the specification of Web-Service interaction. The term *choreography* reflects the idea of a decentralized dynamic setting: partners interact as if they were dancers on a stage, and due to a well-determined choreography everyone knows what they must do in order for a smooth performance without any bumps or clashes. Such a language provides the means to structure the interaction protocols and reason about their correctness. At the level of programming language support for web services we find proposals such as the Web Services Business Process Execution Language (WS-BPEL) [4]. From a system developer point of view, it would be interesting to be able to take a WS-BPEL program and check if it complies with a WS-CDL choreography so as to convey the properties guaranteed by the protocol analysis to the implementation. Similar problems are at the basis of the proposals of Carbone et al. [30] and van der Aalst et al. [91]. The first puts forth a methodology for projecting global interaction specifications, similar to WS-CDL, to local process specifications, so as to guarantee — by construction — that local processes respect the globally defined choreography. The latter builds on a translation of (abstract) BPEL process specifications to Petri nets [82] so as to set the expected global interaction model, which is then used as a reference to be compared with the actual message exchanges (reported in an event log).

A central concern in service-oriented software development may be expressed by the following question: "is this service-oriented system implemented over collaborations which follow well-defined protocols of interaction?". In general, such a question raises difficult analysis problems in such a dynamic and decentralized setting, since it involves accounting for all the possible runtime behaviors of the applications just by looking at their source code. It is an even harder

task — not to say impossible — when the semantics of the programming languages considered are defined in an informal descriptive way, which may leave room for their subjective interpretation and lead compiler developers to produce different behaviors from the same source code.

This informal discussion motivates the general approach of the work reported in this dissertation, which is to develop models equipped with a mathematically defined semantics and formal analysis techniques over such models, in order to provide mechanisms that give answers about critical properties of the developed applications at static verification time. Typically, such models and techniques are not intended to be directly usable in practice, as they have to abstract from many details in order to make the theories tractable and presentable. However, they are intended to serve as a foundational basis for the design of practical programming languages and software development methodologies, so it is important to aim at general, simple and readable ideas in order to allow for them to be conveyed to a more practical setting.

1.1.2 Models of Interaction

The pioneering work of Milner back in the 80's can be referred to as one of the most influential work at the origins of the research field on models of interaction. The Calculus of Communicating Systems (CCS) proposed by Milner in 1980 [74], already presented the basic elements of interaction in a minimal and elegant way, and perhaps more importantly, put forth a mathematical interpretation of these basic ingredients. The CCS model does not comprehend a notion of distribution, as it does not explicitly consider sites nor localities, instead it presents a more abstract model for general concurrent interaction, regardless of the spatial configuration of systems. One of the basic distilled ingredients to specify such concurrent interactions is the parallel composition of processes, denoted in CCS by $\mathit{ProcessA} \mid \mathit{ProcessB}$ which represents that $\mathit{ProcessA}$ and $\mathit{ProcessB}$ are running in parallel or simultaneously. Other ingredient is a communication mechanism that allows for parallel processes to synchronize, namely action capabilities for which there is a notion of duality, e.g., inputs and outputs. In CCS we specify by $\mathit{act}.\mathit{ProcessA}$ a process that is willing to exercise the action act, and after which proceeds in accordance to what is specified by $\mathit{ProcessA}$. Considering $\overline{\mathit{act}}$ is the dual action of act then:

$$\mathit{act}.\mathit{ProcessA} \mid \overline{\mathit{act}}.\mathit{ProcessB}$$

specifies the parallel composition of two processes that are willing to perform dual actions. Hence they can synchronize and evolve to the configuration $\mathit{ProcessA} \mid \mathit{ProcessB}$, an evolution which is usually written with the help of the $\rightarrow$ symbol ($P \rightarrow Q$ means P evolves to Q):

$$\mathit{act}.\mathit{ProcessA} \mid \overline{\mathit{act}}.\mathit{ProcessB} \rightarrow \mathit{ProcessA} \mid \mathit{ProcessB}$$

Another primitive allows to specify alternative behavior, the choice operator. The specification $\mathit{act}.\mathit{ProcessA} + \mathit{otherAct}.\mathit{ProcessB}$ denotes a process that is willing to perform action act and evolve to $\mathit{ProcessA}$ or perform action $\mathit{otherAct}$ and evolve to $\mathit{ProcessB}$. For instance, if placed in a context that is willing to perform the dual of act, we have the following evolution:

$$\overline{\mathit{act}}.\mathit{ProcessC} \mid \mathit{act}.\mathit{ProcessA} + \mathit{otherAct}.\mathit{ProcessB} \rightarrow \mathit{ProcessC} \mid \mathit{ProcessA}$$

Actions in CCS are represented by names and their dual actions by their co-names. Names

can be specified to be known only to a part of the system. For instance, in the following:

$$\mathit{ProcessC} \mid (\boldsymbol{\nu} \mathit{act})(\mathit{act}.\mathit{ProcessA} \mid \overline{\mathit{act}}.\mathit{ProcessB})$$

we specify by $(\boldsymbol{\nu} \mathit{act})$ the hiding of name *act* to the part of the system $\mathit{act}.\mathit{ProcessA} \mid \overline{\mathit{act}}.\mathit{ProcessB}$ making it unaccessible to *ProcessC*. In such way, it is possible to specify some process interactions which are local to a part of the system. In fact, the only way to introduce a notion of spatial configuration in CCS is through name hiding: if some name is known only to a part of the overall system and cannot be forged by other parts, then we can identify a subsystem corresponding to the part of the system where that name is known. Such spatial configurations do not aim at representing physical distribution, being more general and able to capture notions of logical distribution.

The CCS model failed to capture dynamic systems due to the fact that the spatial configuration provided by the name hiding is hardwired. This led to the proposal of the π-Calculus by Milner et al. [79], which extended the CCS with the capability of name passing. By allowing names to be passed around between processes, the π-Calculus model is able to capture the dynamic configuration of systems. For instance, consider the following specification:

$$\mathit{letsTalk}(x).\overline{x}(\text{“}\texttt{hi}\text{”}) \mid (\boldsymbol{\nu} \mathit{privateChat})(\ \overline{\mathit{letsTalk}}(\mathit{privateChat}).\mathit{privateChat}(\mathit{msg})\ )$$

which represents the parallel composition of two processes, where the one on the left hand side is waiting to receive a name through a communication on *letsTalk*, and after which uses this received name to send a text message (“hi”), and the one on the right hand side, that is willing to send a private name on *letsTalk*, and after which waits to receive a text message on that private name. Through a synchronization on *letsChat* the name *privateChat* is passed between the processes, which allows the process on the left to dynamically get to know name *privateChat*:

$$\begin{array}{l}\mathit{letsTalk}(x).\overline{x}(\text{“}\texttt{hi}\text{”}) \mid (\boldsymbol{\nu} \mathit{privateChat})(\ \overline{\mathit{letsTalk}}(\mathit{privateChat}).\mathit{privateChat}(\mathit{msg})\ )\\ \rightarrow\\ (\boldsymbol{\nu} \mathit{privateChat})(\ \overline{\mathit{privateChat}}(\text{“}\texttt{hi}\text{”}) \mid \mathit{privateChat}(\mathit{msg})\ )\end{array}$$

Notice the scope of the name hiding (or name restriction, to be more accurate with π-Calculus terminology) is enlarged as a consequence of the communication (commonly referred to as scope extrusion). This ability to model systems with such dynamic configurations through name mobility lifted the π-Calculus to the status of the reference foundational model for specifying dynamic concurrent systems. It has motivated and influenced countless research efforts on the modeling and analysis of concurrent systems. It has also influenced the design of several prototype programming languages [8, 46, 86, 93, 97].

Among the most popular analysis techniques for programming languages in general we identify type systems. Types provide mechanisms to prevent runtime errors, and are also sometimes useful to extract information about programs that helps reasoning about their runtime behavior. For the π-Calculus in particular there have been many type systems proposals for multiple purposes. The first type system for the π-Calculus was Milner's sorting system [77] which addressed the problem of arity mismatch errors in communication, i.e., when communications cannot take place because the output and input disagree on the length of the tuple being communicated. Afterwards, other type disciplines were proposed to address the consistent usage of names for

input and output introduced by Pierce et al. [84], to impose linearity constraints on the usage of names proposed by Kobayashi et al. [64], just to mention a few. More refined type systems for the π-Calculus have been developed over the years. Examples include proposals where types that characterize the behavior of systems are themselves full-fledged process models [37], others that propose a general framework for π-Calculus type systems [58], yet others have more specialized purposes such as to guarantee deadlock freedom [63], to address secure information flow [56], and to analyze the usage of resources [20, 53, 66].

One prominent proposal for type systems that aim at structuring the interaction of communicating concurrent systems specified in the π-Calculus is that of session types, introduced by Honda [54] and by Honda et al. [55]. Session types characterize the interaction between two parties, typically a client and a server, that play a symmetric sequential interaction game in a dedicated medium of communication. Session types have influenced the development of several proposals targeting many topics in computer science, such as object oriented programming [29, 41], functional programming [94] and security [39]. They have also influenced approaches at a more low-level of application, namely at the level of operating system design [43] and of middleware communication protocols [90].

Session types provide a mechanism to specify and reason about protocols of interaction in a client/server setting. It is thus natural that we find them at the basis of several proposals for modeling and analyzing service-oriented systems (e.g., [12, 13, 68]). However, approaches based on the classical dyadic session types are not able to capture interaction between multiple participants, at least in a typeful way, leaving scenarios such as a service collaboration involving several parties out of their scope. To solve this problem some works have extended session types so as to provide support for multiparty interaction, namely the proposals of Honda et al. [57], of Bettini et al. [9] and of Bonelli et al. [11]. However, such approaches rely on fixed system configurations that do not seem fit to represent the dynamic nature of service collaborations.

In particular, the work of Honda et al. [57] addresses the problem of verifying that all parties in a multiparty session follow a well-defined protocol of interaction, based on a global specification (a global type) that describes the point to point interactions between parties. Such a specification leaves little room for dynamic collaborations as all individual roles in the interaction have to be a priori known. This approach is extended in the work of Bettini et al. [9] but still not in the direction of capturing more dynamic systems. Instead, the focus is on the difficult problem of guaranteeing deadlock freedom of systems where parties are involved in several of such multiparty interactions, based on a notion of ordering of the communication mediums. However, the technique presents limitations at the level of capturing systems where parties dynamically gain access to the communication mediums. Such open problems form the basis of the research questions to which the work presented in this dissertation contributes to answer, namely:

- How do we statically ensure that interactions between distributed participants follow well-defined protocols (protocol safety), without central control?
- How do we ensure that protocol safety is guaranteed, even in challenging scenarios where unanticipated participants join the collaborations dynamically?
- How can we statically verify, just by analyzing the source code (avoiding runtime monitoring) that protocol safety is ensured in such complex scenarios?

- How can we certify that systems where parties are simultaneously involved in several of such complex interaction protocols are free from deadlock?

In the rest of the Introduction we describe the main ideas behind our development and how exactly we contribute to solve the above listed problems.

1.2 Modeling and Analyzing Service-Oriented Systems

While most traditional service-oriented applications can be modeled using the client/server paradigm, since services provided to a final user frequently fit in such description, this does not seem to be the case for general service-oriented systems. As an example, consider a service that implements a whole business process that has to simultaneously cope with electronic purchasing, supply chain management, costumer service interfaces, and so on and so forth. In such a complex application we are most likely to find collaborations between many service providers, for instance at the level of the supply chain management that may rely on the collaboration of external business partners to perform restocking operations or to carry out the shipping of goods. Hence, although a single interaction between service partners is likely to be peer-to-peer, a single protocol may involve several parties that collaborate on a particular task. Clearly, in order for a particular service collaboration to take place, the several partners involved must share a common understanding of the overall protocol of interaction.

The basic ingredients of a service-oriented "soup" are thus a number of software applications that are distributed throughout the network and that are exchanging messages in order to collaborate in a particular task. A key concept for the organization of service-oriented computing systems is this notion of a set of interactions which are related by a common task: a *conversation*. A conversation is a structured, not centrally coordinated, possibly concurrent, set of interactions between several parties. This abstract notion of conversation is at the basis of our development: a single conversation encapsulates the interactions between several service partners that are relative to the same task, in such way allowing to reason and analyze the protocols of a particular task in a dedicated way. Our notion of conversation can be viewed as a generalization of sessions. We elaborate on this point next, by means of a more detailed comparison.

1.2.1 From Binary to Multiparty Interaction

Dyadic sessions capture the interaction between two participants, typically a client and a server, that exchange messages between them in a sequential and symmetric way: if one is about to send a message then the other is ready to receive it and vice-versa. Sessions provide with a means to encapsulate a set of related interactions, and thus serve as a simple mechanism for the specification and analysis of process interaction.

Consider, for example, the message sequence chart shown in Figure 1.1 which illustrates a simple service interaction. A Buyer wants to buy an item from a Seller using service `BuyService` to that end. For starters, Buyer invokes the `BuyService`. After that, Buyer sends to Seller message `buy` which specifies the item he wants to buy, and Seller replies back to Buyer on message `price` informing on the price of the item. Messages `buy` and `price` are related by the same service interaction, instantiated at the moment of service invocation. This scenario can be captured by a session: the interaction starts by the creation of a medium of communication which is shared by the two parties, after which messages can be exchanged in that medium.

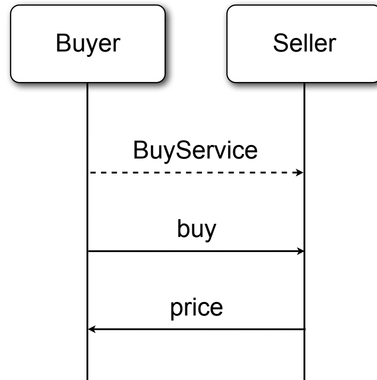

Figure 1.1: Buyer Seller Interaction.

The notion of session is thus particularly adequate to capture the client/server paradigm, however it falls short when it comes to modeling interaction between several participants, a scenario commonly found in real service-oriented applications. The only way to involve several participants in a session is by delegating participation: parties interacting in a session can ask another party to replace them, so while the other party gains access to the session, the delegating party completely loses access to it. Thus, this delegation mechanism does not support multiparty interaction. However, we may ask if a more refined notion of delegation can give support to multiparty interaction: what if delegating parties do not lose access to the interaction medium and may continue interacting in it?

Consider a variation of the previous example shown in Figure 1.2, illustrating a typical service collaboration (adapted from [30]). As in the previous example, Buyer and Seller start a service collaboration through the `BuyService` and exchange messages `buy` and `price`. After that Seller asks Shipper to join in on the ongoing collaboration, by invoking `ShippingService`. Seller then informs Shipper on the product to be delivered by sending message `product`. So, although Seller asked Shipper to join in on the ongoing collaboration, he did not lose access to it. Finally, Shipper directly informs Buyer on the delivery details by sending message `details`. The service conversation as a whole consists in the exchange of messages `buy`, `price`, `product` and `details`, between the three partners Buyer, Seller and Shipper.

This scenario can be modeled by our notion of conversations: for starters, a communication medium is created and shared between Buyer and Seller, supporting message exchanges between the two parties, just like in a session. Then, Shipper is asked to join the ongoing service collaboration and is given access to the shared communication medium, after which messages can be exchanged between the three parties. Thus, access to the conversation medium can be dynamically given to other parties, while delegating parties can keep interacting in it: this can be viewed as a form of partial delegation. This feature is not supported by sessions, and thus distinguishes our conversations from sessions, and allows us to address such challenging scenarios of service collaborations that involve a dynamically determined set of multiple participants, scenarios that fall out of scope of previous session-based approaches.

We describe the relation between conversations and sessions as follows: conversations are a simple extension of the notion of session so as to support multiparty interaction in a single medium of communication. Conversations, like sessions, provide with a means to encapsulate a set of related interactions. However, unlike sessions, the set of interactions in a conversation may

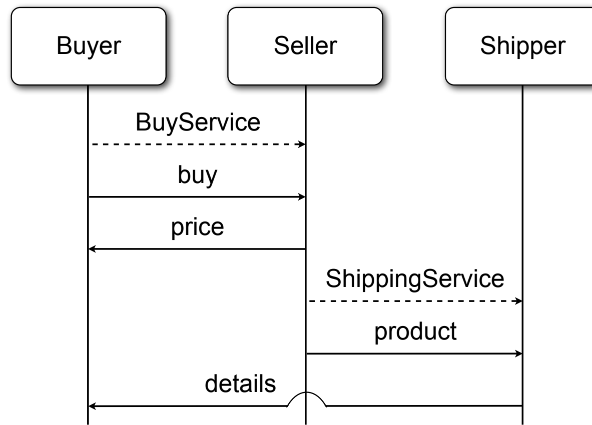

Figure 1.2: Buyer Seller Shipper Interaction.

take place between several different parties. As in sessions, a single interaction in a conversation is binary in the sense that for a single message at a given moment there is at most one party ready to send it and one party waiting to receive it. However, unlike sessions, several parties may be willing to concurrently interact (in distinct messages) in a conversation. Conversations, like sessions, are initially established between two parties. However, unlike sessions, conversations support the dynamic join (and leave) of participants, allowing for the set of participants in a conversation to dynamically increase (or decrease) in number.

This basic understanding of our notion of conversation already allows us to see how they can be useful to model multiparty service collaborations. Before going into how we instantiate these ideas in a concrete model, we take a step back and describe, from an abstract point of view, the key features of service-oriented computing that are addressed in our development. We identify the following key aspects of the service-oriented computational model: *distribution*, process *delegation*, communication *contexts* and *loose coupling*.

1.2.2 Key Aspects of Service-Oriented Computing

Distribution

From the point of view of a service client, the main motivation behind the invocation of a service is the will to have someone else to perform a task in his stead. This leaves the service client free to carry out some other task, allowing to share work load or more importantly to support a notion of specialization in a service collaboration: one partner can concentrate on tasks it can autonomously provide solutions for, and delegate other tasks to partner services. So the purpose of a service call is the delegation of a task to some other partner that will use its own *remote* resources to carry out the desired functionality. This means the relationship between service provider and service client makes particular sense when these are two separate entities, with access to different resources and capabilities — typically running on distinct sites — which implies there is an underlying distributed model.

An essential aspect which we have already identified above is the notion of “remote process delegation”. Remote process delegation differs from the more restricted notion of “remote operation invocation”, as known, e.g., from distributed object systems, or even from earlier client-server or remote procedure call based systems.

Process Delegation versus Operation Invocation

In a distributed computing setting the only communication mechanism actually implementable is message passing, which yields an asynchronous model of interaction. On top of this basic mechanism more sophisticated abstractions may be represented, namely remote procedure call (passing first order data) and remote method invocation (also passing remote object references), which are used in a call/return way. Instead, a service invocation entails the remote execution of a complex task which may require a richer interaction pattern, and may actually be executed by a number of parties. In general, a service client delegates a whole interactive activity (technically, a process) to the service provider. Therefore, we view service invocation at a still higher level of abstraction with respect to remote procedure call and remote method invocation.

As typical examples of sub-activities that may be delegated to external parties we may think of a service to book flights (cf., a travel agency), a service to order goods (cf., a purchasing department), a service to process banking operations (cf., an ATM), a service to store and retrieve documents (cf., an archive), a service to receive and send mail (cf. an expedition service), and so on. The distinguishing feature of service-oriented computing, in our view, is an emphasis on the remote delegation of *interactive processes*, rather than on the remote delegation of individual operations.

The remote process delegation paradigm seems more general than the remote operation invocation paradigm, at least at our current level of description, since one can always see an individual operation as a special case of an interactive process. Conversely, one may argue that the delegation of a given interactive process may always be implemented, at a lower level, by the invocation of individual remote operations. However, such individual remote operations are nevertheless contextually connected by belonging to the same activity, and most likely have to carry around information that explicitly or implicitly relates them. It thus seems that a natural communication abstraction for service-oriented computing is one that is able to accommodate a set of contextually related interactions.

Communication Contexts

A context is a space where computation and communication happens. A context may have a spatial meaning, e.g., as a *site* in a distributed system, but also a behavioral meaning, e.g., as a *context of conversation*, or a *session*, between two or more partners. For example, the same message may appear in two different contexts, with different meanings. In practice, one finds that such messages usually contain information that uniquely relates them to some contextual information — e.g., web-service technology tags messages with correlation tokens that serve as a means to uniquely identify the service conversation to which the messages belong to. Having a means to explicitly represent these contexts seems a convenient mechanism to structure and encapsulate service conversations.

A context is also a natural abstraction to group and publish together closely related services. Typically, services published by the same entity are expected to share common resources, and such sharing may be observed at several scales of granularity. Extreme examples are an object, where the service definitions are the methods and the shared context the object internal state, and an entity such as, e.g., Amazon, that publishes several services for many different purposes which share many of Amazon's internal resources, such as databases, payment gateways, etc..

On the one hand, contexts serve as a means to group related interactions and to group the resources which are shared by some subsystem. On the other hand, contexts introduce boundaries between not so closely related computation and communication. Such boundaries are essential to support the loosely-coupled design of systems.

Loose Coupling

A service-based computation usually consists in an collection of remote partner service instances, in which functionality is to be delegated, some locally implemented processes, and one or more control (or orchestration) processes. The flexibility and openness of a service based design, or at least an aimed feature, results from the loose coupling between these various ingredients. For instance, an orchestration describing a "business process", should be specified in a quite independent way of the particular subsidiary service instances used, paving the way for dynamic binding and dynamic discovery of partner service providers. In the web services description language WSDL [38], loose coupling to external services is enforced, to some extent, by the separate declaration of an abstract level, that describes a service by the messages it sends and receives (a behavioral interface), and of a concrete level which specifies what is the transport and wire format information associated to each service. In such way, services that share an abstract interface can be referred to generically, regardless of lower-level communication details. In the modeling language SRML [45], the binding between service providers and clients is mediated by "wires", which describe plugging constraints between otherwise hard to match interfaces.

To avoid tight coupling of services, the interface between a service instance and the context of instantiation should in general be connected through appropriate mediating processes, in order to hide and/or adapt the service communication protocols (which are in some sense dependent of the particular implementation or service provider chosen) to the abstract behavioral interface expected by the context of instantiation. All computational entities cooperating in a service task should then be encapsulated (delimited inside a conversation context), and able to communicate between themselves and the outer context only via some general message passing mechanism.

We focus on this minimal set of key features of service-oriented computing, leaving out of the scope of our approach other aspects that must be addressed in a full-fledged model of service-oriented computation. The most obvious ones include failure handling and resource disposal, security (in particular access control, authentication and secrecy), time awareness, and a clean mechanism of interoperation. This last aspect seems particularly relevant, and certainly suggests an important evaluation criteria for any service-oriented computation model. Given this basic characterization of the service-oriented setting we are aiming at, we may now present our proposed model for service-oriented computing.

1.2.3 The Conversation Calculus

In this section we present our proposal of a model for service-oriented computing, the Conversation Calculus (CC) [23, 95]. We start by describing the primitives individually, and how they instantiate the previously mentioned aspects of service-oriented computing.

The Conversation Calculus can be seen as a specialization of the π-Calculus for service-oriented computing. For starters, we consider the basic structural operators (borrowed from the π-Calculus and found in most process algebra for concurrent systems): we consider the parallel

composition of processes $P \mid Q$ which denotes processes P and Q are running simultaneously, the choice operator $P + Q$ which expresses a process that either performs as P or performs as Q, the inactive process $\mathbf{0}$ and the name restriction operator $(\boldsymbol{\nu} a)P$ which denotes name a is local to process P. Then, we introduce primitives specific to the service-oriented computational model.

Conversation Context Access

A conversation context is a medium where related interactions can take place. Each context is identified by a unique name (cf., a URI). Thus, to interact in a conversation a process needs only to know the conversation name in order to access the respective conversation context. This allows for processes to access conversations from anywhere in the system. We denote by:

$$c \blacktriangleleft [\ P\ ]$$

a process that is accessing conversation with name c and interacting in it according to what P specifies. A conversation context may actually be distributed in several access points, and processes interact seamlessly between distinct access points of the same conversation context. Intuitively, a conversation context may be seen as a virtual chat room where remote participants exchange messages, while being simultaneously engaged in other conversations.

Potentially, each access point will be placed at a different enclosing context. On the other hand, any such endpoint access will necessarily be placed at a single enclosing context. The relationship between the enclosing context and such an access point may be seen as a call/callee relationship, but where both entities may interact continuously.

Context Awareness

A process interacting in a given conversation context should be able to dynamically access its identity. This capability may be realized by the construct:

$$\textsf{\textbf{this}}(x).P$$

The variable x will be replaced inside the process P by the name of the current context. For instance the process $c \blacktriangleleft [\ \textsf{\textbf{this}}(x).P\ ]$ evolves to $c \blacktriangleleft [\ P\{x/c\}\ ]$. This primitive bears some similarity with the **self** (or **this**) of object-oriented languages, even if it has a different semantics.

Communication

Communication between subsystems is realized by means of message passing. Since several messages may be concurrently exchanged in a single conversation we distinguish messages with labels. We denote the input/output of messages from/to the current conversation context by:

$$\texttt{MessageLabel}^{\downarrow}?(x_1, \ldots, x_n).P \qquad\qquad \texttt{MessageLabel}^{\downarrow}!(v_1, \ldots, v_n).P$$

In the output case, the terms v_i represent message arguments, values to be sent, as expected. In the input case, the variables x_i represent message parameters and are bound in P, as expected. The target symbol $\downarrow$ (read "here") says that the corresponding communication actions must interact in the current conversation context. Second, we denote the input and the output of

messages from/to the outer context by the constructs:

$$\texttt{MessageLabel}^{\uparrow}?(x_1,\ldots,x_n).P \qquad \texttt{MessageLabel}^{\uparrow}!(v_1,\ldots,v_n).P$$

The target symbol ↑ (read "up") says that the corresponding communication actions must interact in the (uniquely determined) outer context, where "outer" is understood relatively to the context where the output/input process is running.

Service-Oriented Idioms

While we do not consider them as primitives of the model (since they can be expressed using the basic communication mechanisms), it is useful to introduce some idioms dedicated to the publication and instantiation of services, as well as an idiom that allows parties to join in on ongoing conversations. A context (which may idiomatically represent a site in our model) may publish one or more service definitions. Service definitions are stateless entities, pretty much as function definitions in a functional programming language. A service definition is expressed by:

$$\textbf{def}\ \texttt{ServiceName} \Rightarrow \mathit{ServiceBody}$$

where `ServiceName` is the service name, and *ServiceBody* is the process that should be executed on the provider side upon service instantiation, or, in other words, the service body. In order to be published, such a definition must be inserted into a context, for instance:

$$\mathit{ServiceProvider} \blacktriangleleft [\ \textbf{def}\ \texttt{ServiceName} \Rightarrow \mathit{ServiceBody}\ \cdots]$$

Such a published service may be instantiated by means of an instantiation idiom:

$$\textbf{new}\ \mathit{ServiceProvider} \cdot \texttt{ServiceName} \Leftarrow \mathit{ClientProtocol}$$

where *ServiceProvider* is the name of the context where the service `ServiceName` is published, and the *ClientProtocol* describes the process that will run on the client side. The outcome of a service instantiation is the creation of a new globally fresh conversation identity (a hidden name), and the creation of two access points to the conversation named by this fresh identity. One will contain the *ServiceBody* process and will be located inside the *ServiceProvider* context. The other will contain the *ClientProtocol* process and will be located in the same context as the **new** expression that requested the service instantiation. These newly created conversation access points appear to their caller contexts as any other local processes, as *ServiceBody* and *ClientProtocol* are able to continuously interact in the calling context by means of ↑ directed messages. As expected, interactions between *ServiceBody* and *ClientProtocol* in the new conversation are realized by means of ↓ directed messages.

Primitives like service publication and instantiation are commonly found in session-based languages, however we do not consider them as native to the underlying model, rather just useful programming idioms that can be programmed using the canonical communication primitives. The **join** idiom we introduce next is new to our approach. In fact, we believe it is not feasible to represent (at least not in a typeful way) in previous session based approaches.

In our model conversation identifiers may be manipulated by processes if needed (accessed

via the $\textsf{\textbf{this}}(x).P$), passed around in messages and subject to scope extrusion: this allows us to model multiparty conversations by the progressive access of multiple, dynamically determined partners, to an ongoing conversation. Joining of another partner to an ongoing conversation is a frequent programming idiom, that may be conveniently abstracted by the:

$$\textsf{\textbf{join}}\ \mathit{ServiceProvider} \cdot \texttt{ServiceName} \Leftarrow \mathit{ContinuationProcess}$$

construct. The **join** and the **new** expression are implemented in a similar way, both relying on name passing. The key difference is that while **new** creates a fresh *new conversation*, **join** allows a service `ServiceName` defined at *ServiceProvider* to join in the *current conversation*, while the calling party continues interacting in the current conversation as specified by *ContinuationProcess*. So, even if the **new** and **join** are represented in a similar way, the abstract notion they realize is actually very different: **new** is used to start a fresh conversation between two parties (e.g., used by a client that instantiates a service) while the **join** is used to allow another service provider to join an ongoing conversation (e.g., used by a participant in a service collaboration to dynamically delegate a task to some remote partner). At a very high level of description the two primitives can be unified as primitives that support the dynamic delegation of tasks (either in a unary conversation or in a n-ary conversation).

1.2.4 Programming in the Conversation Calculus

In this section we provide some more intuition on the meaning of the language primitives by programming some examples. We start by coding the example described in Section 1.2.1, which is simple enough to allow us to describe how the system evolves, so as to illustrate the mechanics of the language, then we present a more complex example which illustrates a collaboration where the number of partners simultaneously involved in the collaboration actually depends on some condition established at runtime. In the examples we assume an extension of the language so as to consider basic values, and a standard **if**—**then**—**else**— statement.

The Purchase Scenario

For a first example of a program in our language let us go back to the interaction described in the message sequence chart of Figure 1.2. For starters we have three parties Buyer, Seller and Shipper, which we model (idiomatically) by considering three distinct conversation contexts named accordingly, so the whole system is specified by the parallel composition of three contexts:

$$\mathit{Buyer} \blacktriangleleft [\ (\cdots)\ ] \mid \mathit{Seller} \blacktriangleleft [\ (\cdots)\ ] \mid \mathit{Shipper} \blacktriangleleft [\ (\cdots)\ ]$$

The Buyer party first invokes the service `BuyService` published by Seller, using the **new** $\mathit{Seller} \cdot \texttt{BuyService} \Leftarrow (\cdots)$ idiom, which specifies the name of the service (`BuyService`) and the context where the service is available at (*Seller*). The service invocation specifies the Buyer's interaction protocol in the service instance: first it sends message `buy`, then it receives message `price` and finally it receives message `details`. The CC code for the Buyer party is:

$$\textsf{\textbf{new}}\ \mathit{Seller} \cdot \texttt{BuyService} \Leftarrow \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)$$

where *prod* is a value that identifies the product (e.g., a string), p is the variable that will be instantiated with the price of the product (e.g., a floating point value) and d is the variable that will be instantiated with the delivery information (e.g., a string). The messages, defined with the ↓ direction, will be exchanged with the other parties collaborating in the service task.

The Seller party publishes the `BuyService` by means of the **def** `BuyService` $\Rightarrow$ $(\cdots)$ idiom. The interaction protocol of the Seller in the service collaboration is such that it first receives message `buy` and then it sends message `price`. In between, we consider that to determine the price there is a consult to a database local to Seller, which is accessed via a mediating process that interacts in the *Seller* conversation. For starters, the Seller's CC code for the `BuyService` specifies:

$$\texttt{buy}^{\downarrow}?(prod).\texttt{askPrice}^{\uparrow}!(prod).\texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p).\,(\cdots)$$

To interact with the database, the service code must access the caller context (the *Seller* context) through ↑ directed messages, so while messages `buy` and `price` are directed to the service conversation (↓) and will be exchanged with the Buyer party, messages `askPrice` and `priceVal` are directed to the enclosing conversation (↑) and will be exchanged with the database interface. We abstract from the specification of the database and consider a process *PriceDB* which is waiting to receive an `askPrice` message, carrying the product identification, and then replies back with a `priceVal` message carrying the price information.

After sending message `price` the Seller invites Shipper to join in on the ongoing conversation. This is supported by the **join** *Shipper* · `DeliveryService` $\Leftarrow$ $(\cdots)$ idiom. After inviting Shipper in, Seller actually gets to interact with Shipper, in the ongoing conversation Shipper has just joined, by sending a `product` message containing the product identification. We may now write the whole specification of the `BuyService` definition.

$$\begin{aligned}\textbf{def}\ \texttt{BuyService} \Rightarrow\ & \texttt{buy}^{\downarrow}?(prod).\texttt{askPrice}^{\uparrow}!(prod).\\ & \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p).\\ & \textbf{join}\ Shipper \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(prod)\end{aligned}$$

Finally we write the code for the Shipper, which publishes `DeliveryService` that specifies in the service code that a `product` message is received and a `details` message is sent:

$$\textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(data)$$

We place the code for the three parties in their respective contexts, along with the *PriceDB* process in the *Seller* context, and obtain the specification of the whole system:

$$\begin{aligned}& Buyer \blacktriangleleft [\ \textbf{new}\ Seller \cdot \texttt{BuyService} \Leftarrow \texttt{buy}^{\downarrow}!(prod).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\\ & \mid\\ & Seller \blacktriangleleft [\ PriceDB \mid\\ & \qquad \textbf{def}\ \texttt{BuyService} \Rightarrow \texttt{buy}^{\downarrow}?(prod).\texttt{askPrice}^{\uparrow}!(prod).\\ & \qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p).\\ & \qquad\qquad \textbf{join}\ Shipper \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(prod)\ ]\\ & \mid\\ & Shipper \blacktriangleleft [\ \textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(data)\ ]\end{aligned}$$

We now illustrate how the system evolves so as to provide some more intuition on the semantics of the primitives of the language. For starters, the `BuyService` is instantiated which results in the creation of a fresh conversation, say *buyChat*, and of two access points to this conversation, one held by Buyer and the other held by Seller. The system thus evolves to:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat}) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{buyChat} \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB} \mid \\
\qquad\qquad \mathit{buyChat} \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\qquad\qquad \textbf{join}\ \mathit{Shipper} \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(\mathit{prod})\ ]\ ]\ ) \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]
\end{array}
$$

At this point message `buy` can be exchanged between Buyer and Seller in conversation *buyChat*, after which the system evolves to:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat}) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{buyChat} \blacktriangleleft [\ \texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB} \mid \\
\qquad\qquad \mathit{buyChat} \blacktriangleleft [\ \texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\qquad\qquad \textbf{join}\ \mathit{Shipper} \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(\mathit{prod})\ ]\ ]\ ) \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]
\end{array}
$$

After that, messages `askPrice` and `priceVal` can be exchanged between the service code and the *PriceDB* process in conversation *Seller*. Then, Seller sends to Buyer the price data in message `price` in conversation *buyChat*. At this point the system has evolved to:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat}) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{buyChat} \blacktriangleleft [\ \texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}' \mid \\
\qquad\qquad \mathit{buyChat} \blacktriangleleft [\ \textbf{join}\ \mathit{Shipper} \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(\mathit{prod})\ ]\ ]\ ) \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]
\end{array}
$$

Now, the Seller participant is willing to invite Shipper to join in on the ongoing conversation *buyChat*. The `DeliveryService` is invoked through the **join** idiom, which gives access to the conversation name to the Shipper, allowing for it to collaborate on the ongoing service task, and

results in the establishment of a third access point to conversation *buyChat*, held by Shipper:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, buyChat) \\
(Buyer \blacktriangleleft [\ \ buyChat \blacktriangleleft [\ \mathtt{details}^{\downarrow}?(d)\ ]\ \ ] \\
\ | \\
Seller \blacktriangleleft [\ \ PriceDB' \mid \\
\qquad\qquad buyChat \blacktriangleleft [\ \mathtt{product}^{\downarrow}!(prod)\ ]\ \ ] \\
\ | \\
Shipper \blacktriangleleft [\ \ buyChat \blacktriangleleft [\ \mathtt{product}^{\downarrow}?(p).\mathtt{details}^{\downarrow}!(data)\ ]\ \ ]\ \ )
\end{array}
$$

At this point messages `product` and `details` can be exchanged between Seller and Shipper, and between Shipper and Buyer, respectively, completing the purchase collaboration.

The Finance Portal Scenario

This second example shows a more complex interaction protocol inspired by the Finance Case Study from the SENSORIA Project [59]. To simplify presentation of the example we abstract away from some details, and focus on the aspects that are most important so as to provide a general idea on the programming flavor supported by the language. We consider a bank portal which is used by clients to mediate operations between them and a bank. Some operations require consult and authorization by bank clerks and bank managers, as is the case of a scenario where the client requests a credit.

We model such a credit request scenario, described as follows: the client invokes a service available in a bank portal and places the request, providing his identification and the desired amount. His request is processed by the portal, which asks a bank clerk to assess the risk of the request. If the risk is high then the request is refused, otherwise the bank manager is consulted to approve the credit. If consulted, the bank manager decides and the client is notified accordingly. Also, if the manager's decision is to approve the credit then an automated service is notified to schedule the money transfer. Then, the automated service is informed by the client on the date of the transfer. The message sequence chart for the service interaction is shown in Figure 1.3.

We start by programming the client, assuming there is an *ClientTerminal* process responsible for interacting with the human client, e.g., by displaying information on a screen.

$$
\begin{array}{l}
Client \blacktriangleleft [\\
\quad ClientTerminal \\
\quad\ | \\
\quad \textbf{new}\ BankPortal \cdot \mathtt{CreditRequest} \Leftarrow \\
\qquad \mathtt{request}^{\downarrow}!(myId, amount). \\
\qquad\quad (\mathtt{requestApproved}^{\downarrow}?().\mathtt{transferDate}^{\downarrow}!(date).\mathtt{approved}^{\uparrow}!() \\
\qquad\quad + \\
\qquad\quad \mathtt{requestDenied}^{\downarrow}?().\mathtt{denied}^{\uparrow}!())\ \ ]
\end{array}
$$

The protocol of the client role in the service is such that he first sends message `request` containing the relevant information, then waits for either one of two messages that inform on the bank's decision: either the request has been approved (signaled by message `requestApproved`) or it has been denied (signaled by message `requestDenied`). In either case the *ClientTerminal*

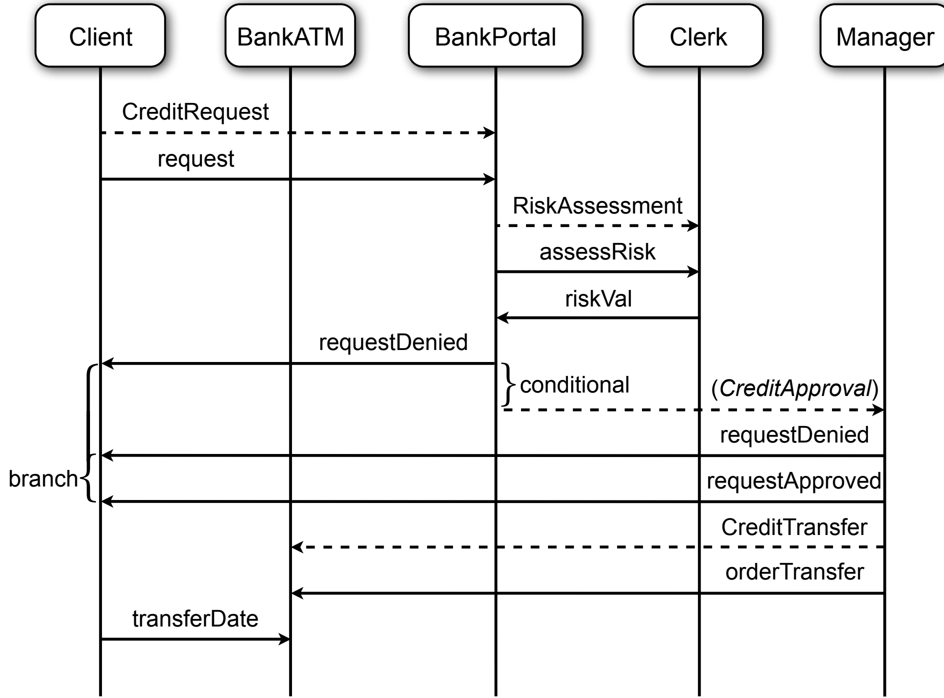

Figure 1.3: Credit Request Interaction.

is informed accordingly, through ↑ directed messages `approved` and `denied`. In the case the request is approved, and before notifying the *ClientTerminal*, message `transferDate` is sent informing on the date when the funds are to be made available.

Next we show the code of the `CreditRequest` service definition available at the *BankPortal* context. To indicate that the service is persistently available we use the notation $\star$ **def**.

$$
\begin{array}{l}
\mathit{BankPortal} \blacktriangleleft [\\
\quad \star\ \textbf{def}\ \texttt{CreditRequest} \Rightarrow\\
\quad\quad \texttt{request}^{\downarrow}?(\mathit{userId}, \mathit{amount}).\\
\quad\quad \textbf{join}\ \mathit{Clerk} \cdot \texttt{RiskAssessment} \Leftarrow\\
\quad\quad\quad \texttt{assessRisk}^{\downarrow}!(\mathit{userId}, \mathit{amount}).\\
\quad\quad\quad \texttt{riskVal}^{\downarrow}?(\mathit{risk}).\\
\quad\quad\quad \textbf{if}\ \mathit{risk}\ =\ \texttt{HIGH}\ \textbf{then}\ \texttt{requestDenied}^{\downarrow}!()\\
\quad\quad\quad \textbf{else}\ \textbf{this}(\mathit{clientChat}).\\
\quad\quad\quad\quad \textbf{new}\ \mathit{Manager} \cdot \texttt{CreditApproval} \Leftarrow\\
\quad\quad\quad\quad\quad \texttt{requestApproval}^{\downarrow}!(\mathit{clientChat}, \mathit{userId}, \mathit{amount}, \mathit{risk})\ \ ]
\end{array}
$$

The service code starts by receiving the request, then proceeds by asking service `RiskAssessment` provided by *Clerk* to join the conversation, sending to the clerk the relevant information in message `assessRisk`. The clerk then replies in message `riskVal` informing on his conclusion (we abstract away from the specification of the `RiskAssessment` service, assuming its interface is completely defined by messages `assessRisk` and `riskVal`). Given the risk information the `CreditRequest` service code then acts accordingly to the risk factor: if it is high then the request is declined and the client is notified (message `requestDenied`); otherwise the manager is asked to approve the credit, and to that end service `CreditApproval` provided by *Manager* is instantiated, and a request for approval is placed. Notice that although service `CreditApproval`

is not asked to join in (instead, a new service instance is created), the manager will further along continue the interaction with the client. To that end the identity of the conversation with the client (*clientChat*) is passed along to the manager (accessed via the **this** primitive).

We now turn to the specification of the `CreditApproval` service, assuming there is a process (*ManagerTerminal*) able to interact with the manager, similarly to the *ClientTerminal* process.

```
Manager ◀ [
    ManagerTerminal
     |
    ⋆ def CreditApproval ⇒
        requestApproval↓?(clientChat, userId, amount, risk).
        this(managerChat).
        showRequest↑!(managerChat, userId, amount, risk).
            (reject↓?().clientChat ◀ [ requestDenied↓!() ]
            +
            accept↓?().clientChat ◀ [
                            requestApproved↓!().
                            join BankATM · CreditTransfer ⇐
                                orderTransfer↓!(userId, amount) ] ]
```

After receiving the request for approval, the `CreditApproval` service definition proceeds by sending to the *ManagerTerminal* process all the pertinent information of the client and of the credit request (by means of $\uparrow$ directed `showRequest` message). The message also carries the identifier for the current conversation so as to allow for the *ManagerTerminal* process to reply back directly to the service conversation. This is similar to a **join**, where an external process is called in to collaborate in the current conversation: in fact, the **join** idiom is implemented by a message exchange which carries a reference to the current conversation. This design is crucial in this case since the service is replicated and thus there might be several instances of the service simultaneously interacting with the *ManagerTerminal* process, making it necessary for the replies to be given directly in their respective contexts.

The reply of the manager consists in one of two possible messages: either an `accept` message denoting the request has been accepted, or a `reject` message otherwise. In the latter case the client is notified accordingly by means of a `requestDenied` message placed directly in the *clientChat* conversation. In the former case not only is the client notified of the decision (also by means of a message placed in the *clientChat* conversation), but also the *BankATM* party is asked to join in on the current conversation — the *clientChat* conversation — through service `CreditTransfer`. The manager completes his task by authorizing the money transfer.

We now specify the code for the *BankATM* party.

```
BankATM ◀ [
    BankATMProcess
    | ⋆ def CreditTransfer ⇒
        orderTransfer↓?(userId, amount).
        transferDate↓?(date).
        scheduleTransfer↑!(userId, amount, date) ]
```

The `CreditTransfer` service code specifies the reception of the transfer order and of the desired date of the transfer, after which forwards the information to a local *BankATMProcess*, which will then schedule the necessary procedure.

This example shows an interesting scenario were not only multiple parties interact in a conversation, but also the number of parties that get to simultaneously interact in the conversation depends on some runtime condition — e.g., only when the manager approves the request is the *BankATM* party asked to join in, otherwise it does not participate in the collaboration.

Such a scenario, where parties dynamically join ongoing conversations and interleave their participation in several conversations, poses difficult problems to verification techniques that address the problem of statically guaranteeing that all participants will follow, at runtime, well-defined protocols and that such protocols never get stuck. In the next section we present the techniques we introduced that cope with such verification problems in such challenging scenarios.

1.2.5 Analyzing Conversations

In this section we present the main ideas behind the analysis techniques we introduced in [27, 28] so as to support the verification of key properties of service-oriented collaborations, namely to provide answers to "do all participants in a multiparty conversation follow a well-defined protocol of interaction?" and "is this system where parties interact in several of such conversations free from deadlock?". Our approach copes with scenarios where multiple parties interact in a conversation, even when some of them are dynamically called in to participate, and where parties interleave their participation in several of such collaborations, even when such interleaving is performed on conversations to which the parties have dynamically joined. Such challenging scenarios are of interest as they can be found in real service-oriented applications, and fall out of scope of previous approaches. Although our technical development focuses on systems specified in the Conversation Calculus, the approach is not specific to this choice of underlying model, as we will later demonstrate.

We introduce two separate but complementary techniques: to discipline multiparty conversations we introduce conversation types, a novel and flexible type structure, able to uniformly describe both the internal and the interface behavior of systems, referred respectively as choreographies and contracts in web-services terminology. To guarantee deadlock freedom we introduce a progress proof system that relies on a notion of ordering of events and, crucially, propagation of orderings in communications. We describe these formalisms next.

Type-Based Analysis of Conversation Fidelity

The starting point for the analysis of the interactions between parties in a distributed system is to have a means to describe the intended protocols of interaction. Usually — for instance in analysis techniques that address security properties — such protocol descriptions take the form of a list of message exchanges, indicating, for each message, who is the sender and who is the receiver, and what sort of information is carried in the message. Works on multiparty session types, namely the approaches of Honda et al. [57] and of Bettini et al. [9], also specify the overall (global) session protocols in such way. For example, consider the following specification of the

protocol of interaction between the Buyer, Seller and Shipper of the purchase example:

1. $\texttt{buy}(Tp) : \mathit{Buyer} \rightarrow \mathit{Seller}$
2. $\texttt{price}(Tm) : \mathit{Seller} \rightarrow \mathit{Buyer}$
3. $\texttt{product}(Tp) : \mathit{Seller} \rightarrow \mathit{Shipper}$
4. $\texttt{details}(Td) : \mathit{Shipper} \rightarrow \mathit{Buyer}$

which describes the purchase interaction as a sequence of messages exchanges that follow a determined order, where the communicating parties involved in each message exchange are identified (e.g., $\mathit{Buyer} \rightarrow \mathit{Seller}$ indicates that the message is sent from Buyer to Seller), and also the contents of the message are described (e.g., Tp represents the type of the content of messages `buy` and `product`).

Using such protocol specifications, we may then look at the implementation and check if the code actually implements the intended protocols. The approach followed by Honda et al. [57] and of Bettini et al. [9] exploits the idea of projecting the global protocol specifications in the individual roles for each participant in the collaboration, allowing for the code for each participant to be checked against its individual role in the collaboration. However, such approaches do not seem fit to capture collaborations that are dynamically established, since the individual roles must be a priori determined. At this point we may pose the following question: if we abstract away from the identities of the communicating parties in the protocol specifications, can we still ensure that protocols will be followed at runtime?

Protocol specifications that do not mention the identifies of the communicating parties seem to be more fit to capture dynamically established collaborations, since we are not confined to an a priori rigid communication structure. On the one hand we are no longer able to certify that the message exchanges take place between some specific parties, on the other hand, however, we are still able to certify that such message exchanges will take place between some parties. This extra flexibility is a first step in the direction to support the dynamic configurations we intend to capture. The conversation type for the purchase interaction is then:

$$\tau\,\texttt{buy}(Tp).\tau\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).\tau\,\texttt{details}(Td)$$

which specifies the sequence of message exchanges as before. Given such a protocol specification, we are then faced with the problem of determining how do we match such protocol with the code that implements it. To that end, we may exploit the notion of projection that splits message exchanges in the dual capabilities that realize the message exchange — output and input — which are the capabilities actually held by the individual parties that form the collaboration. However, if we are to project the overall protocol specification directly to the individual roles of the parties, we also end up confined to a static communication structure.

This leads to a second and crucial step in our development so as to support dynamic configurations: the introduction of a notion of projection that is able to describe the arbitrary decompositions of the protocol in the roles of one or more parties. To that end, conversation types mix, at the same level in the type language, internal/“global” specifications (internal τ message exchanges) with interface/local specifications (output `!` and input `?` types). Building on this capability to mix local and global specifications, we then introduce a (ternary) merge relation between types, noted $B = B_1 \bowtie B_2$, which explains how a behavior B may be projected

in two matching behaviors B_1 and B_2. For example, we have (among others) the following projection for the purchase protocol:

$$\tau\,\texttt{buy}(Tp).\tau\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).\tau\,\texttt{details}(Td)$$
$$=$$
$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td)$$
$$\bowtie$$
$$?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$

which decomposes the overall purchase protocol in types $!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td)$ and $?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).!\,\texttt{details}(Td)$. Notice that the first type actually captures the role of the Buyer party in the purchase collaboration: first it sends message `buy`, then it receives messages `price` and `details`. The second type, which mixes internal and interface types, characterizes the combined roles of the Shipper and the Seller, specifying the dual communication capabilities in the messages that are to be exchanged with the Buyer, and a message exchange internal to the Seller-Shipper subsystem (notice the τ annotation in message `product`). The Seller-Shipper subsystem type may be further decomposed as follows:

$$?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$
$$=$$
$$?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).!\,\texttt{product}(Tp) \quad \bowtie \quad ?\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$

where the decomposition yields the types that characterize the individual roles of the Seller and the Shipper in the purchase collaboration. Notice that, by construction, if we are to merge all such projections back together, then we obtain the initial overall protocol, hence:

$$\tau\,\texttt{buy}(Tp).\tau\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).\tau\,\texttt{details}(Td)$$
$$=$$
$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td)$$
$$\bowtie$$
$$?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).!\,\texttt{product}(Tp)$$
$$\bowtie$$
$$?\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$

These various "local" types are merged by our type system in a compositional way, allowing for the progressive behavioral combination of the individual contributions of each partner, so as to form the overall conversation protocol. Also, and crucially so as to support dynamic join of parties to a conversation, we allow for parties to be initially typed with an arbitrary part of the protocol which may be dynamically (partially) delegated away. For example, this allows us to type the service which implements the Seller role (`startBuy`) in the purchase conversation with the type of the Seller-Shipper subsystem, in such way allowing for the Seller party to dynamically delegate the respective conversation fragment to Shipper upon conversation join.

The types that characterize the conversation fragments that are to be delegated away are obtained from further decompositions (via the merge relation) of the type held by the delegating

party. For instance, when analyzing the `startBuy` service code, at the point where the **join** operation — that calls Shipper in the conversation — gets typed, the (residual) conversation type corresponding to the participation of *Seller* is $\tau\,\texttt{product}(Tp).!\,\texttt{details}(Td)$. At this stage, extrusion of the conversation name to service *Seller* · `newDelivery` will occur, to enable *Shipper* to join in. Notice that the global conversation discipline will nevertheless be respected, since the conversation fragment delegated to *Shipper* is typed $?\,\texttt{product}(Tp).!\,\texttt{details}(Td)$ while the conversation fragment retained by *Seller* is typed $!\,\texttt{product}(Tp)$.

The conversation type system verifies, even in the presence of such decompositions of the overall protocol into fragments — which themselves can be projected in sub-fragments — that may be dynamically delegated away by some parties to others, that the overall protocol is nevertheless met. Notice that since conversation types abstract away from participant identities, the overall conversation type can be projected into the types of the individual roles in several ways, allowing for different implementations of the roles of a given conversation (cf. loose-coupling). It is even possible to type systems with an unbounded number of different participants, as needed to type, e.g., a service broker.

Our type system combines techniques from linear, behavioral, session and spatial types (see [20, 55, 58, 64]): the type structure features prefix $M.B$, parallel composition $B_1 \mid B_2$, and other operators. Messages M describe external (receive ? / send !) exchanges in two views: with the *caller/parent* conversation ($\uparrow$), and in the *current* conversation ($\downarrow$). They also describe internal message exchanges (τ). Key technical ingredients in our approach to conversation types are the amalgamation of global types and of local types (in the general sense of [57]) in the same type language, and the definition of a merge relation ensuring, by construction, that participants typed by the projected views of a type will behave well under composition. Merge subsumes duality, in the sense that for each τ-free B there are types $\overline{B}, B'$ such that $B \bowtie \overline{B} = \tau(B')$, so sessions, that build on the notion of duality, are special cases of conversations. But merge of types allows for extra flexibility on the manipulation of projections of conversation types, in an open-ended way, as illustrated above. In particular, our approach allows fragments of a conversation type (e.g., a choreography) to be dynamically distributed among participants, while statically ensuring that interactions follow the prescribed discipline.

Analysis of Conversation Progress

We motivate our development with an example. Consider the following specification:

$$Amazon \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(product).\texttt{price}^{\downarrow}!(price).eBay \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(product).\texttt{price}^{\downarrow}?(p)\ \ ]\ ]$$

representing an application that is trying to sell some product at *Amazon* and then uses *eBay* to restock the bought item. The code specifies that in conversation *Amazon* a message `buy` is received specifying the product that is to be sold, and then a message `price` is sent indicating the price value. After that, conversation *eBay* is accessed and message `buy` is sent to some seller specifying the item which is to be restocked, and a `price` message is received carrying the price value. Let us consider this application is running in a context where there is another application trying to perform a similar task, specified as follows:

$$eBay \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(product).\texttt{price}^{\downarrow}!(price).Amazon \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(product).\texttt{price}^{\downarrow}?(p)\ \ ]\ ]$$

In fact, the functionality is exactly the same and the only difference is that it is working the other way around: selling at *eBay* and restocking at *Amazon*. When considering the system obtained by composing these two processes in parallel:

$$\begin{array}{l} \mathit{Amazon} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}?(\mathit{product}).\mathtt{price}^{\downarrow}!(\mathit{price}).\mathit{eBay} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}!(\mathit{product}).\mathtt{price}^{\downarrow}?(p)\ \ ]\ ] \\ \mid \\ \mathit{eBay} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}?(\mathit{product}).\mathtt{price}^{\downarrow}!(\mathit{price}).\mathit{Amazon} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}!(\mathit{product}).\mathtt{price}^{\downarrow}?(p)\ \ ]\ ] \end{array}$$

we can observe that the system is deadlocked since both processes are waiting to receive a message. However, well-defined conversation protocols are followed, which can be captured by the fact they respect the type $\tau\,\mathtt{buy}^{\downarrow}(Tp).\tau\,\mathtt{price}^{\downarrow}(Tm)$ in each conversation (in this case the conversation type is the same for both *Amazon* and *eBay*), but the processes interact in the conversations in inverse order. In this example the problem can be detected by a human eye, but if there were several lines of code in between the message exchanges perhaps even a simple dependency such as this one can be overlooked by a human analysis. Formal analysis techniques can help to provide answers to such verification problems.

Traditionally, approaches that address this problem determine if the events of the system can be ordered in a well-founded way. Lynch [73] introduces one of the first of such approaches, formalizing a notion of ordering to prevent deadlocks of systems that access shared resources in an exclusive way (inspired by Dijkstra's dining philosophers scenario [42]). Approaches that address systems specified in the π-Calculus have been introduced more recently, namely the work of Kobayashi [63] and, for session types in particular, the works of Dezani-Ciancaglini et al. [40] and Bettini et al. [9].

Let us go back to the deadlock example so as to see how an event ordering analysis can identify the problem. Events in our setting are message exchanges in conversations, and thus, events are identified by the conversation name and the message label. For the code:

$$\mathit{Amazon} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}?(\mathit{product}).\mathtt{price}^{\downarrow}!(\mathit{price}).\mathit{eBay} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}!(\mathit{product}).\mathtt{price}^{\downarrow}?(p)\ \ ]\ ]$$

we have that the underlying event ordering is such that event $\mathit{Amazon}.\mathtt{buy}$ is smaller than $\mathit{Amazon}.\mathtt{price}$, and so forth, which we denote by:

$$\mathit{Amazon}.\mathtt{buy} \prec \mathit{Amazon}.\mathtt{price} \prec \mathit{eBay}.\mathtt{buy} \prec \mathit{eBay}.\mathtt{price}$$

Instead for the code:

$$\mathit{eBay} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}?(\mathit{product}).\mathtt{price}^{\downarrow}!(\mathit{price}).\mathit{Amazon} \blacktriangleleft [\ \mathtt{buy}^{\downarrow}!(\mathit{product}).\mathtt{price}^{\downarrow}?(p)\ \ ]\ ]$$

the event ordering is such that $\mathit{eBay}.\mathtt{buy} \prec \mathit{eBay}.\mathtt{price} \prec \mathit{Amazon}.\mathtt{buy} \prec \mathit{Amazon}.\mathtt{price}$. Thus, to satisfy the requirements of both processes, an ordering would have to be such that:

$$\mathit{Amazon}.\mathtt{buy} \prec \ldots \prec \mathit{eBay}.\mathtt{buy} \prec \ldots \prec \mathit{Amazon}.\mathtt{buy} \prec \ldots$$

which is not well-founded. In fact there is no well-founded ordering for the parallel composition of the two processes given above. Similar reasoning could be produced for the approaches of [9, 40]. However, these works suffer some limitations that do not allow them to address

scenarios where references to the (ordered) communication mediums are dynamically passed along in communications. We now present how we address this issue. Consider a variation of the code above:

$$\begin{array}{l} \mathit{eBayReseller} \blacktriangleleft [\\ \quad \texttt{sellAt}^{\downarrow}?(x). \\ \qquad x \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{product}).\texttt{price}^{\downarrow}!(\mathit{price}).\mathit{eBay} \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{product}).\texttt{price}^{\downarrow}?(p)\ ]\ ]\ ] \end{array}$$

that specifies a purchase broker *eBayReseller* that performs the previously described functionality: sell in a conversation, restock in another. However, the conversation in which the broker is to sell at is now instructed by a user of the broker, by means of message `sellAt`, while the restocking is performed at *eBay*. The code for a similar broker that restocks at *Amazon* is:

$$\begin{array}{l} \mathit{AmazonReseller} \blacktriangleleft [\\ \quad \texttt{sellAt}^{\downarrow}?(x). \\ \qquad x \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{product}).\texttt{price}^{\downarrow}!(\mathit{price}).\mathit{Amazon} \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{product}).\texttt{price}^{\downarrow}?(p)\ ]\ ]\ ] \end{array}$$

Just by looking at the source code the deadlock is not evident, since it depends on the conversations where the brokers will sell at, namely if placed in the parallel with the process:

$$\mathit{eBayReseller} \blacktriangleleft [\ \texttt{sellAt}^{\downarrow}!(\mathit{Amazon})\ ]\ \mid\ \mathit{AmazonReseller} \blacktriangleleft [\ \texttt{sellAt}^{\downarrow}!(\mathit{eBay})\ ]$$

then the system will end up in a deadlocked configuration similar to the one shown before.

The problem in this example can only be detected if we analyze how the conversation references being passed along must be ordered. The event ordering for the *eBayReseller* must be such that $x.\texttt{buy} \prec x.\texttt{price} \prec \mathit{eBay}.\texttt{buy} \prec \mathit{eBay}.\texttt{price}$ so any name instantiation of x must respect this ordering. Likewise for *AmazonReseller* we have the following prescribed ordering: $x.\texttt{buy} \prec x.\texttt{price} \prec \mathit{Amazon}.\texttt{buy} \prec \mathit{Amazon}.\texttt{price}$. Technically, we proceed by attaching such orderings to the events where the conversation references are passed, which is to say:

$$\mathit{eBayReseller}.\texttt{sellAt}.(x)(x.\texttt{buy} \prec x.\texttt{price} \prec \mathit{eBay}.\texttt{buy} \prec \mathit{eBay}.\texttt{price})$$

and:

$$\mathit{AmazonReseller}.\texttt{sellAt}.(x)(x.\texttt{buy} \prec x.\texttt{price} \prec \mathit{Amazon}.\texttt{buy} \prec \mathit{Amazon}.\texttt{price})$$

which then allows us to check if the name that is actually sent in such a message respects (or not) the ordering expected by the process that receives the name. We may thus exclude such system with our technique, since there is no such well-founded ordering for the events in the system: name *Amazon* is sent in event $\mathit{eBayReseller}.\texttt{sellAt}$ where a conversation x is expected such that $x.\texttt{buy} \prec \ldots \prec \mathit{eBay}.\texttt{buy}$, and name *eBay* is sent in event $\mathit{AmazonReseller}.\texttt{sellAt}$ where a conversation x is expected such that $x.\texttt{buy} \prec \ldots \prec \mathit{Amazon}.\texttt{buy}$.

Our analysis techniques thus allow us to address challenging scenarios where conversation references are passed around, allowing for parties to dynamically join conversations in a typesafe way, and where access to conversations can be interleaved in an orderly fashion, even when such interleaving is performed over conversations to which parties have dynamically gained access to.

Such scenarios fall out of scope of previous techniques, and seem to be crucial, for instance, to allow for services discovered dynamically to participate on an ongoing service task, and to allow for parties to interleave dynamically received conversations so as to access local resources or call on external services to join ongoing conversations. We establish decidability of our techniques (if bound names are type/ordering annotated) as we rely on proof trees (as usual) to witness well-typedness and well-ordering of systems, and the auxiliary operations involved are finitary.

1.3 Contributions and Structure of the Dissertation

1.3.1 Contributions

The contribution of this work is the detailed study of a core semantic model for expressing concurrent interacting systems, particularly suited for modeling service based systems (introduced in [95]). Using such model we present analysis techniques that provide answers to crucial properties of systems, namely conversation fidelity — all conversation participants follow the prescribed protocols — and progress — systems never incur in a deadlocked state — even when conversations are dynamically established and parties interleave their participation in several of such conversations (introduced in [27, 28]), scenarios that fall out of scope of previous approaches. We list our contributions in more detail:

- We introduce the Conversation Calculus [23, 95] (CC), which is essentially a specialized π-Calculus with a communication mechanism based on labeled message passing within conversation contexts, and develop its basic behavioral theory [69, 95]. We prove that the behavioral semantics, defined over a standard notion of strong bisimulation, is a congruence for all operators of the language and show interesting behavioral identities which help envision the abstract semantic model of the language. In fact, such identities led us to the establishment of a normal form result which clarifies the abstract communication model of Conversation Calculus systems (such results were originally presented in [95]).
- We show that the (core) Conversation Calculus language may easily encode higher level abstractions for describing service instantiation, service definition, and dynamical joining of participants to ongoing conversations — while the service definition and the service instantiation were already presented in [95] we introduced them as idioms in [27, 28] together with the conversation join which is unique to our approach; such encodings turn out to smoothly admit the typings intended for the higher level constructs.
- We define and formalize the notion of conversation type [2, 27, 28]. Conversation types are a powerful generalization of session types to loosely coupled, possibly concurrent, multiparty conversations, allowing mixed global/local behavioral descriptions to be expressed at the same level in a friendly and readable way, while allowing systems with dynamic delegation of fragments of ongoing conversations to be analyzed.
- We propose a type system for assigning conversation types to core CC systems [27, 28]. We prove subject reduction and error freedom results which ensure that processes that get past our typing rules are free from communication errors, and races on plain messages, which implies that well-typed systems enjoy a conversation fidelity property.

- We present techniques to establish progress of systems with several interleaved conversations based on a notion of event ordering [27, 28], exploiting the combination of conversation names with message labels and, crucially, propagation of orderings in communications which, in particular, allows us to solve a previously open problem.
- We propose a proof system that singles out well-ordered core CC systems [27, 28] and prove results that ensure that the well-ordered condition is preserved under process reduction and that well-ordered systems are lock-free, which implies well-ordered systems enjoy a progress property.
- We show that our analysis techniques are not particular to Conversation Calculus systems and can be applied over more canonical models, while essentially retaining their expressiveness — originally presented in [28].
- Finally, we present an extension of the Conversation Calculus with exception handling primitives [95] and show how the basic behavioral theory for the core CC smoothly crosses over to this setting. Also, we show how we can use such language extension so as to give support for long-running transactions, namely by encoding compensations [24, 44].

The core of the aforementioned contributions was published in the following articles:

[95] H. Vieira, L. Caires, and J. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In S. Drossopoulou, editor, *ESOP 2008, 17th European Symposium on Programming, Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2008.

[24] L. Caires, C. Ferreira, and H. Vieira. A Process Calculus Analysis of Compensations. In C. Kaklamanis and F. Nielson, editors, *TGC 2008, Fourth International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 87–103. Springer-Verlag, 2009.

[27] L. Caires and H. Vieira. Conversation Types. In G. Castagna, editor, *ESOP 2009, 18th European Symposium on Programming, Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer-Verlag, 2009.

[28] L. Caires and H. Vieira. Conversation Types. *Theoretical Computer Science.* To appear.

1.3.2 Structure of the Dissertation

This dissertation is structured so as to gradually introduce the concepts to the reader. To that end, a first part of the thesis makes use of a simplified setting: we present a basic model and introduce the main artifacts of the conversation type system over such basic model. In such way, not only do we gradually introduce the concepts involved, but also we demonstrate that the conversation type system is not particular to our proposed model and can be used in a more canonical setting. Then, in the second part of the thesis, we introduce the Conversation Calculus and present the analysis techniques developed over it: the conversation type system and the progress proof system — we also show how the progress proof system can be directly applied in the canonical setting. A final part is dedicated to an extension of the model and its

applications. At the end of each chapter we present detailed comparison with related work and make some general remarks on our development.

- Chapter 2 introduces the simple labeled π-Calculus and the basic mathematics used to characterize the mechanics of the calculus. Readers familiar with the π-Calculus may skim through this chapter, as no new concepts are introduced here, just a simple extension.
- Chapter 3 introduces the main ideas and artifacts behind conversation types and companion type system, addressing systems specified in the labeled π-Calculus, which allow us to single out systems that enjoy the conversation fidelity property.
- Chapter 4 introduces the (core) Conversation Calculus and presents a study of the basic behavioral theory of the model. We show how the model easily accommodates higher-level service-oriented primitives as mere idioms of the language, and program some examples.
- Chapter 5 revisits and extends the conversation type language and type system presented in Chapter 3, aiming at the analysis of systems specified in the Conversation Calculus.
- Chapter 6 presents the technical artifacts and the proof system that allow us to single out systems specified in the Conversation Calculus that enjoy the progress property. Also, we show how the system can be directly applied to systems specified to the labeled π-Calculus.
- Chapter 7 presents an extension of the model considering exception handling primitives, and presents some applications of this extension, namely we show how to implement compensations and thus give support for long-running transactions in our model.
- Chapter 8 presents some final remarks.

Chapter 2

Preliminaries

In this chapter, we present the labeled π-Calculus (π_{lab}-Calculus), a basic extension of the π-Calculus, so as to introduce to the unfamiliar reader with π-Calculus like specifications and some mathematics used in their characterization. This calculus will then be used as the underlying model in a successive chapter to introduce conversation types, before turning to the analysis over Conversation Calculus specifications, in such way allowing for a more incremental introduction of the concepts involved. Furthermore, using the π_{lab}-Calculus as an underlying model for our techniques also serves as a proof of concept that the main ideas of our approach are not specific to the Conversation Calculus and may be directed to more canonical models.

We first introduce the syntax of the language, along with auxiliary notions and conventions which help to simplify presentation, then we define the operational semantics of the language by means of a labelled transition system, and show how behaviors can be formally derived through an example. Finally we program the purchase scenario in the π_{lab}-Calculus. Before ending the chapter we make some general remarks and point to some reference literature.

2.1 π_{lab}-Calculus: π-Calculus with Label Indexed Channels

We start by defining the syntax of the π_{lab}-Calculus, an extension of a core π-Calculus [79] where channel communication is indexed with labels. We assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, an infinite set of labels $\mathcal{L}$, and an infinite set of recursion variables χ.

Definition 2.1.1 (π_{lab}-Calculus Syntax)
The syntax of π_{lab}-Calculus processes is given in Figure 2.1.

The static fragment is defined by the inaction $\mathbf{0}$, which represents the inactive process, parallel composition $P \mid Q$, which represents that processes P and Q are running simultaneously, name restriction $(\boldsymbol{\nu} a)P$, which represents that name a is local to process P, and recursion $\mathbf{rec}\,\mathcal{X}.P$, which represents a process that repeatedly executes P. Communication is expressed by the output and input primitives $n \cdot l!(m_1, \ldots, m_k).P$ and $n \cdot l?(x_1, \ldots, x_k).P$, respectively, where $m_1, \ldots, m_k$ are the names to be sent, and variables $x_1, \ldots, x_k$ are the variables to be instantiated with the received names (we consider $x_1, \ldots, x_k$ must be pairwise distinct). Communication is thus defined by both the channel name n and the label l, being the labeling the only difference between the π_{lab}-Calculus language and the π-Calculus. Notice that labels (from $l \in \mathcal{L}$) are not names but free identifiers (cf. record labels or XML tags), and therefore are not subject to fresh

$$
\begin{array}{llll}
a, b, c, \ldots & \in & \Lambda & \text{(Names)} \\
x, y, z, \ldots & \in & \mathcal{V} & \text{(Variables)} \\
n, m, o, u, v \ldots & \in & \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
l, s \ldots & \in & \mathcal{L} & \text{(Labels)} \\
\mathcal{X}, \mathcal{Y}, \ldots & \in & \chi & \text{(Recursion Variables)}
\end{array}
$$

$$
\begin{array}{lcll}
P, Q, R & ::= & \mathbf{0} & \text{(Inaction)} \\
& \mid & P \mid Q & \text{(Parallel Composition)} \\
& \mid & (\boldsymbol{\nu} a)P & \text{(Name Restriction)} \\
& \mid & \mathbf{rec}\, \mathcal{X}.P & \text{(Recursion)} \\
& \mid & \mathcal{X} & \text{(Variable)} \\
& \mid & n \cdot l!(m_1, \ldots, m_k).P & \text{(Output)} \\
& \mid & n \cdot l?(x_1, \ldots, x_k).P & \text{(Input)}
\end{array}
$$

Figure 2.1: The π_{lab}-Calculus Syntax.

generation, restriction or binding. Also, labels are not communicated in messages. Only channel names may be subject to binding, freshly generated via $(\boldsymbol{\nu} a)P$, and be sent in messages.

We introduce some syntactic conventions, which define operator precedence, and commonly used abbreviations, useful to lighten notation and simplify presentation of the examples.

Syntactic conventions:

$\mathbf{rec}\, \mathcal{X}.P \mid Q$ is to be read as $(\mathbf{rec}\, \mathcal{X}.P) \mid Q$.

$(\boldsymbol{\nu} a)P \mid Q$ is to be read as $((\boldsymbol{\nu} a)P) \mid Q$.

$n \cdot l!(\tilde{m}).P \mid Q$ is to be read as $(n \cdot l!(\tilde{m}).P) \mid Q$.

$n \cdot l?(\tilde{x}).P \mid Q$ is to be read as $(n \cdot l?(\tilde{x}).P) \mid Q$.

Abbreviations:

$(\boldsymbol{\nu} \tilde{a})P$ and $(\boldsymbol{\nu} a_1, \ldots, a_k)P$ stand for $(\boldsymbol{\nu} a_1) \ldots (\boldsymbol{\nu} a_k)P$.

$n \cdot l!(\tilde{m}).P$ stands for $n \cdot l!(m_1, \ldots, m_k).P$.

$n \cdot l?(\tilde{x}).P$ stands for $n \cdot l?(x_1, \ldots, x_k).P$.

$n \cdot l!(\tilde{m})$ stands for $n \cdot l!(\tilde{m}).\mathbf{0}$.

$n \cdot l?(\tilde{x})$ stands for $n \cdot l?(\tilde{x}).\mathbf{0}$.

Before introducing a mathematical interpretation of the constructs of the π_{lab}-Calculus we first define some auxiliary syntactical notions, namely: the set of free names, of free labels and of bound names of a process, and the notion of application of a name substitution to a process, that are necessary to manipulate the syntax of processes. The distinguished occurrences of a, $\tilde{x}$ and $\mathcal{X}$ are binding occurrences in $(\boldsymbol{\nu} a)P$, $l^d?(\tilde{x}).P$ and $\mathbf{rec}\, \mathcal{X}.P$, respectively. We define the sets of free ($fn(P)$) and bound ($bn(P)$) names and the set of free labels ($fl(P)$) of a process (P). For the set of free names, since we consider a separate set of variables $\mathcal{V}$ for input prefix parameters,

we need only to collect the names from Λ which are not under the scope of a name restriction, being such set of restricted names the object collected by the set of bound names.

Definition 2.1.2 (Free Names)
The set of free names of a process P, noted $fn(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{lll@{\qquad}lll}
fn(\mathbf{0}) & \triangleq & \emptyset & fn((\boldsymbol{\nu} a)P) & \triangleq & fn(P) \setminus \{a\} \\
fn(P \mid Q) & \triangleq & fn(P) \cup fn(Q) & fn(n \cdot l!(\tilde{m}).P) & \triangleq & ((\{n\} \cup \tilde{m}) \cap \Lambda) \cup fn(P) \\
fn(\mathcal{X}) & \triangleq & \emptyset & fn(n \cdot l?(\tilde{x}).P) & \triangleq & (\{n\} \cap \Lambda) \cup fn(P) \\
fn(\textsf{rec}\, \mathcal{X}.P) & \triangleq & fn(P) & & &
\end{array}
$$

Definition 2.1.3 (Free Labels)
The set of free labels of a process P, noted $fl(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{lll@{\qquad}lll}
fl(\mathbf{0}) & \triangleq & \emptyset & fl((\boldsymbol{\nu} a)P) & \triangleq & fl(P) \\
fl(P \mid Q) & \triangleq & fl(P) \cup fl(Q) & fl(n \cdot l!(\tilde{m}).P) & \triangleq & \{l\} \cup fl(P) \\
fl(\mathcal{X}) & \triangleq & \emptyset & fl(n \cdot l?(\tilde{x}).P) & \triangleq & \{l\} \cup fl(P) \\
fl(\textsf{rec}\, \mathcal{X}.P) & \triangleq & fl(P) & & &
\end{array}
$$

Definition 2.1.4 (Bound Names)
The set of bound names of a process P, noted $bn(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{lll@{\qquad}lll}
bn(\mathbf{0}) & \triangleq & \emptyset & bn((\boldsymbol{\nu} a)P) & \triangleq & bn(P) \cup \{a\} \\
bn(P \mid Q) & \triangleq & bn(P) \cup bn(Q) & bn(n \cdot l!(\tilde{m}).P) & \triangleq & bn(P) \\
bn(\mathcal{X}) & \triangleq & \emptyset & bn(n \cdot l?(\tilde{x}).P) & \triangleq & bn(P) \\
bn(\textsf{rec}\, \mathcal{X}.P) & \triangleq & bn(P) & & &
\end{array}
$$

Next we define the set of free variables ($fv(P)$) of a process (P), hence the set of variable identifiers occurring not within the scope of an input prefix. Also we define the set of free recursion variables ($frv(P)$) of a process (P), hence the set of recursion variables which do not occur in the scope of a binding recursive process construct.

Definition 2.1.5 (Free Variables)
The set of free variables of a process P, noted $fv(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{lll@{\qquad}lll}
fv(\mathbf{0}) & \triangleq & \emptyset & fv((\boldsymbol{\nu} a)P) & \triangleq & fv(P) \\
fv(P \mid Q) & \triangleq & fv(P) \cup fv(Q) & fv(n \cdot l!(\tilde{m}).P) & \triangleq & ((\{n\} \cup \tilde{m}) \cap \mathcal{V}) \cup fv(P) \\
fv(\mathcal{X}) & \triangleq & \emptyset & fv(n \cdot l?(\tilde{x}).P) & \triangleq & (\{n\} \cap \mathcal{V}) \cup (fv(P) \setminus \tilde{x}) \\
fv(\textsf{rec}\, \mathcal{X}.P) & \triangleq & fv(P) & & &
\end{array}
$$

Definition 2.1.6 (Free Recursion Variables)
The set of free recursion variables of a process P, noted $frv(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{lll@{\qquad}lll}
frv(\mathbf{0}) & \triangleq & \emptyset & frv((\boldsymbol{\nu} a)P) & \triangleq & frv(P) \\
frv(P \mid Q) & \triangleq & frv(P) \cup frv(Q) & frv(n \cdot l!(\tilde{m}).P) & \triangleq & frv(P) \\
frv(\mathcal{X}) & \triangleq & \{\mathcal{X}\} & frv(n \cdot l?(\tilde{x}).P) & \triangleq & frv(P) \\
frv(\mathsf{rec}\,\mathcal{X}.P) & \triangleq & frv(P) \setminus \{\mathcal{X}\} & & &
\end{array}
$$

We say P is a closed process if it has no free variables. We often use the term process to refer to closed processes.

Definition 2.1.7 (Closed Processes)
We say P is a closed process if $fv(P) = \emptyset$ and $frv(P) = \emptyset$.

Convention 2.1.8 *We use the term process to refer to closed processes, where appropriate.*

We define the application of a substitution to a process, noted $P\{\tilde{n}/\tilde{m}\}$, which replaces all free occurrences of the n_i identifiers by the respective m_i identifiers. First, we introduce an auxiliary predicate used to indicate two sets are disjoint.

Definition 2.1.9 (Set Disjointness)
We denote by $A \# B$ that A and B are disjoint sets, hence $A \cap B = \emptyset$.

Definition 2.1.10 (Substitution)
Given identifiers $n_1, \ldots, n_k$ and $m_1, \ldots, m_k$ the application of a substitution to a process, noted $P\{n_1, \ldots, n_k/m_1, \ldots, m_k\}$ or $P\{\tilde{n}/\tilde{m}\}$, is inductively defined on the structure of processes as:

$$
\begin{array}{lll@{\quad}l}
\mathbf{0}\{\tilde{n}/\tilde{m}\} & \triangleq & \mathbf{0} & \\
(P \mid Q)\{\tilde{n}/\tilde{m}\} & \triangleq & P\{\tilde{n}/\tilde{m}\} \mid Q\{\tilde{n}/\tilde{m}\} & \\
\mathcal{X}\{\tilde{n}/\tilde{m}\} & \triangleq & \mathcal{X} & \\
\mathsf{rec}\,\mathcal{X}.P\{\tilde{n}/\tilde{m}\} & \triangleq & \mathsf{rec}\,\mathcal{X}.(P\{\tilde{n}/\tilde{m}\}) & \\
((\boldsymbol{\nu} a)P)\{\tilde{n}/\tilde{m}\} & \triangleq & (\boldsymbol{\nu} a)(P\{\tilde{n}/\tilde{m}\}) & (\textit{if } a \notin \tilde{n}, \tilde{m}) \\
(n_i \cdot l?(\tilde{x}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & m_i \cdot l?(\tilde{x}).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if } \tilde{x} \# \tilde{n}, \tilde{m}) \\
(u \cdot l?(\tilde{x}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & u \cdot l?(\tilde{x}).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if } \tilde{x} \# \tilde{n}, \tilde{m} \textit{ and } u \notin \tilde{n}) \\
(n_i \cdot l!(\tilde{o}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & m_i \cdot l!(\tilde{v}).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if } o_j = n_i \textit{ then } v_j = m_i \textit{ else } v_j = o_j) \\
(u \cdot l!(\tilde{o}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & u \cdot l!(\tilde{v}).(P\{\tilde{n}/\tilde{m}\}) & \\
& & \multicolumn{2}{r}{(\textit{if } u \notin \tilde{n} \textit{ and if } o_j = n_i \textit{ then } v_j = m_i \textit{ else } v_j = o_j)}
\end{array}
$$

N.B. By $\tilde{n}, \tilde{m}$ we denote the union of $\tilde{n}$ and $\tilde{m}$, and by i we denote an index such that $i \in 1, \ldots, k$.

We introduce *α-equivalence* which identifies processes which are equivalent up to a (safe) renaming of bound names and variables.

Definition 2.1.11 (α-Equivalence)
α-equivalence, noted $\equiv_\alpha$, is the least congruence on processes that satisfies the following rules:

$$\begin{array}{rcll} (\boldsymbol{\nu} a)P & \equiv_\alpha & (\boldsymbol{\nu} b)(P\{a/b\}) \;\; (b \notin \mathit{fn}(P)) & (\mathit{Restriction}) \\ n \cdot l?(\tilde{x}).P & \equiv_\alpha & n \cdot l?(\tilde{y}).(P\{\tilde{x}/\tilde{y}\}) \;\; (\tilde{y} \# \mathit{fv}(P)) & (\mathit{Input}) \end{array}$$

Processes which are α-equivalent represent the same specification, since they only differ in some irrelevant bound name identities, so we implicitly identify them to simplify presentation.

Convention 2.1.12 *We implicitly identify α-equivalent processes.*

Recursive processes repeat their behavior at the point where the recursion variable occurs. To capture this notion syntactically we define substitution of recursive variables by a process.

Definition 2.1.13 (Recursion Variable Substitution)
Given recursion variable $\mathcal{X}$ and process P the application of a recursion variable substitution to a process in Q, noted $Q\{\mathcal{X}/P\}$, is inductively defined on the structure of processes as follows:

$$\begin{array}{lcllcl} (Q \mid R)\{\mathcal{X}/P\} & \triangleq & Q\{\mathcal{X}/P\} \mid R\{\mathcal{X}/P\} & \mathbf{0}\{\mathcal{X}/P\} & \triangleq & \mathbf{0} \\ \mathcal{Y}\{\mathcal{X}/P\} & \triangleq & \mathcal{Y} & \mathcal{X}\{\mathcal{X}/P\} & \triangleq & P \\ (\textsf{rec}\, \mathcal{Y}.Q)\{\mathcal{X}/P\} & \triangleq & \textsf{rec}\, \mathcal{Y}.(Q\{\mathcal{X}/P\}) & (\textsf{rec}\, \mathcal{X}.Q)\{\mathcal{X}/P\} & \triangleq & \textsf{rec}\, \mathcal{X}.Q \\ (n \cdot l?(\tilde{x}).Q)\{\mathcal{X}/P\} & \triangleq & n \cdot l?(\tilde{x}).(Q\{\mathcal{X}/P\}) & ((\boldsymbol{\nu} a)Q)\{\mathcal{X}/P\} & \triangleq & (\boldsymbol{\nu} a)(Q\{\mathcal{X}/P\}) \\ (n \cdot l!(\tilde{m}).Q)\{\mathcal{X}/P\} & \triangleq & n \cdot l!(\tilde{m}).(Q\{\mathcal{X}/P\}) & & & \end{array}$$

In the next section we define the operational semantics of the π_{lab}-Calculus, so as to provide a formal definition of how π_{lab}-Calculus processes evolve through communications.

2.2 π_{lab}-Calculus Operational Semantics

The operational semantics of the π_{lab}-Calculus is defined by a labeled transition system. A transition $P \xrightarrow{\lambda} Q$ states that process P may evolve to process Q by performing the action represented by the transition label λ. We may describe a transition as an observation performed over the process by an external environment. In fact, P can either evolve autonomously to Q, by means of an internal action, or P can communicate with the external environment by a communication action specified in λ. Before presenting the definition of the transition relation, that captures this notion of process evolution, we first introduce some auxiliary notions. We define transition labels which specify such internal and external communication capabilities.

Definition 2.2.1 (Transition Labels)
Transition labels are defined as follows:

$$\lambda \;\; ::= \;\; \tau \mid c \cdot l!(\tilde{a}) \mid c \cdot l?(\tilde{a}) \mid (\nu a)\lambda \quad \text{(Transition Labels)}$$

We denote by $\mathcal{T}$ the set of all transition labels.

Internal actions are denoted by label τ which represents autonomous evolutions of processes, while communication actions with the environment are denoted by output label $c \cdot l!(\tilde{a})$ and input

label $c \cdot l?(\tilde{a})$, where $\tilde{a}$ specifies the names sent and the names received in the communication, respectively. In $(\nu a)\lambda$ the distinguished occurrence of a is bound with scope λ. We define operators that collect the free and bound name sets of a transition label, denoted by $fn(\lambda)$ and $bn(\lambda)$, respectively, and $na(\lambda)$ to denote both free and bound names of a transition label.

Definition 2.2.2 (Transition Free Names)
We denote by $fn(\lambda)$ the set of free names of transition label λ, defined inductively as follows:

$$fn(\tau) \triangleq \emptyset \qquad fn(c \cdot l!(\tilde{a})) \triangleq \{c\} \cup \tilde{a} \qquad fn(c \cdot l?(\tilde{a})) \triangleq \{c\} \cup \tilde{a} \qquad fn((\nu a)\lambda) \triangleq fn(\lambda) \setminus \{a\}$$

Definition 2.2.3 (Transition Bound Names)
We denote by $bn(\lambda)$ the set of bound names of transition label λ, defined inductively as follows:

$$bn(\tau) \triangleq \emptyset \qquad bn(c \cdot l!(\tilde{a})) \triangleq \emptyset \qquad bn(c \cdot l?(\tilde{a})) \triangleq \emptyset \qquad bn((\nu a)\lambda) \triangleq bn(\lambda) \cup \{a\}$$

Definition 2.2.4 (Transition Names)
We denote by $na(\lambda)$ the set of names of transition label λ, defined as $na(\lambda) \triangleq fn(\lambda) \cup bn(\lambda)$.

An output label that specifies such a bound name $(\nu a)\lambda$ (or, in general, a set of bound names $(\nu a_1)\ldots(\nu a_k)\lambda$, abbreviated by $(\nu\tilde{a})\lambda$) denotes a communication carrying a restricted name (set) which was previously local to a part of the system, but is at that moment being extruded to the external environment. To determine the names that are sent in a communication, so as to capture scope extrusion, we introduce an operator that collects the names that occur exclusively in the object of an output.

Definition 2.2.5 (Emitted Names)
We denote by $out(\lambda)$ the set of names object to an output label, defined inductively as follows:

$$out(\tau) \triangleq \emptyset \qquad out(c \cdot l!(\tilde{a})) \triangleq \tilde{a} \setminus \{c\} \qquad out(c \cdot l?(\tilde{a})) \triangleq \emptyset \qquad out((\nu a)\lambda) \triangleq out(\lambda) \setminus \{a\}$$

For a communication label λ we denote by $\overline{\lambda}$ the dual matching label obtained by swapping inputs with outputs, defined next.

Definition 2.2.6 (Transition Label Duality)
We denote by $\overline{\lambda}$ the dual of transition label λ, defined as follows:

$$\overline{c \cdot l!(\tilde{a})} \triangleq c \cdot l?(\tilde{a}) \qquad \overline{c \cdot l?(\tilde{a})} \triangleq c \cdot l!(\tilde{a})$$

Duality in transition labels captures the notion of actions that may be used by parallel processes to synchronize (in this case, passing names along the communication). Notice τ has no dual transition label, and thus it does not synchronize with any other action, since it represents an internal action to a process, e.g., an already succeeded synchronization. Also, we abstract away from transitions which specify bound names, hence duality is defined exclusively for basic communication actions.

We may now present the definition of the transition relation which formally characterizes how processes evolve, either by internal actions or by communications with the environment.

$$c \cdot l!(\tilde{a}).P \xrightarrow{c \cdot l!(\tilde{a})} P \;(Out) \qquad\qquad c \cdot l?(\tilde{x}).P \xrightarrow{c \cdot l?(\tilde{a})} P\{\tilde{x}/\tilde{a}\} \;(In)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a \in out(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{(\nu a)\lambda} Q}(Open) \qquad \frac{P \xrightarrow{\lambda} Q \quad a \notin na(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{\lambda} (\boldsymbol{\nu} a)Q}(Res)$$

$$\frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R)}{P \mid R \xrightarrow{\lambda} Q \mid R}(Par\text{-}l) \qquad \frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R)}{R \mid P \xrightarrow{\lambda} R \mid Q}(Par\text{-}r)$$

$$\frac{P \xrightarrow{(\nu \tilde{a})\overline{\lambda}} P' \quad Q \xrightarrow{\lambda} Q' \quad \tilde{a} \# fn(Q)}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} \tilde{a})(P' \mid Q')}(Close\text{-}l) \qquad \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{(\nu \tilde{a})\overline{\lambda}} Q' \quad \tilde{a} \# fn(P)}{P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu} \tilde{a})(P' \mid Q')}(Close\text{-}r)$$

$$\frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}(Comm) \qquad \frac{P\{\mathcal{X}/\mathbf{rec}\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathbf{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q}(Rec)$$

Figure 2.2: Transition Rules.

The relation as a whole is characterized by a set of inference rules which generically describe process evolutions, each rule specifying an observation that can be performed over a process given an observation that can be performed over a subprocess. For instance, consider rule:

$$\frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R)}{P \mid R \xrightarrow{\lambda} Q \mid R}$$

which specifies that if P exhibits transition λ to Q then the process consisting of the parallel composition of P and some process R also exhibits transition λ, since P and R are both active, arriving at a configuration which is the parallel composition of Q and R. Hence any behavior of a branch of a parallel composition is exhibited at the level of the parallel composition (provided the bound names being sent in λ, if any, are distinct of the free names of Q, i.e., $bn(\lambda) \# fn(R)$). Given this basic understanding, we now present the transition relations.

Definition 2.2.7 (Transition Relations)
The transition relations $\{\xrightarrow{\lambda} \mid \lambda \in \mathcal{T}\}$ *are defined by the rules of Figure 2.2.*

Transition rules presented in Figure 2.2 should be fairly clear to a reader familiar with mobile process calculi. In rule (Out) an output is observed over the output-prefixed process, representing a message emission to the external environment, being the arrival state of the transition the continuation of the output-prefixed process. The symmetric rule (In) specifies the input observation over the input-prefixed process, representing a message reception from the external environment. The arrival state of the transition is the continuation of the input prefixed-process where the variables $\tilde{x}$ are replaced by the received names $\tilde{a}$.

Rule ($Open$) corresponds to the bound output or extrusion rule, in which a bound name a is extruded to the environment in an output message λ. Rule (Res) states that when the restricted name is not mentioned in the transition label, then such transition transparently goes through the name restriction. Rules (Par-l) and (Par-r) state that the parallel composition exhibits the

transitions of either branch, avoiding unintended name collisions.

Rule (*Comm*) describes a synchronization taking place when the branches of the parallel composition exhibit dual transitions, thus originating an internal action τ. Similarly in rules (*Close-l*) and (*Close-r*) but where also a set of restricted names $\tilde{a}$ is transmitted, so the scope of the bound names is enlarged to contain both emitter (who knows the bound names) and receiver (who gains knowledge of the bound names), so the scope is *close*d at that level, provided the bound name identifiers are fresh to Q so as to avoid unintended name capture. In rule (*Rec*) a recursive process exhibits the transition of the unfolding of the recursion.

Sometimes it is useful to talk only about autonomous evolution of processes, captured by τ transitions. Namely when the focus is on closed systems, not subject to interaction with the environment, we are only interested on autonomous behavior. This notion is directly captured by the reduction relation which we define on top of the labelled transition system.

Definition 2.2.8 (Reduction)
The reduction relation between processes, noted $P \to Q$, is defined as $P \xrightarrow{\tau} Q$. Also, we denote by $\xrightarrow{*}$ the reflexive transitive closure of the reduction relation.

The $\xrightarrow{*}$ relation is useful for reasoning about process evolution in a wider sense as it captures sequences of zero or more steps of evolution, while each reduction corresponds to a single evolution. We may read $P \xrightarrow{*} Q$ as "P evolves in a number of steps to Q".

For the sake of illustration, in the next section we show how to put the transition inference rules to work by means of an example illustrating a formal proof of a process transition.

2.2.1 Proving a Transition

To provide some intuition we show how a transition can be proven using such a set of rules. Consider the following process:

$$(\boldsymbol{\nu} chat) c \cdot \texttt{talk!}(chat).P \mid R \mid c \cdot \texttt{talk?}(x).Q$$

which specifies the parallel composition of three processes, where the one on the left hand side is willing to send a private name *chat* on a `talk`-labeled output on c, and the one on the right hand side is waiting on such communication. Using rules (*Out*), (*Open*) and (*Par-l*) we derive:

$$\cfrac{\cfrac{c \cdot \texttt{talk!}(chat).P \xrightarrow{c\cdot\texttt{talk!}(chat)} P}{(\boldsymbol{\nu} chat) c \cdot \texttt{talk!}(chat).P \xrightarrow{(\nu chat) c\cdot\texttt{talk!}(chat)} P}}{(\boldsymbol{\nu} chat) c \cdot \texttt{talk!}(chat).P \mid R \xrightarrow{(\nu chat) c\cdot\texttt{talk!}(chat)} P \mid R}$$

Through rule (*In*) we have: $c \cdot \texttt{talk?}(x).Q \xrightarrow{c\cdot\texttt{talk?}(chat)} Q\{x/chat\}$. Then, using rule (*Close-l*) we may derive:

$$\frac{(\boldsymbol{\nu} chat) c \cdot \texttt{talk!}(chat).P \mid R \xrightarrow{(\nu chat) c\cdot\texttt{talk!}(chat)} P \mid R \qquad c \cdot \texttt{talk?}(x).Q \xrightarrow{c\cdot\texttt{talk?}(chat)} Q\{x/chat\}}{(\boldsymbol{\nu} chat) c \cdot \texttt{talk!}(chat).P \mid R \mid c \cdot \texttt{talk?}(x).Q \xrightarrow{\tau} (\boldsymbol{\nu} chat)(P \mid R \mid Q\{x/chat\})}$$

Notice that in the arrival configuration of the transition $(\boldsymbol{\nu}\, chat)(P \mid R \mid Q\{x/chat\})$ the scope of the restricted name *chat* is enlarged so as to contain the receiver process (collision with names in R is avoided in the meanwhile, through the side conditions which we omit from the description).

2.2.2 Programming the Purchase Scenario in the π_{lab}-Calculus

In this section we show how we can program the purchase scenario, presented in the Introduction (Section 1.2.1), using the π_{lab}-Calculus. We use standard name passing communication to represent the service instantiation primitives and conversation join. In fact, this is how we later define such service-oriented primitives in the Conversation Calculus. However in the π_{lab}-Calculus it is not clear how to present such idioms in a generic way, so we hard code them in the example. The service definition primitive is represented by a name input, as, for instance, the code:

$$Seller \cdot \texttt{BuyService?}(x).\, P$$

which represents a `BuyService` is available at channel `Seller`. To create a new service instance we use the output capability, so, for example, to instantiate the above service we use the code:

$$(\boldsymbol{\nu}\, buyChat)(\ Seller \cdot \texttt{BuyService!}(buyChat).\, Q\ )$$

where a fresh name is passed to the service provider side (*buyChat*) so as to allow for the service interaction to take place in a medium with a freshly created identity. So, P and Q will interact between them by using this freshly created name. The other way to use the service definition is to ask it to join an ongoing service interaction: the join idiom. To implement the join we also consider the output capability, but now, instead of passing a freshly created name, the output must carry the name of the medium where the current service interaction is taking place. So, for instance, considering the current service interaction is taking place at x, we program by:

$$(\ \cdots\ ).Shipper \cdot \texttt{DeliveryService!}(x).R$$

a call to service `DeliveryService` provided by *Shipper*, passing it name x, allowing it to join the current service interaction.

We thus implement the purchase scenario with the following code:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, buyChat) \\
(Seller \cdot \texttt{BuyService!}(buyChat). \\
\qquad buyChat \cdot \texttt{buy!}(prod).\ buyChat \cdot \texttt{price?}(p).\ buyChat \cdot \texttt{details?}(d)\ ) \\
\ \mid \\
PriceDB \mid \\
Seller \cdot \texttt{BuyService?}(x). \\
\qquad x \cdot \texttt{buy?}(prod).\ Seller \cdot \texttt{askPrice!}(prod). \\
\qquad Seller \cdot \texttt{priceVal?}(p).\ x \cdot \texttt{price!}(p). \\
\qquad Shipper \cdot \texttt{DeliveryService!}(x).\ x \cdot \texttt{product!}(prod) \\
\ \mid \\
Shipper \cdot \texttt{DeliveryService?}(y). \\
\qquad y \cdot \texttt{product?}(p).\ y \cdot \texttt{details!}(data)
\end{array}
$$

which represents the three participants in the purchase service collaboration: the first is willing to buy a product, wanting to know price and delivery details; the second offers a buy service, which interacts with a price database to determine the price, and delegates the shipping part to an external service; the third offers a delivery service that provides some details of the delivery, given the product information. The code shown above, in one step, evolves to:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat})\;\; (\\
\qquad \mathit{buyChat} \cdot \mathtt{buy}!(\mathit{prod}).\, \mathit{buyChat} \cdot \mathtt{price}?(p).\, \mathit{buyChat} \cdot \mathtt{details}?(d) \\
\;| \\
\mathit{PriceDB} \mid \\
\qquad \mathit{buyChat} \cdot \mathtt{buy}?(\mathit{prod}).\, \mathit{Seller} \cdot \mathtt{askPrice}!(\mathit{prod}). \\
\qquad \mathit{Seller} \cdot \mathtt{priceVal}?(p).\, \mathit{buyChat} \cdot \mathtt{price}!(p). \\
\qquad \mathit{Shipper} \cdot \mathtt{DeliveryService}!(\mathit{buyChat}).\, \mathit{buyChat} \cdot \mathtt{product}!(\mathit{prod})\;\;) \\
\;| \\
\mathit{Shipper} \cdot \mathtt{DeliveryService}?(y). \\
\qquad y \cdot \mathtt{product}?(p).\, y \cdot \mathtt{details}!(\mathit{data})
\end{array}
$$

where the name *buyChat* has been passed from the process instantiating the service to the process publishing the service. Notice the scope of the corresponding name restriction has grown so as to encompass the process that received the name. At that point, synchronizations on $\mathit{buyChat} \cdot \mathtt{buy}$ and $\mathit{buyChat} \cdot \mathtt{price}$ can take place between the two processes (interleaved by the interaction with the *PriceDB* process on $\mathit{Seller} \cdot \mathtt{askPrice}$ and on $\mathit{Seller} \cdot \mathtt{priceVal}$). At that point the process has evolved to the following configuration:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat})\;\; (\\
\qquad \mathit{buyChat} \cdot \mathtt{details}?(d) \\
\;| \\
\mathit{PriceDB}' \mid \\
\qquad \mathit{Shipper} \cdot \mathtt{DeliveryService}!(\mathit{buyChat}).\, \mathit{buyChat} \cdot \mathtt{product}!(\mathit{prod})\;\;) \\
\;| \\
\mathit{Shipper} \cdot \mathtt{DeliveryService}?(y). \\
\qquad y \cdot \mathtt{product}?(p).\, y \cdot \mathtt{details}!(\mathit{data})
\end{array}
$$

which then allows *Shipper* to join the ongoing interaction on *buyChat*, so the system evolves to:

$$
\begin{array}{l}
(\boldsymbol{\nu}\, \mathit{buyChat})\;\; (\\
\qquad \mathit{buyChat} \cdot \mathtt{details}?(d) \\
\;| \\
\mathit{PriceDB}' \mid \\
\qquad \mathit{buyChat} \cdot \mathtt{product}!(\mathit{prod}) \\
\;| \\
\qquad \mathit{buyChat} \cdot \mathtt{product}?(p).\, \mathit{buyChat} \cdot \mathtt{details}!(\mathit{data})\;\;)
\end{array}
$$

where, once again, the name *buyChat* has been passed (and its scope has grown accordingly), and now three distinct processes are willing to interact on *buyChat*. The remaining interactions

of the purchase service collaboration can then take place, first on $buyChat \cdot \texttt{product}$ and then on $buyChat \cdot \texttt{details}$ between the three processes.

2.3 Remarks

Introducing the π-Calculus in further detail is out of reach for this text, and therefore we refer the interested reader to the reference texts, books and tutorials by Milner, namely [76, 77, 78], and to the current standard reference for the π-Calculus by Sangiorgi et al. [85].

Although we did not find a presentation of a variant of the π-Calculus that exactly suited our purposes (however, we do not claim there is none), it is clear how the labeled π-Calculus we described can be encoded in, for instance, a π-Calculus equipped with pattern matching, e.g., the applied π-Calculus [1], or with polyadic synchronization [32]. Considering pattern matching, labels can be encoded by placing them at the first argument of both the output and the input: an output in the labeled π-Calculus $n \cdot l!(m_1, \ldots, m_k).P$ is encoded in $n!(l, m_1, \ldots, m_k)$ and an input $n \cdot l?(x_1, \ldots, x_k).P$ is encoded in $n?(l, x_1, \ldots, x_k)$, which will then behave in the same way as synchronization will occur only when output and input agree in the first argument.

In conclusion, the labeled π-Calculus is a slight extension of a core π-Calculus that preserves the fundamental and canonical aspects of the original model, which can be argued by verifying that the operational semantics is defined in a facsimile set of rules, being the only difference the extension of the transition labels. On the other hand, it is perhaps as far as our approach can go in the direction of a π-Calculus like canonical model, since we make important use of the combination of two identifiers — channel name and label index — as we will see next.

Chapter 3

Introducing Conversation Types

In this chapter we introduce the (core) conversation type language that supports the static verification of the conversation fidelity property: "do all participants in a collaboration follow well-defined protocols of interaction?" (originally introduced in [27, 28]). We start by presenting the main ideas behind our development, then we present the technical artifacts that instantiate these ideas. We proceed by presenting the typing rules that associate π_{lab}-Calculus processes to conversation types and show the safety results we prove for *well-typed* processes. We also show a typing derivation for a simple example (the purchase scenario), so as to demonstrate the sort of systems we are able to address using our type system, while already allowing us to distinguish our approach as such systems fall out of scope of previous works. Before ending the chapter we present some notes on related work and on some overall assessment of our development.

3.1 Analysis of Dynamic Conversations

As motivated in the Introduction, it is crucial to develop mechanisms that guarantee the safe interaction in service-oriented systems, namely to have a means to guarantee protocol safety in a multiparty collaboration: all parties interacting in a service collaboration follow well-defined protocols of interaction. A global protocol description (a choreography, in web-service terminology) structures a set of related interactions — a multiparty conversation — that is to be carried out by a number of parties. Consider, for instance, the following global protocol description for the purchase scenario illustrated in the message sequence chart shown in Figure 1.2:

1. $\texttt{buy}(Tp) : Buyer \rightarrow Seller$
2. $\texttt{price}(Tm) : Seller \rightarrow Buyer$
3. $\texttt{product}(Tp) : Seller \rightarrow Shipper$
4. $\texttt{details}(Td) : Shipper \rightarrow Buyer$

where we specify, in order, the several interactions of the service conversation by identifying the message and the sender and the receiver of the message. Similar kind of specifications are considered in some approaches that verify protocol safety (e.g., the global types of [9, 57]), which use them to determine the roles of the individual participants (cf., local types) by filtering out the interactions relative to some party from the global specification, so as to check the individual participants implement their local roles. Such approaches lack in flexibility as they only allow to describe systems with fixed configurations, where the individual communication roles involved

in the service collaboration are a priori set, leaving systems involving runtime delegation of tasks to external services out of their reach.

To be able to address such challenging scenarios we introduce conversation types, a flexible type structure that allows for loosely coupled specifications of service collaborations. Conversation types combine, at the same level in the type language, global and local protocol specifications, which allows us to characterize, in a compositional way, how protocols are implemented by the several parts of the system. Consider for instance the conversation type which captures the interaction scheme described in the global protocol specified above:

$$\tau\,\texttt{buy}(Tp).\tau\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).\tau\,\texttt{details}(Td)$$

which describes the overall service protocol consisting in a sequence of internal (τ) message exchanges, where there is no explicit indication of the particular identities for the communicating parties (cf., loose coupling): as a result, we do not enforce who carries out the communication capability in particular, instead we are interested in guaranteeing that *someone* exercises the communication capability. In our theory, such overall specifications can be "projected" into global/local mixed specifications, allowing for the characterization not only of the overall protocol and that of the individual participants but also of *parts* of the system, corresponding to the combined roles of some (not necessarily all) of the participants in the interaction. For example, we may obtain the projection of the above protocol in the individual type of the buyer:

$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td)$$

which specifies the output (`!`) of message `buy`, followed by the input (`?`) of messages `price` and `details`, but also we may obtain the projection of the global protocol in the rest of the system, i.e., the roles of seller and shipper combined:

$$?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$

Notice that input and output (local) specifications are mixed with internal (choreographic) descriptions in the same type term. Such message exchange specifications form the basis of our behavioral types which we use to structure a set of related interactions: a conversation. Behavioral types incorporate the basic constructs to specify such structured concurrent behavior: message prefix $M.B$ and parallel composition $B_1 \mid B_2$.

Since processes may interact, both simultaneously and throughout their execution, in several of such conversations, in order to characterize the interaction potential of a process we must collect all the interactions the process specifies in all such conversations. The typing judgement:

$$P :: n : [B_n] \mid m : [B_m] \mid o : [B_o] \mid \ldots$$

that associates process P to type $n : [B_n] \mid m : [B_m] \mid o : [B_o] \mid \ldots$, says that process P interacts in conversation n according to behavioral type B_n, in conversation m according to behavioral type B_m, and so forth, where each behavioral type (B) characterizes, in a structured way, the set of interactions the process has in that specific conversation. The behavioral types are thus an abstraction of the behavior of the process relative to each individual conversation.

To obtain such typed characterizations of process behavior we are technically faced with

$$
\begin{array}{lll}
n, m, o, \ldots & \in \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
l, s \ldots & \in \mathcal{L} & \text{(Labels)} \\
\mathcal{X}, \mathcal{Y}, \ldots & \in \chi & \text{(Recursion Variables)}
\end{array}
$$

$$
\begin{array}{llll}
B & ::= & M.B \mid B_1 \mid B_2 \mid \mathbf{0} \mid \mathtt{rec}\,\mathcal{X}.B \mid \mathcal{X} & \text{(Behavioral)} \\
M & ::= & p\,l(C) & \text{(Message)} \\
p & ::= & \mathtt{!} \mid \mathtt{?} \mid \tau & \text{(Polarity)} \\
C & ::= & [B] & \text{(Conversation)} \\
L & ::= & n : C \mid L_1 \mid L_2 \mid \mathbf{0} & \text{(Located)}
\end{array}
$$

Figure 3.1: Basic Conversation Types Syntax.

questions such as: "given the behaviors of two processes running in parallel, how do we combine them so as to obtain the behavior of the parallel composition?" and "how do we account for behaviors that are delegated through message exchanges carrying conversation identifiers?". In the remaining of this chapter we introduce the type language and a set of operations over types that allow us to capture the combination of behaviors, to know when can such behaviors be safely combined, and also how we can characterize a relation between types so as to allow for more general descriptions in the typing characterizations: the subtyping relation.

We may then characterize, using conversation types, systems specified in the π_{lab}-Calculus, so as to guarantee they follow well-defined protocols of interaction. The intended type safety property will be formally stated in Corollary 3.3.10: it implies conversations agree to declared protocols, and the absence of certain kind of runtime errors.

3.2 Type Language

In this section we formally present the conversation types language. The two main syntactical categories of the type language are behavioral types B and located types L. Behavioral types capture the behavior of structured conversations, while located types associate the conversation names to the respective behaviors.

Definition 3.2.1 (Core Conversation Type Language)
The syntax of the core conversation type language is given in Figure 3.1.

Behavioral types feature parallel composition $B_1 \mid B_2$ to describe independent concurrent behavior, $\mathbf{0}$ to describe inactive behavior, and recursion. The prefix $M.B$ specifies a process that sends ($\mathtt{!}$), receives ($\mathtt{?}$), or internally (τ) exchanges a message M before proceeding with behavior B. Message types M are thus defined by a polarity p (either output $\mathtt{!}$, input $\mathtt{?}$ or internal action τ), a label l, and the type C of what is communicated in the message. Notice that a message M may refer to an *internal* exchange between two partners, if it is of the form $\tau\, l(C)$.

Conversation types C are a delimited behavior $[B]$, where B specifies the message interactions that may take place in the conversation. Located types L collect (using composition) type associations between conversation names and their types. In a typing judgment, a located type will then specify the conversation type of each visible conversation.

We introduce some convenient syntactic abbreviations and conventions for behavioral types.

Syntactic conventions:

$M.B_1 \mid B_2$ is to be read as $(M.B_1) \mid B_2$.

$\mathtt{rec}\,\mathcal{X}.B_1 \mid B_2$ is to be read as $(\mathtt{rec}\,\mathcal{X}.B_1) \mid B_2$.

Abbreviations:

M stands for $M.\mathbf{0}$, where appropriate.

$\star M$ stands for $\mathtt{rec}\,\mathcal{X}.M.\mathcal{X}$.

We next present the technical artifacts used in the type system for π_{lab}-Calculus processes, while introducing some auxiliary notions along the way. We start by presenting a means by which we distinguish the two common communication patterns in service-oriented computing: interactions relative to the instantiation of services and interactions relative to an ongoing service collaboration. Then we characterize when are two behaviors independent by means of an apartness predicate, a notion essential to capture the safe composition of systems. Using such notion of independent behavior we proceed to the characterization of the relation between types (the subtyping relation), which serves as a mechanism that allows for more flexible typing characterizations. Finally, we present a key operation in our type system that captures the behavioral composition of types, and therefore explains the behavioral composition of processes.

3.2.1 Linear and Shared Types

For typing purposes, we split the set of labels $\mathcal{L}$ into shared $\mathcal{L}_\star$ and plain $\mathcal{L}_p$ labels.

Definition 3.2.2 (Plain and Shared Labels)
We denote by $\mathcal{L}_\star$ and $\mathcal{L}_p$ subsets of $\mathcal{L}$ such that $\mathcal{L}_\star \cap \mathcal{L}_p = \emptyset$ and $\mathcal{L}_\star \cup \mathcal{L}_p = \mathcal{L}$.

Messages which are to be used linearly are defined with plain labels, and messages which are to be used exponentially are defined with shared labels. In such way, we distinguish two common interaction patterns in service-oriented computing: a service is expected to be available *exponentially* in the sense that there can be multiple clients trying to use the same service simultaneously. Also, there may be multiple peers providing the same service. On the other hand during an ongoing service interaction messages that flow between collaborating partners are expected to be used *linearly*, in the sense there should be a unique pair of parties that can communicate on a specific action at a given moment — *race absence*. Such linear interactions are to be protected from external interference and support the definition of deterministic protocols.

To characterize the external interface of (persistent) services it is useful to introduce a notion of *exponential output* interface, which specifies the calls to external services. Such notion will then characterize precisely the external interface of a persistent service that consist solely in the list of external services it requires to function. We thus define exponential output types.

Definition 3.2.3 (Exponential Output Type)
Exponential output message types, noted $\star M^!$, exponential output behavioral types, noted $\star B^!$,

and exponential output located types, noted $\star L^!$*, are defined as follows:*

$$
\begin{array}{llll}
\star M^! & ::= & \star M \qquad (M = !\, l(C) \text{ and } l \in \mathcal{L}_\star) & \text{(Exponential Output Message Type)} \\
\star B^! & ::= & \star M^! \mid \star B_1^! \mid \star B_2^! \mid \mathbf{0} & \text{(Exponential Output Behavioral Type)} \\
\star L^! & ::= & n : [\star B^!] \mid \star L_1^! \mid \star L_2^! \mid \mathbf{0} & \text{(Exponential Output Located Type)}
\end{array}
$$

Exponential output types are thus collections of persistent message outputs. Since several copies of persistently available services may be concurrently active, their interfaces must be defined exclusively on shared messages (from $\mathcal{L}_\star$). To determine, in general, when two types characterize systems that may safely run concurrently, we introduce the notion of *apartness*, presented next.

3.2.2 Apartness

A key notion in our development is type apartness: intuitively, two types are apart when they type subsystems that may be composed without undesirable interferences. To define the notion of interference needed here we distinguish between the cases of linear and exponential types: types that share plain (linear) messages are not apart, while types that share exponential (shared) labels are apart, provided such shared messages are defined with the same argument types.

To define apartness we first introduce the set of message types $Msg_{\mathcal{L}}(B)$ and the set of message labels $Lab_{\mathcal{L}}(B)$ of a behavioral type B. Both sets are indexed by a label set $\mathcal{L}$, which can be refined to either $\mathcal{L}_\star$ and $\mathcal{L}_p$ so as to allow, e.g., the distinction of the message sets with labels from shared labels (from $\mathcal{L}_\star$) and those from plain labels (from $\mathcal{L}_p$).

Definition 3.2.4 (Message Set)
We denote by $Msg_{\mathcal{L}}(B)$ *the set of message types defined with labels from* $\mathcal{L}$ *of a behavioral type* B*, defined as follows:*

$$
\begin{array}{lll}
Msg_{\mathcal{L}}(\mathbf{0}) & \triangleq & \emptyset \\
Msg_{\mathcal{L}}(B_1 \mid B_2) & \triangleq & Msg_{\mathcal{L}}(B_1) \cup Msg_{\mathcal{L}}(B_2) \\
Msg_{\mathcal{L}}(\mathcal{X}) & \triangleq & \emptyset \\
Msg_{\mathcal{L}}(\mathtt{rec}\, \mathcal{X}.B) & \triangleq & Msg_{\mathcal{L}}(B) \\
Msg_{\mathcal{L}}(p\, l(C).B) & \triangleq & \{(p\, l(C)) \mid l \in \mathcal{L}\} \cup Msg_{\mathcal{L}}(B)
\end{array}
$$

For example, given some behavioral type B, $Msg_{\mathcal{L}_p}(B)$ is the set of all plain (in $\mathcal{L}_p$) message types $(p\, l(C))$ occurring in B, leaving out message types defined on shared labels (those belonging to $\mathcal{L}_\star$). Using the message set, we may define the set of labels of a behavioral type, noted $Lab_{\mathcal{L}}(T)$.

Definition 3.2.5 (Label Set)
We denote by $Lab_{\mathcal{L}}(B)$ *the set of labels from* $\mathcal{L}$ *of a behavioral type* B*, defined as follows:*

$$
Lab_{\mathcal{L}}(T) \triangleq \{l \mid (p\, l(C)) \in Msg_{\mathcal{L}}(B)\}
$$

The set of labels gives us enough information to determine the apartness of types with respect to messages defined on plain labels. To determine apartness with respect to messages defined on shared labels we introduce conformance.

Definition 3.2.6 (Conformance)
We say two behavioral types B_1, B_2 are conformant, noted $B_1 \asymp B_2$, if for any two message types $\mathsf{p}_1\, l(C_1)$ and $\mathsf{p}_2\, l(C_2)$ such that:

$$(\mathsf{p}_1\, l(C_1)) \in Msg_{\mathcal{L}_\star}(B_1) \qquad \textit{and} \qquad (\mathsf{p}_2\, l(C_2)) \in Msg_{\mathcal{L}_\star}(B_2)$$

then $C_1 = C_2$ and if $\mathsf{p}_i = \mathtt{?}$ then $\mathsf{p}_j \neq \mathtt{!}$ for $\{i,j\} = \{1,2\}$.

Two behavioral types are compatible on shared messages if they specify messages defined on shared labels with identical argument types and also if they are defined on determined polarities. For instance two message types defined on shared labels and polarity ! are conformant as they represent compatible calls to the same service. We only exclude the case when messages are of dual polarities (! and ?), which is used to force such messages to synchronize and is explained by the behavioral merge (Definition 3.2.13). Using the label set and conformance, we may now define type apartness between behavioral types.

Definition 3.2.7 (Apartness)
We say two behavioral types B_1, B_2 are apart, noted $B_1 \# B_2$, if their plain label sets are disjoint ($Lab_{\mathcal{L}_p}(B_1) \# Lab_{\mathcal{L}_p}(B_2)$) and they are conformant ($B_1 \asymp B_2$).

Essentially, apartness ensures disjointness of plain ("linear") types, and consistency of shared ("exponential") types (cf. [64]). Given this notion of type independence we may now present the subtyping relation.

3.2.3 Subtyping

Types are related by the subtyping relation we now present. Intuitively, we say type L_1 is a subtype of type L_2, noted $L_1 <: L_2$, when a process of type L_1 can safely be used in a context where a process of type L_2 is expected. Subtyping provides a way to generalize the typing characterization of processes, by its use in a subsumption rule of the form:

$$\frac{P :: L_1 \quad L_1 <: L_2}{P :: L_2}$$

which then allows to characterize a process by a more general type (the supertype) given the characterization of the process by a more specific type (the subtype), in such way introducing more flexibility in the typing characterization.

We start by presenting the subtyping relation for behavioral types, then we proceed to the subtyping relation for located types which is essentially the congruence closure of the first. Our subtyping rules express expected structural relationships of types, namely:

$$B_1 \mid B_2 <: B_2 \mid B_1$$

which specifies a commutativity principle for parallel composition of behavioral types. In such case, since we have $B_1 \mid B_2 <: B_2 \mid B_1$ and $B_2 \mid B_1 <: B_1 \mid B_2$ we abbreviate the symmetry by saying the types are equivalent, noted by the $\equiv$ symbol, hence:

$$B_1 \mid B_2 \equiv B_2 \mid B_1$$

The subtyping relation also specifies associativity of parallel composition of behavioral types:

$$B_1 \mid (B_2 \mid B_3) \equiv (B_1 \mid B_2) \mid B_3$$

and that $\mathbf{0}$ is a neutral element for parallel composition:

$$B \mid \mathbf{0} \equiv B$$

Thus $(- \mid -, \mathbf{0})$ is a commutative monoid with respect to subtyping. We adopt an equi-recursive approach to recursive types [83], based on simple unfoldings of recursive type terms:

$$\mathtt{rec}\,\mathcal{X}.B \equiv B\{\mathcal{X}/\mathtt{rec}\,\mathcal{X}.B\}$$

We embed the subtyping relation with reflexivity — $B <: B$ — and with transitivity:

$$\frac{B_1 <: B_3 \quad B_3 <: B_2}{B_1 <: B_2}$$

The following rules specify congruence principles so as to allow subtyping underneath parallel composition, recursive definitions and message prefix operators:

$$\frac{B_1 <: B_2}{B_3 \mid B_1 <: B_3 \mid B_2} \qquad \frac{B_1 <: B_2}{\mathtt{rec}\,\mathcal{X}.B_1 <: \mathtt{rec}\,\mathcal{X}.B_2} \qquad \frac{B_1 <: B_2}{M.B_1 <: M.B_2}$$

A key subtyping rule that allows us to introduce flexibility at the level of protocol specification is the following:

$$M.(B_1 \mid B_2) <: M.B_1 \mid B_2 \;\; (M \;\#\; B_2, \mathit{fv}(B_2) = \emptyset)$$

which allows for sequential protocols to export a more general concurrent interface, provided the behaviors specified in parallel are apart $M \;\#\; B_2$. The intuition is that if a process performs action M and after which exhibits behavior B_2, then it can safely be used in a context that expects a process that exhibits simultaneously action M and behavior B_2.

The following rule expresses a contraction principle for exponential output message types:

$$\star M^! \mid \star M^! <: \star M^!$$

which describes that a process that sends twice (each infinitely often) a message ($\star M^!$) can be safely used in a context where a process that sends such message once (infinitely often) is expected. Also regarding exponential output types we introduce a weakening principle by rule:

$$M <: \star M^! \;\; (M = \mathtt{!}\, l(C), l \in \mathcal{L}_\star)$$

which specifies that a process which outputs once a message defined on a shared label can be used in a context where a process that outputs the message an unbounded number of times is expected. Since processes waiting to receive such messages will be persistent, the composition of either a request or a persistent request with a persistent offer will capture the fact that the request(s) are satisfied. Given this basic understanding on the meaning of the rules, we may now define subtyping for behavioral types.

$$B_1 \mid B_2 \equiv B_2 \mid B_1 \;\; (\mathit{SubParComm}) \qquad B_1 \mid (B_2 \mid B_3) \equiv (B_1 \mid B_2) \mid B_3 \;\; (\mathit{SubParAssoc})$$

$$B \mid \mathbf{0} \equiv B \;\; (\mathit{SubParZero}) \qquad \texttt{rec}\,\mathcal{X}.B \equiv B\{\mathcal{X}/\texttt{rec}\,\mathcal{X}.B\} \;\; (\mathit{SubRecUnfold})$$

$$B <: B \;\; (\mathit{SubReflex}) \qquad \frac{B_1 <: B_3 \quad B_3 <: B_2}{B_1 <: B_2}(\mathit{SubTrans})$$

$$\frac{B_1 <: B_2}{B_3 \mid B_1 <: B_3 \mid B_2}(\mathit{SubPar}) \qquad \frac{B_1 <: B_2}{\texttt{rec}\,\mathcal{X}.B_1 <: \texttt{rec}\,\mathcal{X}.B_2}(\mathit{SubRec})$$

$$\frac{B_1 <: B_2}{M.B_1 <: M.B_2}(\mathit{SubPref}) \qquad \star M^! \mid \star M^! <: \star M^! \;\; (\mathit{SubExpFold})$$

$$M.(B_1 \mid B_2) <: M.B_1 \mid B_2 \;\; (M \;\#\; B_2, \mathit{fv}(B_2) = \emptyset) \;\; (\mathit{SubParPref})$$

$$M <: \star M^! \;\; (M = \,!\, l(C), l \in \mathcal{L}_\star) \;\; (\mathit{SubExpWeak})$$

Figure 3.2: Behavioral Types Subtyping Rules.

Definition 3.2.8 (Behavioral Types Subtyping and Equivalence)
We say behavioral type B_1 is a subtype of behavioral type B_2, noted $B_1 <: B_2$, if $B_1 <: B_2$ is derivable by the rules shown in Figure 3.2. Also, we say B_1 and B_2 are equivalent, noted $B_1 \equiv B_2$, if $B_1 <: B_2$ and $B_2 <: B_1$.

The subtyping relation between located types essentially lifts the behavioral types relation to a setting where such behavioral types are located. Subtyping for located types also specifies the commutative monoid rules for $(- \mid -, \mathbf{0})$ and congruence principles. We also have a split rule:

$$n : [B_1 \mid B_2] \equiv n : [B_1] \;\mid n : [B_2]$$

which captures the notion that the behavior in a single conversation can be described through distinct pieces. Analogously to behavioral types subtyping, we use $L_1 \equiv L_2$ to denote $L_1 <: L_2$ and $L_2 <: L_1$. We define the subtyping relation for located types.

Definition 3.2.9 (Located Types Subtyping and Equivalence)
We say located type L_1 is a subtype of located type L_2, noted $L_1 <: L_2$, if $L_1 <: L_2$ is derivable by the rules shown in Figure 3.3. Also, we say L_1 and L_2 are equivalent, noted $L_1 \equiv L_2$, if $L_1 <: L_2$ and $L_2 <: L_1$.

In the next section we present an operation used to behaviorally compose types, that thus explains the behavioral composition of processes.

3.2.4 Merge Relation

We introduce a key operation in which our typing rules rely: the ternary merge relation, noted by $B = B_1 \bowtie B_2$. The merge relation is used to define the composition of two types, so that if $B = B_1 \bowtie B_2$ then B is a particular (in general not unique) behavioral combination of the types B_1 and B_2. Merge is defined not only in terms of spatial separation, but also, and crucially, in terms of merging behavioral "traces". Notice also that it is not always the case that there

$$L_1 \mid L_2 \equiv L_2 \mid L_1 \;\; (SubLocParComm) \qquad L_1 \mid (L_2 \mid L_3) \equiv (L_1 \mid L_2) \mid L_3 \;\; (SubLocParAssoc)$$

$$L \mid \mathbf{0} \equiv L \;\; (SubLocParZero) \qquad n : [B_1 \mid B_2] \equiv n : [B_1] \mid n : [B_2] \;\; (SubLocSplit)$$

$$L <: L \;\; (SubLocReflex) \qquad \frac{L_1 <: L_3 \quad L_3 <: L_2}{L_1 <: L_2}(SubLocTrans)$$

$$\frac{L_1 <: L_2}{L_3 \mid L_1 <: L_3 \mid L_2}(SubLocPar) \qquad \frac{B_1 <: B_2}{n : [B_1] <: n : [B_2]}(SubLoc)$$

Figure 3.3: Located Types Subtyping Rules.

is B such that $B = B_1 \bowtie B_2$. On the other hand, if some such B exists, we use $B_1 \bowtie B_2$ to non-deterministically denote any such B (e.g., in conclusions of type rules).

Notation 3.2.10 *We use $B_1 \bowtie B_2$ to represent B such that $B = B_1 \bowtie B_2$.*

Intuitively, $B = B_1 \bowtie B_2$ holds if B_1 and B_2 may safely synchronize or interleave so as to produce behavioral type B. Before presenting the definition of the merge relation we introduce some auxiliary operations: the initial label set of a behavioral type B, noted $\mathcal{I}(B)$, which collects the set of labels of the actions immediately active in B, and message type substitution, noted $B\{M_1/M_2\}$, which replaces all occurrences of message M_1 by M_2 in B.

Definition 3.2.11 (Initial Label Set)
The initial label set of a behavioral type B, noted $\mathcal{I}(B)$, is defined as follows:

$$\begin{array}{lll} \mathcal{I}(\mathbf{0}) & \triangleq & \emptyset \\ \mathcal{I}(\mathcal{X}) & \triangleq & \emptyset \\ \mathcal{I}(p\,l(C).B) & \triangleq & \{l\} \end{array} \qquad \begin{array}{lll} \mathcal{I}(B_1 \mid B_2) & \triangleq & \mathcal{I}(B_1) \cup \mathcal{I}(B_2) \\ \mathcal{I}(\mathtt{rec}\,\mathcal{X}.B) & \triangleq & \mathcal{I}(B) \end{array}$$

Definition 3.2.12 (Message Type Substitution)
We denote by $B\{M_1/M_2\}$ the type obtained by replacing all occurrences of message type M_1 with message type M_2 in type B, defined inductively in the structure of types as follows:

$$\begin{array}{llll} \mathbf{0}\{M_1/M_2\} & \triangleq & \mathbf{0} & \\ (B_1 \mid B_2)\{M_1/M_2\} & \triangleq & (B_1\{M_1/M_2\}) \mid (B_2\{M_1/M_2\}) & \\ \mathcal{X}\{M_1/M_2\} & \triangleq & \mathcal{X} & \\ (\mathtt{rec}\,\mathcal{X}.B)\{M_1/M_2\} & \triangleq & \mathtt{rec}\,\mathcal{X}.(B\{M_1/M_2\}) & \\ (M_1.B)\{M_1/M_2\} & \triangleq & M_2.(B\{M_1/M_2\}) & \\ (M.B)\{M_1/M_2\} & \triangleq & M.(B\{M_1/M_2\}) & (\textit{if } M \neq M_1) \end{array}$$

Given these basic operations we may now present the merge relation. We start by describing informally the key rules, then we present the definition. Rule:

$$\frac{B_1 \# B_2}{B_1 \mid B_2 = B_1 \bowtie B_2}(Apart)$$

captures the composition of two independent behaviors B_1 and B_2, by specifying them in parallel in the resulting merge. The behaviors are independent since they are apart #. The merge of behaviors which are not independent must synchronize the actions that are not apart. So, to compose two message types, it must be the case that either they are apart or they can be synchronized, otherwise the composition is not possible. There are two rules that explain such synchronizations, one for messages defined on plain labels, and the other for messages defined on shared labels. For plain messages synchronization we have the following rule:

$$\frac{B = B_1 \bowtie B_2 \qquad l \in \mathcal{L}_p}{\tau\, l(C).B = ?\, l(C).B_1 \bowtie !\, l(C).B_2}(\mathit{Plain})$$

that merges dual output and input actions in an internal action τ. The continuation of the internal action is given by the merge of the continuations of the output and input. Rule (*Plain*) thus allows for $\tau\, l$ plain message types to be separated into send ! and receive ? capabilities which can then be distributed to different subprocesses.

Shared message synchronization is captured by rule:

$$\frac{B \asymp !\, l(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l(C)/\tau\ l(C)\} \mid \star\, ?\, l(C) = B \bowtie \star ?\, l(C)}(\mathit{Shared})$$

which synchronizes a persistently available input message type with all corresponding output message types. The resulting merge is then the type obtained replacing all $!\, l$ message types with $\tau\, l$ in B, in parallel with the persistent input message type: this allows for shared labeled inputs to synchronize and leave open the possibility for further synchronizations, expecting further outputs from the external environment. A τ shared message type thus represents that there is (at least one) persistently available input that matches the output, while a τ plain message type characterizes the uniquely determined synchronization on that plain label.

The following rule allows for the merge to interleave a message prefix:

$$\frac{M \# B_2 \quad B' \mid B'' \equiv B_1 \bowtie B_2 \quad M \# B'' \quad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \quad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2)}{M.B' \mid B'' = M.B_1 \bowtie B_2}(\mathit{Shuffle})$$

Rule (*Shuffle*) explains the composition of behaviors $M.B_1$ and B_2 by first composing B_1 and B_2 (since M is apart from B_2 it does not interfere with B_2) and second by placing the message prefix $M.B'$ so as to maintain (some of) the sequentiality information originally specified in $M.B_1$. On the one hand, no extra sequentiality may be imposed by prefixing B' with M in the resulting merge, with respect to the one originally specified in $M.B_1$, so as to guarantee that processes do not implement sequential protocols in parallel. This is guaranteed by condition $\mathcal{I}(B') \subseteq \mathcal{I}(B_1)$, which says that the labels of the immediately active messages of B' are a subset of the labels of the immediately active messages of B_1. On the other hand, no actions that were originally only in B_1 will be specified in parallel in B'', as guaranteed by $\mathcal{I}(B'') \subseteq \mathcal{I}(B_2)$. In such way, we allow for type synchronizations to occur in the continuation of message prefixes, without imposing any extra sequentiality information.

The following rule:

$$\frac{B_1 \not\# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_3}{B = B_1 \bowtie B_2 \mid B_3}(\mathit{ShufflePar})$$

explains the merge of the parallel composition $B_2 \mid B_3$ with type B_1 by first merging B_1 with B_2 then merging the resulting type with B_3, provided B_2 is not apart ($\#$) from B_1. This allows for several messages that are originally specified in parallel to merge with the same thread. We may now define the behavioral types merge relation.

Definition 3.2.13 (Behavioral Types Merge Relation)
We say type behavioral type B is the merge of behavioral types B_1 and B_2, noted $B = B_1 \bowtie B_2$, if $B = B_1 \bowtie B_2$ is derivable by the rules shown in Figure 3.4.

We state some properties of the behavioral types merge relation.

Proposition 3.2.14 (Behavioral Type Merge Relation Properties)
The behavioral types merge relation is commutative and associative:

(1). If $B = B_1 \bowtie B_2$ then $B = B_2 \bowtie B_1$.

(2). If $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ then there is B'' such that $B'' = B_2 \bowtie B_3$ and $B = B_1 \bowtie B''$.

Proof. (*1*) follows immediately from the definition, while (*2*) follows by induction on the derivation of $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ (See Appendix A.3). ∎

We now illustrate by means of a couple of examples how the merge relation is used so as to behaviorally combine types, as well as an example derivation that illustrates Proposition 3.2.14(*2*).

Example 3.2.15 *Consider type:*

$$?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm).?\,\mathtt{accept}().\tau\,\mathtt{product}(Tp).!\,\mathtt{details}(Td)$$

which specifies a process that inputs message `buy`*, then outputs message* `price`*, then inputs message* `accept`*, then internally exchanges message* `product`*, and finally outputs message* `details`*. When merged with the type:*

$$?\,\mathtt{price}(Tm).!\,\mathtt{accept}()$$

specifying the dual polarities for `price` *and* `accept`*, it yields type:*

$$?\,\mathtt{buy}(Tp).\tau\,\mathtt{price}(Tm).\tau\,\mathtt{accept}().\tau\,\mathtt{product}(Tp).!\,\mathtt{details}(Td)$$

which specifies the composite behavior that inputs message `buy`*, then has internal interactions on messages* `price`*,* `accept` *and* `product`*, and finally outputs message* `details`*.*

Example 3.2.16 *Consider type:*

$$!\,\mathtt{assessRate}(Td) \mid ?\,\mathtt{rateVal}(Tr)$$

that specifies two message types in parallel: an output on message `assessRate` *and an input on message* `rateVal`*. When composed with the type:*

$$?\,\mathtt{assessRate}(Td).!\,\mathtt{rateVal}(Tr)$$

$$\frac{B \asymp\ !\, l(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l(C)/\tau\ l(C)\} \mid \star\, ?\, l(C) = B \bowtie \star ?\, l(C)}\,(\textit{Shared-r})$$

$$\frac{B \asymp\ !\, l(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l(C)/\tau\ l(C)\} \mid \star\, ?\, l(C) = \star ?\, l(C) \bowtie B}\,(\textit{Shared-l})$$

$$\frac{B = B_1 \bowtie B_2 \qquad l \in \mathcal{L}_p}{\tau\, l(C).B =\ !\, l(C).B_1 \bowtie\ ?\, l(C).B_2}\,(\textit{Plain-r})$$

$$\frac{B = B_1 \bowtie B_2 \qquad l \in \mathcal{L}_p}{\tau\, l(C).B =\ ?\, l(C).B_1 \bowtie\ !\, l(C).B_2}\,(\textit{Plain-l})$$

$$\frac{M \# B_1 \quad B' \mid B'' \equiv B_1 \bowtie B_2 \quad M \# B' \quad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2) \quad \mathcal{I}(B') \subseteq \mathcal{I}(B_1)}{B' \mid M.B'' = B_1 \bowtie M.B_2}\,(\textit{Shuffle-r})$$

$$\frac{M \# B_2 \quad B' \mid B'' \equiv B_1 \bowtie B_2 \quad M \# B'' \quad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \quad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2)}{M.B' \mid B'' = M.B_1 \bowtie B_2}\,(\textit{Shuffle-l})$$

$$\frac{B_1 \not\# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_3}{B = B_1 \bowtie B_2 \mid B_3}\,(\textit{ShufflePar-r})$$

$$\frac{B_1 \not\# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_3}{B = B_1 \bowtie B_3 \mid B_2}\,(\textit{ShufflePar-rr})$$

$$\frac{B_2 \not\# B_3 \qquad B' = B_2 \bowtie B_3 \qquad B = B_1 \bowtie B'}{B = B_1 \mid B_2 \bowtie B_3}\,(\textit{ShufflePar-l})$$

$$\frac{B_2 \not\# B_3 \qquad B' = B_2 \bowtie B_3 \qquad B = B_1 \bowtie B'}{B = B_2 \mid B_1 \bowtie B_3}\,(\textit{ShufflePar-ll})$$

$$\frac{B = B_1 \bowtie B_2}{\texttt{rec}\, \mathcal{X}.B = \texttt{rec}\, \mathcal{X}.B_1 \bowtie \texttt{rec}\, \mathcal{X}.B_2}\,(\textit{Rec}) \qquad \frac{}{\mathcal{X} = \mathcal{X} \bowtie \mathcal{X}}\,(\textit{Var}) \qquad \frac{B_1 \# B_2}{B_1 \mid B_2 = B_1 \bowtie B_2}\,(\textit{Apart})$$

$$\frac{B' = B_1 \bowtie B_2 \qquad B'' = B_3 \bowtie B_4 \qquad B' \# B''}{B' \mid B'' = (B_1 \mid B_3) \bowtie (B_2 \mid B_4)}\,(\textit{Par-1})$$

$$\frac{B' = B_1 \bowtie B_4 \qquad B'' = B_3 \bowtie B_2 \qquad B' \# B''}{B' \mid B'' = (B_1 \mid B_3) \bowtie (B_2 \mid B_4)}\,(\textit{Par-2})$$

Figure 3.4: Behavioral Type Merge Relation Rules.

which specifies the dual polarities of the same messages, but where the messages are now specified sequentially, instead of in parallel, we obtain the type:

$$\tau\,\texttt{assessRate}(Td).\tau\,\texttt{rateVal}(Tr)$$

which specifies the internal message exchanges in a sequential way, where the sequentiality is due to the particular implementation of the protocol by one of the parties.

Example 3.2.17 *Consider type:*

$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td)$$

which characterizes a process that outputs message `buy`*, then inputs messages* `price` *and* `details`*. When merged with type:*

$$?\,\texttt{product}(Tp).!\,\texttt{details}(Td)$$

which characterizes a process that inputs message `product` *and then outputs message* `details`*, we obtain type (among other possibilities):*

$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm) \mid ?\,\texttt{product}(Tp).\tau\,\texttt{details}(Td)$$

where message `details` *is internally exchanged after the reception of message* `product`*. In such case, the original sequentiality information tells us that the reception of message* `details` *happens after the reception of message* `price` *and also that the emission of message* `details` *happens after the reception of message* `product`*. Since* `product` *and* `price` *are temporally unrelated,* `details` *will be specified after one or the other. Thus, the above merge may also yield:*

$$!\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).\tau\,\texttt{details}(Td) \mid ?\,\texttt{product}(Tp)$$

Such derivations are made using rule (Shuffle) which allow us to shuffle messages on the left and right hand sides, while making sure that no extra sequentiality is imposed. For instance:

$$\frac{?\,\texttt{product}(Tp).\tau\,\texttt{details}(Td) = ?\,\texttt{details}(Td) \bowtie ?\,\texttt{product}(Tp).!\,\texttt{details}(Td)}{\begin{array}{l} ?\,\texttt{price}(Tm) \mid ?\,\texttt{product}(Tp).\tau\,\texttt{details}(Td) \\ = \\ ?\,\texttt{price}(Tm).?\,\texttt{details}(Td) \bowtie ?\,\texttt{product}(Tp).!\,\texttt{details}(Td) \end{array}}$$

through (Shuffle-l), where the continuation of `price` *is moved to the continuation of* `product`*.*

Example 3.2.18 *Consider types:*

$$\begin{array}{lcl} \textit{clientRole} & \triangleq & !\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td) \\ \textit{sellerRole} & \triangleq & ?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).!\,\texttt{product}(Tp) \\ \textit{shipperRole} & \triangleq & ?\,\texttt{product}(Tp).!\,\texttt{details}(Td) \end{array}$$

which characterize the three parties involved in the purchase service collaboration (Section 2.2.2).

We have that the merge of the clientRole and sellerRole yields type:

$$\begin{aligned}&\mathit{clientSellerRole}\\&\triangleq\\&\tau\,\texttt{buy}(\mathit{Tp}).\tau\,\texttt{price}(\mathit{Tm}).(?\,\texttt{details}(\mathit{Td}) \mid !\,\texttt{product}(\mathit{Tp}))\\&\qquad = !\,\texttt{buy}(\mathit{Tp}).?\,\texttt{price}(\mathit{Tm}).?\,\texttt{details}(\mathit{Td}) \bowtie ?\,\texttt{buy}(\mathit{Tp}).!\,\texttt{price}(\mathit{Tm}).!\,\texttt{product}(\mathit{Tp})\end{aligned}$$

through rules (Plain-r), (Plain-l) and (Apart). Then we may merge clientSellerRole with shipperRole as follows:

$$\begin{aligned}&\mathit{purchaseConversation}\\&\triangleq\\&\tau\,\texttt{buy}(\mathit{Tp}).\tau\,\texttt{price}(\mathit{Tm}).\tau\,\texttt{details}(\mathit{Td}).\tau\,\texttt{product}(\mathit{Tp})\\&\qquad = \tau\,\texttt{buy}(\mathit{Tp}).\tau\,\texttt{price}(\mathit{Tm}).(?\,\texttt{details}(\mathit{Td}) \mid !\,\texttt{product}(\mathit{Tp}))\\&\qquad\quad \bowtie\\&\qquad\quad ?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td})\end{aligned}$$

derived from rules (Shuffle-l) twice, leading to the following derivation — rule (ShufflePar-l):

$$\frac{\begin{array}{c} !\,\texttt{product}(\mathit{Tp}) \mathrel{\#\!\!\!/} ?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}) \\ \tau\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}) = !\,\texttt{product}(\mathit{Tp}) \bowtie ?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}) \\ \tau\,\texttt{details}(\mathit{Td}).\tau\,\texttt{product}(\mathit{Tp}) = ?\,\texttt{details}(\mathit{Td}) \bowtie \tau\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}) \end{array}}{\begin{array}{l} \tau\,\texttt{details}(\mathit{Td}).\tau\,\texttt{product}(\mathit{Tp}) \\ = \\ ?\,\texttt{details}(\mathit{Td}) \mid !\,\texttt{product}(\mathit{Tp}) \bowtie ?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}) \end{array}}$$

On the other hand if we are to merge first sellerRole with ShipperRole we obtain type:

$$\begin{aligned}&\mathit{sellerShipperRole}\\&\triangleq\\&?\,\texttt{buy}(\mathit{Tp}).!\,\texttt{price}(\mathit{Tm}).\tau\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td})\\&\qquad = ?\,\texttt{buy}(\mathit{Tp}).!\,\texttt{price}(\mathit{Tm}).!\,\texttt{product}(\mathit{Tp}) \bowtie ?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td})\end{aligned}$$

derived via rules (Shuffle) and (Plain). We then merge clientRole with sellerShipperRole:

$$\begin{aligned}&\mathit{purchaseConversation}\\&\triangleq\\&\tau\,\texttt{buy}(\mathit{Tp}).\tau\,\texttt{price}(\mathit{Tm}).\tau\,\texttt{product}(\mathit{Tp}).\tau\,\texttt{details}(\mathit{Td})\\&\qquad = !\,\texttt{buy}(\mathit{Tp}).?\,\texttt{price}(\mathit{Tm}).?\,\texttt{details}(\mathit{Td})\\&\qquad\quad \bowtie\\&\qquad\quad ?\,\texttt{buy}(\mathit{Tp}).!\,\texttt{price}(\mathit{Tm}).\tau\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td})\end{aligned}$$

also through rules (Plain) and (Shuffle).

The type system relies on a merge relation between located types, which we define by lifting the merge between behavioral types, realized through per-conversation behavioral type merges. We first define the domain of a located type.

$$\frac{B = B_1 \bowtie B_2}{n : [B] = n : [B_1] \bowtie n : [B_2]}(MergeLoc) \quad L = L \bowtie \mathbf{0} \;\; (MergeZero\text{-}r) \quad L = \mathbf{0} \bowtie L \;\; (MergeZero\text{-}l)$$

$$\frac{\forall_{i \in 1,2}\; L_i = L_i^+ \bowtie L_i^- \quad dom(L_1) \# dom(L_2)}{L_1 \mid L_2 = L_1^+ \mid L_2^+ \bowtie L_1^- \mid L_2^-}(MergeLocPar) \qquad \frac{dom(L_1) \# dom(L_2)}{L_1 \mid L_2 = L_1 \bowtie L_2}(MergeApart)$$

Figure 3.5: Located Type Merge Relation Rules.

Definition 3.2.19 (Domain of a Located Type)
The domain of a located type L, noted $dom(L)$, is defined as follows:

$$dom(L) \triangleq \{n \;\mid\; L \equiv L' \mid n : C\}$$

Definition 3.2.20 (Located Types Merge Relation)
We say located type L is the merge of located types L_1 and L_2, noted $L = L_1 \bowtie L_2$, if we may derive $L = L_1 \bowtie L_2$ via the rules given in Figure 3.5.

The merge relation is then used to explain how the behaviors of process can be combined, in a compositional way. Ultimately, such behavioral combinations result in a *closed* type, which capture systems that have no open external dependencies.

Definition 3.2.21 (Closed Types)
We say a behavioral type B is closed, noted $closed(B)$, if for any message type $(p\,l(C))$ such that $(p\,l(C)) \in Msg_{\mathcal{L}}(B)$ then either $p = \tau$, or $p = ?$ and $l \in \mathcal{L}_\star$. We say a located type L is closed, noted $closed(L)$, if for any behavioral type B such that $L \equiv L' \mid n : [B]$ then $closed(B)$.

Closed behavioral types characterize processes that have matching receives for all sends. Closed types are thus defined exclusively on messages of polarity τ, except for shared input message types that are still open to match with further outputs.

3.3 Type System

In this section we present the typing rules that associate conversation types to systems specified in the π_{lab}-Calculus presented in Chapter 2. Processes specified in π_{lab}-Calculus interact via channel-based communication, where each communication is indexed by a label. Labeled channels may thus implement our notion of conversations: a single medium of communication where we may structure the several interactions (or message exchanges), using the label to uniquely identify them. We associate conversation types to π_{lab}-Calculus processes so as to discipline the interaction in each channel, via protocols defined over the labels, in a set of typing rules. When we succeed to associate to a process a conversation type, we say that the process is well typed, being such well-typed processes the object of the results that express our safety properties.

We informally describe the typing rules. The following rule types the parallel composition of two processes by merging the types of the two parallel branches:

$$\frac{P :: L_1 \quad Q :: L_2}{P \mid Q :: L_1 \bowtie L_2}$$

By merging the two types we are then able to describe all the possible behaviors of the concurrent processes, including interactions with the external environment and internal synchronizations between the parallel processes — represented by τ message types in the resulting merge. Rules:

$$\frac{}{\mathbf{0} :: \star L^!} \qquad\qquad \frac{}{\mathcal{X} :: \star L^!}$$

type the inactive process and the recursion variable with an exponential output located type. This means the inactive process/recursion variable can be used in a context where a process requiring some specific exponential behavior is expected (cf. subtyping), paving the way for a weakening property (Proposition 3.3.3). The rule for the recursive definition:

$$\frac{P :: \star L^!}{\mathbf{rec}\, \mathcal{X}.P :: \star L^!}$$

types a recursive process if the body of the recursion is characterized by such an exponential output type. The rule profits from the subtyping relation, as it allows for particular message types (outputs on shared labels) to be lifted to the more general unbounded case (*SubExpWeak*), thus allowing for systems that exhibit a single output in such messages to fit in the requirements of the premise of the rule. Intuitively, this rule characterizes systems that have as an external interface a number of calls to remote services, and that interact with such service instances using freshly created names, which type is no longer visible in the derivation, as specified by the rule:

$$\frac{P :: L \mid a : [B] \quad (\mathit{closed}(B), a \notin \mathit{dom}(L))}{(\boldsymbol{\nu} a)P :: L}$$

which types a process that specifies a restricted name, by checking the behavior prescribed for the restricted conversation is *closed*, so as to avoid hiding conversation names where unmatched communications still persist (necessary to ensure deadlock absence). Rule:

$$\frac{P :: L \bowtie n : [B] \quad (!\, l(C) \# B)}{n \cdot l!(m).P :: (L \bowtie n : [!\, l(C).B]) \bowtie m : C}$$

types the output prefixed process. The premise states that process P has some composite type which is the result of a merge between located behavior L and behavior B at n. Then, in the conclusion of the rule, the output prefixed process is characterized by the type that now specifies the output message type with continuation B in conversation n. Such merges in premise and conclusion are essential to support challenging specifications where, through aliasing, behaviors that a process initially expects from the external environment end up actually being implemented internally to the process (e.g., two roles of a conversation that dynamically end up being realized by the same party).

Notice the type of the name m being sent is introduced as a separate $\bowtie$ view of the context (as well as declared in the argument type of the message), which means that the type being sent may actually be some separate part of the type of some conversation, which will be (partially) delegated away. This mechanism is crucial to allow external partners to join in on ongoing conversations in a disciplined way.

The rule for the input prefixed process:

$$\frac{P :: (L \bowtie n : [B]) \mid x : C \quad (?\, l(C) \# B, x \notin \mathit{dom}(L) \cup \{n\},\ l \in \mathcal{L}_p)}{n \cdot l?(x).P :: L \bowtie n : [?\, l(C).B]}$$

$$\frac{P :: L_1 \quad Q :: L_2}{P \mid Q :: L_1 \bowtie L_2}(Par) \qquad \frac{}{\mathbf{0} :: \star L^!}(Stop)$$

$$\frac{P :: \star L^!}{\textsf{rec}\, \mathcal{X}.P :: \star L^!}(Rec) \qquad \frac{}{\mathcal{X} :: \star L^!}(RecVar)$$

$$\frac{P :: L \mid a : [B] \quad (closed(B), a \notin dom(L))}{(\boldsymbol{\nu} a)P :: L}(Res)$$

$$\frac{P :: L \bowtie n : [B] \quad (!\, l(C) \# B)}{n \cdot l!(m).P :: (L \bowtie n : [!\, l(C).B]) \bowtie m : C}(Output)$$

$$\frac{P :: (L \bowtie n : [B]) \mid x : C \quad (?\, l(C) \# B, x \notin dom(L) \cup \{n\},\ l \in \mathcal{L}_p)}{n \cdot l?(x).P :: L \bowtie n : [?\, l(C).B]}(Input)$$

$$\frac{P :: \star L^! \mid x : C \quad (x \notin dom(L))}{\textsf{rec}\, \mathcal{X}.n \cdot l?(x).(\mathcal{X} \mid P) :: \star L^! \bowtie n : [\star ?\, l(C)]}(InputRec)$$

$$\frac{P :: L_1 \quad L_1 <: L_2}{P :: L_2}(Sub)$$

Figure 3.6: Typing Rules.

is similar to the output rule, being the only difference the specification of the type for the received name in the premise, which then allows process P to specify some behavior in such received name, accordingly to the argument type of the message. Notice this rule types exclusively inputs defined on plain labels. The following rule types inputs defined on shared labels:

$$\frac{P :: \star L^! \mid x : C \quad (x \notin dom(L))}{\textsf{rec}\, \mathcal{X}.n \cdot l?(x).(\mathcal{X} \mid P) :: \star L^! \bowtie n : [\star ?\, l(C)]}$$

which essentially extends the rule for the general recursive process allowing for an input message to be at the head of the recursive behavior. Although shared labeled inputs can only be typed via this last rule, the rule itself also accommodates plain labeled inputs. We may now present the definition of our typing relation between π_{lab}-Calculus processes and basic conversation types.

Definition 3.3.1 (Well-Typed π_{lab}-Calculus Processes)
We say a π_{lab}-Calculus process P is well typed if there is a located type L such that $P :: L$ can be derived by the rules given in Figure 3.6.

In the next section we show the results that assert some properties of such well-typed systems.

3.3.1 Type Safety

We next present our results that ensure a notion of safety for well-typed processes. In short, we show that well-typed processes are free from particular errors, and that well-typed processes always evolve to well-typed processes, hence well-typed processes are free from errors throughout their evolution. The notion of preservation of typing under evolution is captured by subject reduction (Theorem 3.3.5), and it is defined relying on a notion of reduction on types, since each reduction step at the process level may require a modification in the typing, as expected from a behavioral type system.

$$\frac{}{n : [\tau\, l(C).B] \to n : [B]}(\mathit{RedTau}) \qquad \frac{L_1 \to L_2}{L_1 \mid L_3 \to L_2 \mid L_3}(\mathit{RedCongPar})$$

$$\frac{L_1 \equiv L_1' \to L_2' \equiv L_2}{L_1 \to L_2}(\mathit{RedEquiv}) \qquad L \to L \;\; (\mathit{RedReflex})$$

Figure 3.7: Located Types Reduction Rules.

Definition 3.3.2 (Located Types Reduction)
We say located type L_1 reduces to located type L_2, noted $L_1 \to L_2$, if it $L_1 \to L_2$ is derivable by the rules given in Figure 3.7.

Essentially, each evolution of the process will be explained at the level of the type by the activation of the continuation of a τ message prefix (*RedTau*). The other rules specify a congruence principle for parallel composition of types (*RedCongPar*), closure of reduction under type equivalence (*RedEquiv*), and closure of the reduction relation under reflexivity (*RedReflex*). The last case is necessary to capture reduction under restricted conversations (which type is not visible).

We may now present our main soundness results. We start by stating a weakening property that supports the introduction of exponential output located types in the typing derivation.

Proposition 3.3.3 (Weakening)
Let P be a well-typed process such that $P :: L_1$. If exponential output located type $\star L_2^!$ is such that L_1 and $\star L_2^!$ are apart, hence $L_1 \# \star L_2^!$, then $P :: L_1 \mid \star L_2^!$.

Proof. By induction on the length of the derivation of $P :: L_1$. Type $\star L_2^!$ is introduced at the level of the axioms (*Stop*) and (*RecVar*), and pushed down in the derivation separately, being the independence guaranteed by the apartness $L_1 \# \star L_2^!$ (see Appendix A.3). ■

The following auxiliary result regards the typing of processes up to substitution.

Lemma 3.3.4 (Substitution Lemma)
Let P be a well-typed process such that $P :: L \mid x : C$ and $x \notin dom(L)$. If there is type L' such that $L' = L \bowtie a : C$ then $P\{x/a\} :: L'$.

Proof. By induction on the length of the derivation of $P :: L \mid x : C$ (see Appendix A.1). ■

The substitution lemma plays a crucial role in proving the preservation of typing under evolution, since synchronizations allow for names to be passed around, so the Lemma will be used to prove that the body of an input is well-typed after the variable instantiation takes place. The condition that the "new" type $a : C$ can be merged back into the "old" overall type L is enough to ensure this, and will be guaranteed at the level of the typing of the process emitting the name.

We may now state our subject reduction result.

Theorem 3.3.5 (Subject Reduction)
Let P be a well-typed process such that $P :: L$. If $P \to Q$ then there is $L \to L'$ such that $Q :: L'$.

Proof. By induction on the length of the derivation of $P \to Q$ (see Appendix A.1). ■

Subject reduction thus provides the guarantee that well-typed processes always evolve to well-typed processes. We now turn to characterizing the error processes that our type system avoids,

so as to arrive at our safety result which guarantees that such error processes are unreachable from well-typed processes. To define error processes we introduce static process contexts and an auxiliary notion of communication prefixed processes.

Definition 3.3.6 (π_{lab}-Calculus Static Context)
π_{lab}-Calculus static process contexts, noted $\mathcal{C}[\cdot]$, are defined as follows:

$$\mathcal{C}[\cdot] \ ::= \ (\boldsymbol{\nu} a)\mathcal{C}[\cdot] \ \big| \ P \mid \mathcal{C}[\cdot] \ \big| \ \cdot \qquad (\pi_{lab}\text{-Calculus Static Context})$$

Definition 3.3.7 (π_{lab}-Calculus Prefix Process)
We denote by $c \cdot l-.P$ a prefix process that is ready to communicate on $c \cdot l$, defined as follows:

$$c \cdot l-.P \ ::= \ c \cdot l!(\tilde{a}).P \ \big| \ c \cdot l?(\tilde{x}).P \qquad (\pi_{lab}\text{-Calculus Prefix Process})$$

We define error processes as processes that exhibit a race in a communication defined with a plain label, using static contexts to capture any subprocess where a race may happen, including under name restriction, and using a prefix process to interact with such competing communications.

Definition 3.3.8 (Error Process)
We say P is an error process if there is a context $\mathcal{C}$, processes Q, R with $P = \mathcal{C}[Q \mid R]$, and name c and labels $l, \texttt{flag}$ with $l \in \mathcal{L}_p$ and $\texttt{flag} \notin fl(P)$, such that:

$$\mathcal{C}[Q \mid c \cdot l-.c \cdot \texttt{flag}!()] \rightarrow \mathcal{C}'[c \cdot \texttt{flag}!()] \quad \textit{and} \quad \mathcal{C}[R \mid c \cdot l-.c \cdot \texttt{flag}!()] \rightarrow \mathcal{C}'[c \cdot \texttt{flag}!()]$$

In such way, we "behaviorally" characterize an error as a process that has distinct subprocesses able to synchronize with the same communication prefix, where such prefix is defined on a plain label (from $\mathcal{L}_p$). Thus, a process is not an error only if for each possible immediate interaction in a plain message there is at most a single sender and a single receiver.

Proposition 3.3.9 (Error Freeness)
If P is a well-typed process then P is not an error process.

Proof. Follows by auxiliary results and by the definition of merge (see Appendix A.1). ■

Combining error freeness (Proposition 3.3.9) with subject reduction (Theorem 3.3.5), we conclude that any process reachable from a well-typed process is not an error.

Corollary 3.3.10 (Type Safety)
Let P be a well-typed process. If there is Q such that $P \xrightarrow{*} Q$, then Q is not an error process.

Proof. Immediate from Theorem 3.3.5 and Proposition 3.3.9. ■

Our type safety result ensures that, in any reduction sequence arising from a well-typed process, for each plain-labeled message ready to communicate there is always at most a unique input/output outstanding synchronization. More: arbitrary interactions in shared labels do not invalidate this invariant. Another consequence of subject reduction (Theorem 3.3.5) is that any message exchange inside the process must be explained by a τM prefix in the related conversa-

tion type (via type reduction), thus implying conversation fidelity, i.e., all conversations follow the prescribed protocols.

3.3.2 Typing the Purchase Example

In this section we show the typing derivation for the purchase scenario running example presented in the Introduction and implemented in π_{lab}-Calculus in Section 2.2.2. Consider the code:

$$
\begin{array}{l}
\mathit{PurchaseSystem} \triangleq \\
\quad (\boldsymbol{\nu}\, \mathit{buyChat}) \\
\quad (\mathit{Seller} \cdot \mathtt{BuyService}!(\mathit{buyChat}). \\
\qquad \mathit{buyChat} \cdot \mathtt{buy}!(\mathit{prod}).\, \mathit{buyChat} \cdot \mathtt{price}?(p).\, \mathit{buyChat} \cdot \mathtt{details}?(d)\ ) \\
\quad \mid \mathit{PriceDB} \mid \\
\quad \mathit{Seller} \cdot \mathtt{BuyService}?(x). \\
\qquad x \cdot \mathtt{buy}?(\mathit{prod}).\, \mathit{Seller} \cdot \mathtt{askPrice}!(\mathit{prod}). \\
\qquad \mathit{Seller} \cdot \mathtt{priceVal}?(p).\, x \cdot \mathtt{price}!(p). \\
\qquad \mathit{Shipper} \cdot \mathtt{DeliveryService}!(x).\, x \cdot \mathtt{product}!(\mathit{prod}) \\
\quad \mid \\
\quad \mathit{Shipper} \cdot \mathtt{DeliveryService}?(y). \\
\qquad y \cdot \mathtt{product}?(p).\, y \cdot \mathtt{details}!(\mathit{data})
\end{array}
$$

which implements the three-party service collaboration. The overall (choreographic) description that governs the interactions in this service collaboration is given by the type (considering some basic types Tp, Tm and Td):

$$\tau\,\mathtt{buy}(Tp).\tau\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)$$

which specifies a protocol that consists in the sequence of interactions `buy`, `price`, `product` and `details`. Given the overall protocol, we now show how our types ensure that all three participants follow the prescribed protocol. For starters, by looking at the process that instantiates the `BuyService` (the client), we may verify that its role in the collaboration is given by the code:

$$P \triangleq \mathit{buyChat} \cdot \mathtt{buy}!(\mathit{prod}).\, \mathit{buyChat} \cdot \mathtt{price}?(p).\, \mathit{buyChat} \cdot \mathtt{details}?(d)$$

for which we may derive the following typing:

$$P :: \mathit{buyChat} : [\ !\,\mathtt{buy}(Tp).?\,\mathtt{price}(Tm).?\,\mathtt{details}(Td)\ ]$$

which characterizes the interactions the process has in conversation *buyChat*, specifying that it first sends message `buy`, then receives message `price` and finally receives message `details`. As for the process that publishes the *BuyService* (the seller) we can observe by looking at the code that its (individual) role in the service collaboration is given by the code:

$$Q \triangleq x \cdot \mathtt{buy}?(\mathit{prod}).\, x \cdot \mathtt{price}!(p).\, x \cdot \mathtt{product}!(\mathit{prod})$$

where we focus solely on the interactions relative to the service conversation, leaving out the interactions with the price database and the call to service `DeliveryService`. So the role of the

seller in the service collaboration is characterized by the typing:

$$Q :: x : [\ ?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).!\,\texttt{product}(Tp)\ ]$$

Finally, by looking at the code of the process that publishes `DeliveryService` (the shipper) we may observe that its role in the service collaboration is given by the code:

$$R \triangleq y \cdot \texttt{product}?(p).\, y \cdot \texttt{details}!(data)$$

for which we may derive the following typing:

$$R :: y : [\ ?\,\texttt{product}(Tp).!\,\texttt{details}(Td)\ ]$$

We then have three distributed participants realizing different parts of the overall protocol. If we are to take the behavioral types that characterize the three distinct roles in the conversation and behaviorally combine — merge — them we obtain the intended prescribed protocol:

$$\begin{array}{l} \tau\,\texttt{buy}(Tp).\tau\,\texttt{price}(Tm).\tau\,\texttt{product}(Tp).\tau\,\texttt{details}(Td) \\ = \\ !\,\texttt{buy}(Tp).?\,\texttt{price}(Tm).?\,\texttt{details}(Td) \\ \qquad\qquad \bowtie\ ?\,\texttt{buy}(Tp).!\,\texttt{price}(Tm).!\,\texttt{product}(Tp) \\ \qquad\qquad\qquad\qquad \bowtie\ ?\,\texttt{product}(Tp).!\,\texttt{details}(Td) \end{array}$$

So, in fact the three participants will follow the prescribed protocol of interaction. To statically guarantee this fact, we must observe how the several behaviors are to be delegated throughout system evolution. For starters, we observe that the `DeliveryService` receives a name and interacts in it according to $?\,\texttt{product}(Tp).?\,\texttt{details}(Td)$, and therefore we type the shipper process as follows:

$$\begin{array}{l} Shipper \cdot \texttt{DeliveryService}?(y).R \\ :: \\ Shipper : [\ ?\,\texttt{DeliveryService}(?\,\texttt{product}(Tp).!\,\texttt{details}(Td))\ ] \end{array}$$

We thus have that the behavior delegated by the seller to the shipper at the moment seller asks shipper to join the service interaction is defined in the `DeliveryService` message argument type. We then have the following typing characterization (by rule (*Output*)):

$$\dfrac{x \cdot \texttt{product}!(prod) :: x : [\ !\,\texttt{product}(Tp)\ ] \bowtie Shipper : [\ \mathbf{0}\ ]}{\begin{array}{l} Shipper \cdot \texttt{DeliveryService}!(x).\ x \cdot \texttt{product}!(prod) \\ :: \\ Shipper : [\ !\,\texttt{DeliveryService}(?\,\texttt{product}(Tp).!\,\texttt{details}(Td)).\mathbf{0}\ ] \\ \mid \\ (x : [\ !\,\texttt{product}(Tp)\ ] \bowtie x : [\ ?\,\texttt{product}(Tp).!\,\texttt{details}(Td)\ ]) \end{array}}$$

which specifies a call to `DeliveryService` with the corresponding type, and where the type of x is determined by the merge of the delegated behavior with the behavior specified for x in the continuation process $x \cdot \texttt{product}!(prod)$. We may then consider the result of the merge and

obtain the following typing:

$$\begin{array}{l} \mathit{Shipper} \cdot \mathtt{DeliveryService!}(x).\ x \cdot \mathtt{product!}(\mathit{prod}) \\ :: \\ \mathit{Shipper} : [\ \mathtt{!\,DeliveryService(?\,product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{))}\ ] \\ \mid \\ x : [\ \tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{)}\ ] \end{array}$$

Given this, we may now type the whole seller process as follows:

$$\begin{array}{l} \mathit{Seller} \cdot \mathtt{BuyService?}(x). \\ \quad x \cdot \mathtt{buy?}(\mathit{prod}).\ \mathit{Seller} \cdot \mathtt{askPrice!}(\mathit{prod}). \\ \quad \mathit{Seller} \cdot \mathtt{priceVal?}(p).\ x \cdot \mathtt{price!}(p). \\ \quad \mathit{Shipper} \cdot \mathtt{DeliveryService!}(x).\ x \cdot \mathtt{product!}(\mathit{prod}) \\ :: \\ \mathit{Seller} : [\ \mathtt{?\,BuyService(?\,buy(}\mathit{Tp}\mathtt{).!\,price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{))} \\ \qquad \mid \mathtt{!\,askPrice(}\mathit{Tp}\mathtt{).?\,priceVal(}\mathit{Tm}\mathtt{)}\ ] \\ \mid \\ \mathit{Shipper} : [\ \mathtt{!\,DeliveryService(?\,product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{))}\ ] \end{array}$$

which specifies that the behavior delegated to the `BuyService` upon a service call is:

$$\mathtt{?\,buy(}\mathit{Tp}\mathtt{).!\,price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{)}$$

which characterizes the subsystem consisting on seller and shipper, resulting from the behavioral combination of the types that represent the roles of seller and shipper in the service interaction (where the role of the shipper will be dynamically delegated). The type also indicates the protocol between the service code and the price database $\mathtt{!\,askPrice(}\mathit{Tp}\mathtt{).?\,priceVal(}\mathit{Tm}\mathtt{)}$ which we choose to derive in parallel. Finally we may type the buyer process as follows:

$$\frac{P :: \mathit{buyChat} : [\ \mathtt{!\,buy(}\mathit{Tp}\mathtt{).?\,price(}\mathit{Tm}\mathtt{).?\,details(}\mathit{Td}\mathtt{)}\ ]}{\begin{array}{l} \mathit{Seller} \cdot \mathtt{BuyService!}(\mathit{buyChat}).P \\ :: \\ \mathit{Seller} : [\ \mathtt{!\,BuyService(?\,buy(}\mathit{Tp}\mathtt{).!\,price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{))}\ ] \\ \mid \\ \mathit{buyChat} : [\ \mathtt{!\,buy(}\mathit{Tp}\mathtt{).?\,price(}\mathit{Tm}\mathtt{).?\,details(}\mathit{Td}\mathtt{)}\ ] \\ \bowtie \mathit{buyChat} : [\ \mathtt{?\,buy(}\mathit{Tp}\mathtt{).!\,price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{)}\ ] \end{array}}$$

where the result of the merge is the prescribed conversation protocol for *buyChat*, hence:

$$\begin{array}{l} \mathit{Seller} \cdot \mathtt{BuyService!}(\mathit{buyChat}).P \\ :: \\ \mathit{Seller} : [\ \mathtt{!\,BuyService(?\,buy(}\mathit{Tp}\mathtt{).!\,price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).!\,details(}\mathit{Td}\mathtt{))}\ ] \\ \mid \\ \mathit{buyChat} : [\ \tau\,\mathtt{buy(}\mathit{Tp}\mathtt{).}\tau\,\mathtt{price(}\mathit{Tm}\mathtt{).}\tau\,\mathtt{product(}\mathit{Tp}\mathtt{).}\tau\,\mathtt{details(}\mathit{Td}\mathtt{)}\ ] \end{array}$$

Given that the type of *buyChat* is a closed type, we may type the restriction via rule (*Res*), which elides the *buyChat* type from the derivation:

$$(\boldsymbol{\nu} buyChat)(\ Seller \cdot \mathtt{BuyService!}(buyChat).P\ )$$
$$::$$
$$Seller : [\ ?\,\mathtt{BuyService}(?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).!\,\mathtt{details}(Td))\ ]$$

We thus have the following typing for the entire system:

$$PurchaseSystem$$
$$::$$
$$Seller : [\ \tau\,\mathtt{BuyService}(?\,\mathtt{buy}(Tp).!\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).!\,\mathtt{details}(Td)) \mid \tau\,\mathtt{askPrice}(Tp).\tau\,\mathtt{priceVal}(Tm)\ ]$$
$$\mid$$
$$Shipper : [\ \tau\,\mathtt{DeliveryService}(?\,\mathtt{product}(Tp).!\,\mathtt{details}(Td))\ ]$$

specifying the two internal (τ) service interactions defined at *Seller* and at *Shipper* and the internal interaction protocol with the price database (assuming the *PriceDB* is typed as expected).

We can then assert, in the light of the type safety results shown in Section 3.3.1, that all conversations in the purchase system follow the prescribed protocols.

3.4 Remarks

In this section we present some notes on closely related work and assess our development along the way. We present a summary comparison between conversation types and classical dyadic session types and leave a more detailed comparison with multiparty session types for the conclusion of Chapter 5, where we also discuss, in broad lines, how our type system compares with more general behavioral type systems.

Session types, as introduced by Honda [54] and by Honda et al. [55], describe the single-threaded interaction between two parties, while conversation types capture the, possibly concurrent, set of interactions between multiple parties. Thus, dyadic session types are a particular case of conversation types. Consider the following specification:

$$(\boldsymbol{\nu} session)$$
$$(\ chat \cdot l!(session).\ session \cdot l_1!(\mathtt{hi}).\ session \cdot l_2?(y)\ )$$
$$\mid$$
$$chat \cdot l?(x).\ x \cdot l_1?(z).\ x \cdot l_2!(\mathtt{bye})$$

where we model a (single-threaded) two party interaction, characterized by conversation type:

$$chat : [\tau\, l(?\, l_1(\mathtt{string}).!\, l_2(\mathtt{string}))]$$

where, if we abstract away from the labels, we may recognize a session-type like specification. Labels are essential to support communication between multiple parties in a single medium: if three parties are ready to communicate on a channel and there is no label to distinguish the communications then we immediately have an undesired race. We view our approach as more fundamental than most session-type presentations, since we avoid introducing session initiation

primitives as native in the language (see [92] for an attempt).

On the one hand sessions are a medium for two-party interaction, where session participants access the session through a session endpoint. On the other hand conversations are also a single medium but for multiparty interaction, where any of the conversation participants accesses the conversation through a conversation endpoint (access piece). Session channels support single-threaded interaction protocols between the two session participants. Conversation contexts, on the other hand, support concurrent interaction protocols between multiple participants. Sessions always have two endpoints, created at session initialization. Participants can delegate their participation in a session, but the delegation is full in the sense that the delegating party loses access to the session. Conversations also initially have two endpoints. However the number of endpoints may increase (decrease) as participants join (leave) ongoing conversations. Participants can ask a party to join a conversation and not lose access to it (partial delegation).

Since there are only two session participants, session types may describe the entire protocol by describing the behavior of just one of the participants (the type of the other participant is dual). Conversations types, on the other hand, describe the interactions between multiple parties, so they specify the entire conversation protocol (a choreography description) that decomposes in the types of the several participants (e.g., $B = B_{buyer} \bowtie B_{seller} \bowtie B_{shipper}$). Duality of sessions is a particular case of a behavioral merge: two (τ-free) types B_1, B_2 are dual if there is B such that $B = B_1 \bowtie B_2$ and $closed(B)$ — i.e., if we are to behaviorally combine two τ-free types (local types that specify only outputs and inputs) and obtain a closed type then such types are dual.

On a more technical perspective, we remark that some session-typed based approaches have developed interesting theories and technical artifacts that we did not explore. For instance, we adopt an equi-recursive approach to recursive types, while in the work of Gay et al. [48] the authors consider a more flexible theory based on coinductive definitions. Also, the works of Castagna et al. [34] and Padovani [80] provide with notions of semantic subtyping for session types, which allow for a more general presentation of the subtyping relation relying on behavioral equivalences between types. For simplicity, we restricted our development to consider only monadic messages. We expect our techniques can be extended to the polyadic case, in which case we would also exclude arity mismatch errors.

The structure of the basic conversation types language already elucidates on what is the sort of communication model we are aiming at: interaction in a system is described by a number of conversations, each one grouping a set of related interactions in a structured way. Building on this abstract notion of conversation we propose a model for service-oriented computing, which includes primitives that support the access to conversations and that provide a means to specify interaction in such conversations in a simple and natural way, described next.

Chapter 4

The Conversation Calculus

In this chapter we introduce the (core) Conversation Calculus, our proposal for modeling service-oriented systems, introduced in [95]. We start by motivating the general ideas that are at the basis of the constructs of the language, after which we present the syntax and operational semantics of the language. Then, we present a basic study of the behavioral semantics of the language, where we show some interesting theoretical properties of our model — namely that the standard definition of behavioral equivalence is a congruence for all operators of the language. We then show how to program some examples, using some useful idioms that offer a convenient abstraction for some typical service-oriented programming patterns. We conclude the chapter by assessing our work through a comparison with related approaches.

4.1 Modeling Service-Oriented Computation

At the basis of our development we find the abstract notion of a conversation: a set of related interactions between multiple parties. Such an abstract notion may be implemented in different ways, as, for example, by using correlation tokens to encode conversations (cf., web-service technology): a special nonce is inserted in a message so as to uniquely identify the service interaction to which the message belongs to. Some formal models have been introduced that are based on such notion of correlation (see [51, 61, 71]). Others prefer to encapsulate such a set of related interactions in a dedicated medium, a session (e.g., [12, 13, 68]), however sessions do not support multiparty interaction. Our approach is closest to the session-based approach, since we also encapsulate the set of related interactions in the same medium of communication. However, differently from sessions, conversations support multiparty interaction.

Consider the following π_{lab}-Calculus specification:

$$chat \cdot \texttt{hi}!().chat \cdot \texttt{bye}?()$$

where a process is communicating in channel *chat* messages labeled by `hi` and `bye`. This process is specifying a protocol for a party interacting in conversation *chat*. Since the two messages are related and belong to the same conversation we may as well write them as:

$$\texttt{hi}!().\texttt{bye}?()$$

and somehow indicate they are to be exchanged in conversation *chat*: in the Conversation

Calculus this is achieved by placing such message protocol specification inside a conversation access piece:

$$chat \blacktriangleleft [\, \texttt{hi}!().\texttt{bye}?() \,]$$

We thus specify that first a message `hi` is to be sent and then a message `bye` is to be received in conversation *chat*. We are then able to encapsulate a set of related interactions by placing such protocol specifications in access pieces of the same conversation, for instance:

$$chat \blacktriangleleft [\, \texttt{hi}!().\texttt{bye}?() \,] \mid chat \blacktriangleleft [\, \texttt{hi}?().\texttt{bye}!() \,]$$

Moreover, we may specify as many accesses to a single conversation as needed. So, for instance we might have a parallel protocol between four parties in conversation *chat*:

$$\begin{array}{l} chat \blacktriangleleft [\, \texttt{hi}!().\texttt{bye}?() \,] \mid chat \blacktriangleleft [\, \texttt{hi}?().\texttt{bye}!() \,] \\ \mid \\ chat \blacktriangleleft [\, \texttt{hello}!().\texttt{goodbye}?() \,] \mid chat \blacktriangleleft [\, \texttt{hello}?().\texttt{goodbye}!() \,] \end{array}$$

or a protocol that sequentially involves three different parties:

$$chat \blacktriangleleft [\, \texttt{hi}!().\texttt{bye}?() \,] \mid chat \blacktriangleleft [\, \texttt{hi}?().\texttt{hello}!() \,] \mid chat \blacktriangleleft [\, \texttt{hello}?().\texttt{bye}!() \,]$$

Since we intend to model dynamic collaborations, which is to say systems where parties are, at runtime, asked to participate in a collaboration, it is essential that we have some way of capturing the dynamic acquisition of the shared medium of communication. We achieve this by allowing conversation names to be passed in communications (just like names are passed in communications in the π-Calculus). So for instance we may specify:

$$\begin{array}{l} chat \blacktriangleleft [\, \texttt{private}!(privateChat).privateChat \blacktriangleleft [\, \texttt{hi}!() \,] \,] \\ \mid \\ chat \blacktriangleleft [\, \texttt{private}?(x).x \blacktriangleleft [\, \texttt{hi}?() \,] \,] \end{array}$$

where the process on the bottom receives a name (in message `private` in conversation *chat*) and then uses this name to access the conversation to which he previously had no access to (we typically ensure the name is freshly created by a name restriction — $(\boldsymbol{\nu} privateChat)$).

These are the basic ingredients of the language, which may be viewed as a specialization of the π-Calculus, introduced by Milner et al. [79], so as to address specific issues of service-oriented computation, namely to the particular setting of modeling multiparty interaction. In the remaining of the chapter we present the syntax of the language (Section 4.2) and provide a mathematical interpretation of the language constructs by means of a labeled transition system (Section 4.3). Then, we prove that the standard notion of behavioral equivalence induced by the labeled transition system — bisimilarity — enjoys some essential properties, namely that bisimilarity is a congruence for all operators of the language, thus showing that our syntactically chosen constructs are proper functions at the level of the abstract behavior (Section 4.4). We then turn to a more practical perspective and illustrate the expressiveness of the language by programming a couple of examples, where we make use of higher level service-oriented idioms that we show can be encoded in more basic communication primitives in our language (Section 4.5).

$$
\begin{array}{lcll}
a, b, c, \ldots & \in & \Lambda & \text{(Names)} \\
x, y, z, \ldots & \in & \mathcal{V} & \text{(Variables)} \\
n, m, o, u, v \ldots & \in & \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
l, s \ldots & \in & \mathcal{L} & \text{(Labels)} \\
\mathcal{X}, \mathcal{Y}, \ldots & \in & \chi & \text{(Recursion Variables)}
\end{array}
$$

$$
\begin{array}{lcll}
P, Q, R & ::= & \mathbf{0} & \text{(Inaction)} \\
 & | & P \mid Q & \text{(Parallel Composition)} \\
 & | & (\boldsymbol{\nu} a)P & \text{(Name Restriction)} \\
 & | & \mathbf{rec}\, \mathcal{X}.P & \text{(Recursion)} \\
 & | & \mathcal{X} & \text{(Variable)} \\
 & | & n \blacktriangleleft [P] & \text{(Conversation Access)} \\
 & | & \Sigma_{i \in I}\ \alpha_i.P_i & \text{(Prefix Guarded Choice)}
\end{array}
$$

$$
\begin{array}{lcll}
d & ::= & \downarrow \mid \uparrow & \text{(Directions)} \\
\alpha & ::= & l^d!(n_1, \ldots, n_k) & \text{(Output)} \\
 & | & l^d?(x_1, \ldots, x_k) & \text{(Input)} \\
 & | & \mathbf{this}(x) & \text{(Conversation Awareness)}
\end{array}
$$

Figure 4.1: The Core Conversation Calculus Syntax.

4.2 The Core Conversation Calculus

In this section we first define the syntax of the core Conversation Calculus, then we describe in detail the individual constructs of the language. In a nutshell, the core Conversation Calculus extends the static fragment of the π-Calculus with the conversation construct $n \blacktriangleleft [P]$, and replaces channel based communication with context-sensitive message based communication.

We assume given an infinite set of names Λ, an infinite set of variables $\mathcal{V}$, an infinite set of labels $\mathcal{L}$, and an infinite set of recursion variables χ. We next define the syntax of the language.

Definition 4.2.1 (CC Syntax)
The syntax of the Core Conversation Calculus is given in Figure 4.1.

The static fragment is defined by the inaction $\mathbf{0}$, parallel composition $P \mid Q$, name restriction $(\boldsymbol{\nu} a)P$ and recursion $\mathbf{rec}\, \mathcal{X}.P$. The conversation access construct $n \blacktriangleleft [P]$ allows a process to initiate interactions, as specified by P, in the conversation n.

Communication is expressed by the guarded choice construct $\Sigma_{i \in I}\ \alpha_i.P_i$, which expresses that the process may select some initial action α_i and then progress as P_i. Communication actions are of two forms: $l^d!(n_1, \ldots, n_k)$ for sending messages (e.g., $\mathtt{askPrice}^{\uparrow}!(prod)$) and $l^d?(x_1, \ldots, x_k)$ for receiving messages (e.g., $\mathtt{price}^{\downarrow}?(p)$), where the variables in $x_1, \ldots, x_k$ are pairwise distinct. Message communication is thus defined by the label l and the direction d. There are two message directions: $\downarrow$ (read "here") meaning that the interaction should take place in the current conversation, and $\uparrow$ (read "up") meaning that the interaction should take place in the caller conversation. A basic action may also be of the form $\mathbf{this}(x)$, allowing the process to dynamically access the identity of the current conversation.

Notice that message labels (from $\mathcal{L}$) are not names but free identifiers (cf. record labels or XML tags), and therefore are not subject to fresh generation, restriction or binding. Also,

message labels are not communicated in messages. Only conversation names may be subject to binding, freshly generated via $(\boldsymbol{\nu} a)P$, and be sent in messages.

To lighten notation we introduce the following syntactic conventions and abbreviations.

Syntactic conventions:

$\textsf{rec}\, \mathcal{X}.P \mid Q$ is to be read as $(\textsf{rec}\, \mathcal{X}.P) \mid Q$.

$(\boldsymbol{\nu} a)P \mid Q$ is to be read as $((\boldsymbol{\nu} a)P) \mid Q$.

$\alpha.P \mid Q$ is to be read as $(\alpha.P) \mid Q$.

Abbreviations:

$(\boldsymbol{\nu} \tilde{a})P$ and $(\boldsymbol{\nu} a_1, \ldots, a_k)P$ stand for $(\boldsymbol{\nu} a_1) \ldots (\boldsymbol{\nu} a_k)P$.

$l^d!(\tilde{n}).P$ stands for $l^d!(n_1, \ldots, n_k).P$.

$l^d?(\tilde{x}).P$ stands for $l^d?(x_1, \ldots, x_k).P$.

S $(S_1, S_2, \ldots)$ stands for $\Sigma_{i \in I}\ \alpha_i.P_i$.

$l!(\tilde{x}).P$ stands for $l^{\downarrow}!(\tilde{x}).P$.

$l?(\tilde{x}).P$ stands for $l^{\downarrow}?(\tilde{x}).P$.

α stands for $\alpha.\mathbf{0}$, where appropriate.

In order to present the operational semantics of the language we first need to introduce some auxiliary operators that collect some syntactic information of processes, that will be used to syntactically manipulate the processes when we are characterizing how they evolve. Namely, we introduce the sets of free and bound names of a process, the set of free labels, and the sets of free variables and free recursion variables. The distinguished occurrences of a, $\tilde{x}$, x and $\mathcal{X}$ are binding occurrences in $(\boldsymbol{\nu} a)P$, $l^d?(\tilde{x}).P$, $\textsf{this}(x).P$, and $\textsf{rec}\, \mathcal{X}.P$, respectively. To define the set of free names of a process P, noted $fn(P)$, since the set of variables $\mathcal{V}$ is separate from the set of names Λ, we need only to collect the names from Λ which are not under the scope of a name restriction. The set of restricted names is the object collected by the set of bound names of a process P, noted $bn(P)$. The set of free labels of a process P, noted $fl(P)$, collects all labels that occur in the process.

Definition 4.2.2 (Free Names)
The set of free names of a process P, noted $fn(P)$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{llllll}
fn(\mathbf{0}) & \triangleq & \emptyset & fn(n \blacktriangleleft [P]) & \triangleq & (\{n\} \cap \Lambda) \cup fn(P) \\
fn(P \mid Q) & \triangleq & fn(P) \cup fn(Q) & fn(l^d!(\tilde{n}).P) & \triangleq & (\tilde{n} \cap \Lambda) \cup fn(P) \\
fn(\mathcal{X}) & \triangleq & \emptyset & fn(l^d?(\tilde{x}).P) & \triangleq & fn(P) \\
fn(\textsf{rec}\ \mathcal{X}.P) & \triangleq & fn(P) & fn(\textsf{this}(x).P) & \triangleq & fn(P) \\
fn((\boldsymbol{\nu} a)P) & \triangleq & fn(P) \setminus \{a\} & fn(\Sigma_{i \in I}\ \alpha_i.P_i) & \triangleq & \bigcup_{i \in I} fn(\alpha_i.P_i)
\end{array}
$$

Definition 4.2.3 (Bound Names)
The set of bound names of a process P, noted $bn(P)$, is inductively defined on the structure of processes as follows:

$$\begin{array}{lllllll} bn(\mathbf{0}) & \triangleq & \emptyset & \quad & bn(n \blacktriangleleft [P]) & \triangleq & bn(P) \\ bn(P \mid Q) & \triangleq & bn(P) \cup bn(Q) & & bn(l^d!(\tilde{n}).P) & \triangleq & bn(P) \\ bn(\mathcal{X}) & \triangleq & \emptyset & & bn(l^d?(\tilde{x}).P) & \triangleq & bn(P) \\ bn(\textsf{rec}\ \mathcal{X}.P) & \triangleq & bn(P) & & bn(\textsf{this}(x).P) & \triangleq & bn(P) \\ bn((\boldsymbol{\nu} a)P) & \triangleq & bn(P) \cup \{a\} & & bn(\Sigma_{i \in I}\ \alpha_i.P_i) & \triangleq & \bigcup_{i \in I} bn(\alpha_i.P_i) \end{array}$$

Definition 4.2.4 (Free Labels)
The set of free labels of a process P, noted $fl(P)$, is inductively defined on the structure of processes as follows:

$$\begin{array}{lllllll} fl(\mathbf{0}) & \triangleq & \emptyset & \quad & fl(n \blacktriangleleft [P]) & \triangleq & fl(P) \\ fl(P \mid Q) & \triangleq & fl(P) \cup fl(Q) & & fl(l^d!(\tilde{n}).P) & \triangleq & \{l\} \cup fl(P) \\ fl(\mathcal{X}) & \triangleq & \emptyset & & lb(l^d?(\tilde{x}).P) & \triangleq & \{l\} \cup fl(P) \\ fl(\textsf{rec}\ \mathcal{X}.P) & \triangleq & fl(P) & & fl(\textsf{this}(x).P) & \triangleq & fl(P) \\ fl((\boldsymbol{\nu} a)P) & \triangleq & fl(P) & & fl(\Sigma_{i \in I}\ \alpha_i.P_i) & \triangleq & \bigcup_{i \in I} fl(\alpha_i.P_i) \end{array}$$

Next we define the set free variables of a process P, noted $fv(P)$, hence the set of variable identifiers that occur not within the scope of an input prefix or a **this**. Also we define the set of free recursion variables of a process P, noted $frv(P)$, hence the set of recursion variables that occur not within the scope of a binding recursive process construct.

Definition 4.2.5 (Free Variables)
The set of free variables of a process P, noted $fv(P)$, is inductively defined on the structure of processes as follows:

$$\begin{array}{lllllll} fv(\mathbf{0}) & \triangleq & \emptyset & \quad & fv(n \blacktriangleleft [P]) & \triangleq & (\{n\} \cap \mathcal{V}) \cup fv(P) \\ fv(P \mid Q) & \triangleq & fv(P) \cup fv(Q) & & fv(l^d!(\tilde{n}).P) & \triangleq & (\tilde{n} \cap \mathcal{V}) \cup fv(P) \\ fv(\mathcal{X}) & \triangleq & \emptyset & & fv(l^d?(\tilde{x}).P) & \triangleq & fv(P) \setminus \tilde{x} \\ fv(\textsf{rec}\ \mathcal{X}.P) & \triangleq & fv(P) & & fv(\textsf{this}(x).P) & \triangleq & fv(P) \setminus \{x\} \\ fv((\boldsymbol{\nu} a)P) & \triangleq & fv(P) & & fv(\Sigma_{i \in I}\ \alpha_i.P_i) & \triangleq & \bigcup_{i \in I} fv(\alpha_i.P_i) \end{array}$$

Definition 4.2.6 (Free Recursion Variables)
The set of free process variables of a process P, noted $frv(P)$, is inductively defined on the structure of processes as follows:

$$\begin{array}{lllllll} frv(\mathbf{0}) & \triangleq & \emptyset & \quad & frv(n \blacktriangleleft [P]) & \triangleq & frv(P) \\ frv(P \mid Q) & \triangleq & frv(P) \cup frv(Q) & & frv(l^d!(\tilde{n}).P) & \triangleq & frv(P) \\ frv(\mathcal{X}) & \triangleq & \{\mathcal{X}\} & & frv(l^d?(\tilde{x}).P) & \triangleq & frv(P) \\ frv(\textsf{rec}\ \mathcal{X}.P) & \triangleq & frv(P) \setminus \{\mathcal{X}\} & & frv(\textsf{this}(x).P) & \triangleq & frv(P) \\ frv((\boldsymbol{\nu} a)P) & \triangleq & frv(P) & & frv(\Sigma_{i \in I}\ \alpha_i.P_i) & \triangleq & \bigcup_{i \in I} frv(\alpha_i.P_i) \end{array}$$

We say a process is closed if it has no free variables. Since we usually are interested only in

closed processes we often refer to closed processes simply as processes.

Definition 4.2.7 (Closed Process)
We say P is a closed process if $fv(P) = \emptyset$ and $frv(P) = \emptyset$.

Convention 4.2.8 *We use the term process to refer to closed processes, where appropriate.*

We define application of substitution to a process, noted $P\{n_1, \ldots, n_k/m_1, \ldots, m_k\}$, which replaces all free occurrences of the n_i identifiers by the respective m_i identifiers. In the definition we use $\#$ to indicate set disjointness and $\tilde{n}, \tilde{m}$ to represent $\tilde{n} \cup \tilde{m}$.

Definition 4.2.9 (Substitution)
Given identifiers $n_1, \ldots, n_k$ and $m_1, \ldots, m_k$, the application of substitution to a process, noted $P\{n_1, \ldots, n_k/m_1, \ldots, m_k\}$ or $P\{\tilde{n}/\tilde{m}\}$, is inductively defined on the structure of processes as:

$$
\begin{array}{lcll}
\mathbf{0}\{\tilde{n}/\tilde{m}\} & \triangleq & \mathbf{0} & \\
(P \mid Q)\{\tilde{n}/\tilde{m}\} & \triangleq & P\{\tilde{n}/\tilde{m}\} \mid Q\{\tilde{n}/\tilde{m}\} & \\
\mathcal{X}\{\tilde{n}/\tilde{m}\} & \triangleq & \mathcal{X} & \\
\textsf{rec}\, \mathcal{X}.P\{\tilde{n}/\tilde{m}\} & \triangleq & \textsf{rec}\, \mathcal{X}.(P\{\tilde{n}/\tilde{m}\}) & \\
((\boldsymbol{\nu} a)P)\{\tilde{n}/\tilde{m}\} & \triangleq & (\boldsymbol{\nu} a)(P\{\tilde{n}/\tilde{m}\}) & (\textit{if}\ a \notin \tilde{n}, \tilde{m}) \\
n_i \blacktriangleleft [P]\{\tilde{n}/\tilde{m}\} & \triangleq & m_i \blacktriangleleft [P\{\tilde{n}/\tilde{m}\}] & \\
u \blacktriangleleft [P]\{\tilde{n}/\tilde{m}\} & \triangleq & u \blacktriangleleft [P\{\tilde{n}/\tilde{m}\}] & (\textit{if}\ u \notin \tilde{n}) \\
(l^d!(\tilde{o}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & l^d!(\tilde{v}).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if}\ o_j = n_i\ \textit{then}\ v_j = m_i\ \textit{else}\ v_j = o_j) \\
(l^d?(\tilde{x}).P)\{\tilde{n}/\tilde{m}\} & \triangleq & l^d?(\tilde{x}).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if}\ \tilde{x} \# \tilde{n}, \tilde{m}) \\
(\textsf{this}(x).P)\{\tilde{n}/\tilde{m}\} & \triangleq & \textsf{this}(x).(P\{\tilde{n}/\tilde{m}\}) & (\textit{if}\ x \notin \tilde{n}, \tilde{m}) \\
(\Sigma_{i \in I}\ \alpha_i.P_i)\{\tilde{n}/\tilde{m}\} & \triangleq & \Sigma_{i \in I}\ (\alpha_i.P_i\{\tilde{n}/\tilde{m}\}) &
\end{array}
$$

N.B. By i we denote an index such that $i \in 1, \ldots, k$.

We introduce *α-equivalence* which identifies processes which are equivalent up to a (safe) renaming of bound names and variables.

Definition 4.2.10 (α-Equivalence)
α-equivalence, noted $\equiv_\alpha$, is the least congruence on processes that satisfies the following rules:

$$
\begin{array}{rcll}
(\boldsymbol{\nu} a)P & \equiv_\alpha & (\boldsymbol{\nu} b)(P\{a/b\}) \ \ (b \notin fn(P)) & (\textit{Restriction}) \\
l^d?(\tilde{x}).P & \equiv_\alpha & l^d?(\tilde{y}).(P\{\tilde{x}/\tilde{y}\}) \ \ (\tilde{y} \# fv(P)) & (\textit{Input}) \\
\textsf{this}(x).P & \equiv_\alpha & \textsf{this}(y).(P\{x/y\}) \ \ (y \notin fv(P)) & (\textit{This})
\end{array}
$$

Processes which are α-equivalent essentially represent the same specification, as they differ only in the particular identities of some bound names that can freely be renamed without changing the meaning of the process. We thus implicitly identify them so as to simplify presentation.

Convention 4.2.11 *We implicitly identify α-equivalent processes.*

Recursive processes repeat their behavior at the point where the recursion variable occurs, which we represent by unfolding the recursive process definition, which amounts to say we substitute the recursion variable by the whole recursive definition. To that end, we define the application of substitution of a recursion variable by a process.

Definition 4.2.12 (Recursion Variable Substitution)
Given recursion variable $\mathcal{X}$ and process P the application of a recursion variable substitution to a process in Q, noted $Q\{\mathcal{X}/P\}$, is inductively defined on the structure of processes as follows:

$$
\begin{array}{llllll}
(Q \mid R)\{\mathcal{X}/P\} & \triangleq & Q\{\mathcal{X}/P\} \mid R\{\mathcal{X}/P\} & \mathbf{0}\{\mathcal{X}/P\} & \triangleq & \mathbf{0} \\
\mathcal{X}\{\mathcal{X}/P\} & \triangleq & P & \mathcal{Y}\{\mathcal{X}/P\} & \triangleq & \mathcal{Y} \\
(\mathsf{rec}\,\mathcal{X}.Q)\{\mathcal{X}/P\} & \triangleq & \mathsf{rec}\,\mathcal{X}.Q & (\mathsf{rec}\,\mathcal{Y}.Q)\{\mathcal{X}/P\} & \triangleq & \mathsf{rec}\,\mathcal{Y}.(Q\{\mathcal{X}/P\}) \\
n \blacktriangleleft [Q]\{\mathcal{X}/P\} & \triangleq & n \blacktriangleleft [Q\{\mathcal{X}/P\}] & ((\boldsymbol{\nu} a)Q)\{\mathcal{X}/P\} & \triangleq & (\boldsymbol{\nu} a)(Q\{\mathcal{X}/P\}) \\
(l^d!(\tilde{n}).Q)\{\mathcal{X}/P\} & \triangleq & l^d!(\tilde{n}).(Q\{\mathcal{X}/P\}) & (l^d?(\tilde{x}).Q)\{\mathcal{X}/P\} & \triangleq & l^d?(\tilde{x}).(Q\{\mathcal{X}/P\}) \\
(\mathsf{this}(x).Q)\{\mathcal{X}/P\} & \triangleq & \mathsf{this}(x).(Q\{\mathcal{X}/P\}) & (\Sigma_{i\in I}\ \alpha_i.Q_i)\{\mathcal{X}/P\} & \triangleq & \Sigma_{i\in I}\ (\alpha_i.Q_i\{\mathcal{X}/P\})
\end{array}
$$

Before presenting the operational semantics of the core CC model, which provides a means to precisely characterize how CC processes evolve, we informally describe the language constructs.

Static Fragment

Inaction $\mathbf{0}$, parallel composition $P \mid Q$, restriction $(\boldsymbol{\nu} a)P$ and recursion $\mathsf{rec}\,\mathcal{X}.P$ are constructs commonly found in process calculi based on the π-Calculus. Inaction represents the process that has no behavior. Parallel composition represents two processes that are simultaneously active, so the behavior of the parallel composition can be the result of behavior of either one of the branches individually, but also from synchronizations between the two branches. Restriction allows for the creation of new unique names that are local to a part of a system, but where the scope of the name can dynamically grow since the name can be communicated in messages. Recursion allows for representing infinite behavior, as $\mathsf{rec}\,\mathcal{X}.P$ behaves as $P\{\mathcal{X}/\mathsf{rec}\,\mathcal{X}.P\}$.

Communication

Communication between subsystems is realized by means of message passing. In particular, we denote the input and the output of messages from/to the current context by the constructs:

$$\mathtt{messageLabel}^{\downarrow}?(\tilde{x}).P \qquad\qquad \mathtt{messageLabel}^{\downarrow}!(\tilde{n}).P$$

In the output case, the terms n_i represent message arguments, values to be sent, as expected. In the input case, the variables x_i represent message parameters that are bound in P, as expected. The target symbol $\downarrow$ (read "here") says that the corresponding communication actions must interact in the current context. Since messages to the current conversation ($\downarrow$) are the most frequently used, we sometimes omit the $\downarrow$ symbol to simplify presentation. Second, we denote the input and the output of messages from/to the outer context by the constructs:

$$\mathtt{messageLabel}^{\uparrow}?(\tilde{x}).P \qquad\qquad \mathtt{messageLabel}^{\uparrow}!(\tilde{n}).P$$

The target symbol $\uparrow$ (read "up") says that the corresponding communication actions must interact in the (uniquely determined) outer context, where "outer" is understood relatively to the context where the $^{\uparrow}?$ or $^{\uparrow}!$ process is running.

For example, consider the following process:

$$\texttt{info}^{\downarrow}?(\mathit{data}).\texttt{fwdInfo}^{\uparrow}!(\mathit{data})$$

which can be described as a forwarding process, since it is waiting to receive on message `info` some data in the current ($\downarrow$) conversation, and upon reception forwards the received data in message `fwdInfo` to the enclosing ($\uparrow$) conversation. If we are to place this process in a context where a process is willing to send message `info` in the current conversation (e.g., $\texttt{info}^{\downarrow}!(\mathit{myData})$), then the `info` message can be exchanged, as captured by the following reduction:

$$\texttt{info}^{\downarrow}?(\mathit{data}).\texttt{fwdInfo}^{\uparrow}!(\mathit{data}) \mid \texttt{info}^{\downarrow}!(\mathit{myData}) \rightarrow \texttt{fwdInfo}^{\uparrow}!(\mathit{myData})$$

which describes such system evolves to a process that is willing to emit message `fwdInfo` to the enclosing conversation, carrying the *myData* exchanged in the `info` message. To determine the enclosing conversation we need to look at the enclosing environment where the processes specifying the message exchanges are placed, namely to enclosing conversation access constructs.

Conversation Access

The conversation access construct $n \blacktriangleleft [P]$ allows a process to initiate interactions, as specified by P, in conversation n. Conversation contexts are a medium for related interactions, and processes may access such contexts by specifying a conversation access indicating the name of the conversation to be accessed (e.g., n in $n \blacktriangleleft [P]$). Consider the following specification of a process willing to send a message `chat` in conversation *chatRoom*:

$$\mathit{chatRoom} \blacktriangleleft [\, \texttt{chat}!(\texttt{hi}) \,]$$

where we omit the $\downarrow$ direction in the message. On the other hand, processes may also interact in the enclosing conversation by means of $\uparrow$ directed messages, which is determined as the conversation enclosing their current context. For instance:

$$\mathit{chatRoom} \blacktriangleleft \big[\, \mathit{privateChat} \blacktriangleleft \big[\, \texttt{chat}^{\uparrow}!(\texttt{hi}) \,\big] \,\big]$$

specifies a process that is willing to send message `chat` in conversation *chatRoom* as before, but where now the process is actually running inside a nested conversation *privateChat* and is interacting in the enclosing conversation via a $\uparrow$ directed message.

Processes can access conversations from different points of a system, so a conversation context can actually be distributed in many access pieces, and processes inside any piece can seamlessly interact with processes located in any other piece of the same conversation. For example, consider the following processes P and Q, defined as:

$$P \triangleq \mathit{chatRoom} \blacktriangleleft [\, \texttt{chat}!(b) \mid \texttt{chat}?(x) \,]$$
$$Q \triangleq \mathit{chatRoom} \blacktriangleleft [\, \texttt{chat}!(b) \,] \mid \mathit{chatRoom} \blacktriangleleft [\, \texttt{chat}?(x) \,]$$

where the chat message exchange can take place in both the configurations given by P and by Q. Each conversation piece will potentially be placed at a different enclosing context. On the other hand, any such conversation piece will necessarily be placed at a single enclosing context. The relationship between the enclosing context and a nested conversation piece may be seen as a call/callee relationship, but where both entities may interact continuously through ↑ messages. For instance, the context where a service instantiation **new** is located in is the caller context which performs the call to the service, while the conversation in which the service interaction takes place is the callee, and processes located in the latter can interact in the former via ↑ messages. Consider process R:

$$R \triangleq \mathit{chatRoom} \blacktriangleleft [\, \texttt{chat}?(x) \mid \mathit{privateChat} \blacktriangleleft [\, \texttt{chat}^{\uparrow}!(b) \,]\,]$$

where the chat message exchange takes place in conversation *chatRoom*, similarly to the message exchange in process P, but where the process emitting the message is actually located in a nested conversation (*privateChat*), and the output prefix is specified with the ↑ direction. Each context is uniquely identified by a name (cf., a URI). Therefore, access pieces specifying different names will access distinct communication mediums, so in the following process:

$$R \triangleq \mathit{chatRoomA} \blacktriangleleft [\, \texttt{chat}!(b) \,] \mid \mathit{chatRoomB} \blacktriangleleft [\, \texttt{chat}?(x) \,]$$

the *chat* message exchange cannot take place since messages pertain to different conversations ($\mathit{chatRoomA} \neq \mathit{chatRoomB}$). Sometimes it is useful for a process to dynamically gain access to the identify of the conversation in which it is interacting, a capability supported by the context awareness primitive described next.

Conversation Awareness

A process running inside a given conversation context is able to dynamically access the identity of the conversation. This capability may be realized by the construct:

$$\textbf{this}(x).P$$

The variable x will be replaced inside the process P by the name n of the current context. The computation will proceed as $P\{x/n\}$. This primitive bears some similarity with the **self** or **this** of object-oriented languages, even if it has a different semantics. For instance, consider the process:

$$\mathit{chatRoom} \blacktriangleleft [\, \textbf{this}(x).P \,]$$

which specifies a process that is willing to access the identity of the conversation in which it is running in, the *chatRoom* conversation. This system can evolve to:

$$\mathit{chatRoom} \blacktriangleleft [\, P\{x/\mathit{chatRoom}\} \,]$$

where the variable x is replaced by the conversation name *chatRoom*. This primitive will reveal its most important use when we program the service oriented idioms, namely the joining of a partner to the current conversation, allowing to avoid explicit reference to conversation names.

Given this intuitive presentation of the primitives of the language, we now turn to providing them with a rigorously defined operational semantics, so as to allow us to precisely characterize how core CC processes evolve through interaction.

4.3 Operational Semantics

The operational semantics of the core CC is defined by a labeled transition system. A transition $P \xrightarrow{\lambda} Q$ states that process P may evolve to process Q by performing the action represented by the transition label λ. Our transition labels characterize both the autonomous evolutions of processes — the internal actions — and also the interactions a process has with the external environment. To simplify the presentation of the definition of transition labels, we introduce an auxiliary notion of *actions* which characterize the basic observations that can be made over processes. We define transition labels, actions, and extended directions.

Definition 4.3.1 (Extended Directions, Actions and Transition Labels)
Extended directions, actions and transition labels are defined as follows:

$$
\begin{array}{lcll}
d_e & ::= & d \mid \downarrow\uparrow & \text{(Extended Directions)} \\
\sigma & ::= & \tau \;\mid\; l^d!(\tilde{a}) \;\mid\; l^d?(\tilde{a}) \;\mid\; \mathtt{this}^{d_e} & \text{(Actions)} \\
\lambda & ::= & c\ \sigma \;\mid\; \sigma \;\mid\; (\nu a)\lambda & \text{(Transition Labels)}
\end{array}
$$

We denote by $\mathcal{T}$ the set of all transition labels.

An action τ denotes an internal communication, actions $l^d!(\tilde{a})$ and $l^d?(\tilde{a})$ represent communications with the environment, and action $\mathtt{this}^{d_e}$ represents a conversation identity access. Output and input actions have a direction d as processes can specify that a message is to be exchanged in either the current $\downarrow$ or the enclosing $\uparrow$ conversation. Although processes can only explicitly access the identity of the current conversation (captured by a $\mathtt{this}^{\downarrow}$ action), we also use the $\mathtt{this}$ action to represent other behaviors that are contextually dependent — we identify such behaviors in Definition 4.3.8. Then, depending on the direction d_e ($\downarrow$, $\uparrow$ or $\downarrow\uparrow$), the $\mathtt{this}^{d_e}$ action specifies a different conversation which is to be accessed: a $\mathtt{this}^{\downarrow}$ label accesses the identity of the current conversation, a $\mathtt{this}^{\uparrow}$ label accesses the identity of the enclosing conversation, and a $\mathtt{this}^{\downarrow\uparrow}$ label checks if the identity of the enclosing and of the current conversations is the same.

To capture the communication capabilities of processes, transition labels need to register not only the action but also the conversation where the action takes place. So, a transition label λ containing $c\ \sigma$ is said to be *located at* conversation c (or just *located*), otherwise is said to be *unlocated*. In $(\nu a)\lambda$ the distinguished occurrence of a is bound with scope λ (cf., the π-Calculus bound output actions). We use $fn(\lambda)$ and $bn(\lambda)$ to denote (respectively) the free and bound names of a transition label, and $na(\lambda)$ to denote both free and bound names of a transition label. We introduce the free names of an action $fn_A(\sigma)$, and then define $fn(\lambda)$, $bn(\lambda)$ and $na(\lambda)$.

Definition 4.3.2 (Action Free Names)
The set of free names of an action σ, noted $fn_A(\sigma)$, is defined as follows:

$$
fn_A(\tau) \triangleq \emptyset \qquad fn_A(l^d!(\tilde{a})) \triangleq \tilde{a} \qquad fn_A(l^d?(\tilde{a})) \triangleq \tilde{a} \qquad fn_A(\mathtt{this}^{d_e}) \triangleq \emptyset
$$

Definition 4.3.3 (Transition Free Names)
The set of free names of a transition λ*, noted* $fn(\lambda)$*, is defined as follows:*

$$fn(c\ \sigma) \triangleq c \cup fn_A(\sigma) \qquad fn(\sigma) \triangleq fn_A(\sigma) \qquad fn((\nu a)\lambda) \triangleq fn(\lambda) \setminus \{a\}$$

Definition 4.3.4 (Transition Bound Names)
The set of bound names of a transition λ*, noted* $bn(\lambda)$*, is defined as follows:*

$$bn(c\ \sigma) \triangleq \emptyset \qquad bn(\sigma) \triangleq \emptyset \qquad bn((\nu a)\lambda) \triangleq \{a\} \cup bn(\lambda)$$

Definition 4.3.5 (Transition Names)
The set of names of a transition λ*, noted* $na(\lambda)$*, is defined as follows:*

$$na(\lambda) \triangleq fn(\lambda) \cup bn(\lambda)$$

We usually abbreviate the set of bound names in a transition as follows.

Notation 4.3.6 *We abbreviate* $(\nu a_1) \dots (\nu a_k)\lambda$ *with* $(\nu \tilde{a})\lambda$*.*

In order to define the labeled transition system, we introduce a synchronization algebra (introduced by Winskel [96]) that describes how two parallel processes may synchronize. The synchronization algebra specifies how any pair of transition labels may be combined, and what is the label obtained from that combination. Since not all combinations represent accepted synchronizations we use $\circ$ to represent undefined (in such way avoiding the introduction of a partial function). We motivate the definition of our synchronization algebra with an example.

Example 4.3.7 *Consider the following process:*

$$l^{\downarrow}!().P \mid c \blacktriangleleft \left[l^{\downarrow}?().Q\right] \tag{4.3.7.1}$$

which specifies that a message l *is to be sent at the current conversation, after which* P *is activated, and a message* l *is to be received at conversation* c*, after which* Q *is activated. If such a process is to be placed in a conversation* c *piece, i.e.:*

$$c \blacktriangleleft \left[l^{\downarrow}!().P \mid c \blacktriangleleft \left[l^{\downarrow}?().Q\right]\right]$$

then both input and output refer to the same conversation c *and therefore the message may be exchanged in conversation* c*. We then have:*

$$c \blacktriangleleft \left[l^{\downarrow}!().P \mid c \blacktriangleleft \left[l^{\downarrow}?().Q\right]\right] \xrightarrow{\tau} c \blacktriangleleft [P \mid c \blacktriangleleft [Q]]$$

When locally describing the behavior of the process shown in (4.3.7.1) we must then account for a possible synchronization ***if*** *the current conversation is the* c *conversation. This can be captured by a* c $\texttt{this}^{\downarrow}$ *transition which may only progress in a* c *conversation piece:*

$$l^{\downarrow}!().P \mid c \blacktriangleleft \left[l^{\downarrow}?().Q\right] \xrightarrow{c\ \texttt{this}^{\downarrow}} P \mid c \blacktriangleleft [Q]$$

Since messages do not a priori explicitly refer the conversation to which they pertain, the operational semantics of the core CC must locally account for synchronizations which may arise depending on the surrounding context. The synchronization algebra realizes this notion by specifying `this` labels as the result of combining transition labels that may synchronize depending on the surrounding conversations. If two labels directly synchronize, i.e., regardless of the enclosing environment, then the resulting label is τ, as usual. We define the synchronization algebra.

Definition 4.3.8 (Core CC Synchronization Algebra)
The synchronization algebra for core CC processes is defined as a triple $(\mathcal{T} \cup \circ, \circ, \bullet)$ *where* $\bullet$ *is a function on pairs of* $\mathcal{T} \cup \circ$ *defined by the following table:*

$\bullet$	τ	$l^{\downarrow}!(\tilde{a})$	$l^{\downarrow}?(\tilde{a})$	$l^{\uparrow}!(\tilde{a})$	$l^{\uparrow}?(\tilde{a})$	$c\ l^{\downarrow}!(\tilde{a})$	$c\ l^{\downarrow}?(\tilde{a})$	$\ldots$	$c\ \mathtt{this}^{\downarrow}$	$\ldots$	$\circ$
τ	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$
$l^{\downarrow}!(\tilde{a})$	$\circ$	$\circ$	τ	$\circ$	$\mathtt{this}^{\downarrow\uparrow}$	$\circ$	$c\ \mathtt{this}^{\downarrow}$	$\ldots$	$\circ$	$\ldots$	$\circ$
$l^{\downarrow}?(\tilde{a})$	$\circ$	τ	$\circ$	$\mathtt{this}^{\downarrow\uparrow}$	$\circ$	$c\ \mathtt{this}^{\downarrow}$	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$
$l^{\uparrow}!(\tilde{a})$	$\circ$	$\circ$	$\mathtt{this}^{\downarrow\uparrow}$	$\circ$	τ	$\circ$	$c\ \mathtt{this}^{\uparrow}$	$\ldots$	$\circ$	$\ldots$	$\circ$
$l^{\uparrow}?(\tilde{a})$	$\circ$	$\mathtt{this}^{\downarrow\uparrow}$	$\circ$	τ	$\circ$	$c\ \mathtt{this}^{\uparrow}$	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$
$c\ l^{\downarrow}!(\tilde{a})$	$\circ$	$\circ$	$c\ \mathtt{this}^{\downarrow}$	$\circ$	$c\ \mathtt{this}^{\uparrow}$	$\circ$	τ	$\ldots$	$\circ$	$\ldots$	$\circ$
$c\ l^{\downarrow}?(\tilde{a})$	$\circ$	$c\ \mathtt{this}^{\downarrow}$	$\circ$	$c\ \mathtt{this}^{\uparrow}$	$\circ$	τ	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\circ$
$c\ \mathtt{this}^{\downarrow}$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\circ$
$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\circ$	$\ldots$	$\circ$	$\ldots$	$\circ$

We then have that a located label $c\ l^{\downarrow}?(\tilde{a})$ and an unlocated label $l^{\downarrow}!(\tilde{a})$ may synchronize, provided the interaction is to take place in conversation c, i.e., provided the current conversation is c, which is captured by the $c\ \mathtt{this}^{\downarrow}$ label. Likewise we have that $c\ l^{\downarrow}?(\tilde{a})$ and $l^{\uparrow}!(\tilde{a})$ can synchronize provided the enclosing conversation is c, captured by $c\ \mathtt{this}^{\uparrow}$. Also, we have that $l^{\uparrow}?(\tilde{a})$ and $l^{\downarrow}!(\tilde{a})$ may synchronize, provided that the current and enclosing conversations have the same identity, captured by the $\mathtt{this}^{\downarrow\uparrow}$ label.

Our transition system rules rely on some operations over transition labels and some auxiliary notation, which we now introduce. We define an operator that collects the set of names emitted in an output transition, used to determine extrusion of restricted names.

Definition 4.3.9 (Emitted Names)
We denote by $out(\lambda)$ *the set of names emitted in an output label, defined inductively as follows:*

$$out(\tau) \triangleq \emptyset \qquad out(l^{d}!(\tilde{a})) \triangleq \tilde{a} \qquad out(l^{d}?(\tilde{a})) \triangleq \emptyset \qquad out(\mathtt{this}^{d_e}) \triangleq \emptyset$$

$$out(c\ \sigma) \triangleq out(\sigma) \setminus \{c\} \qquad out((\nu a)\lambda) \triangleq out(\lambda) \setminus \{a\}$$

To simplify presentation of the transition rules that require some update in the labels, we introduce an auxiliary notation used to represent the change of the direction of a transition label, used in the rules that characterize how transitions cross conversation boundaries.

Notation 4.3.10 *We note by λ^d a transition label λ^d defined in direction d $(\uparrow, \downarrow)$, and by $\lambda^{d'}$ the label obtained by replacing d by d' in λ^d.*

So, for example, if $\lambda^\uparrow$ is $\texttt{askPrice}^\uparrow?(a)$ then $\lambda^\downarrow$ is $\texttt{askPrice}^\downarrow?(a)$. Next we define an operation used to locate a transition in a determined conversation.

Definition 4.3.11 (Locating a Transition)
Given an unlocated label λ, we represent by $c \cdot \lambda$ the label obtained by locating λ at c defined inductively as follows:

$$c \cdot \tau \triangleq \tau \qquad c \cdot l^d!(\tilde{a}) \triangleq c\ l^d!(\tilde{a}) \qquad c \cdot l^d?(\tilde{a}) \triangleq c\ l^d?(\tilde{a})$$
$$c \cdot \texttt{this}^{d_e} \triangleq \texttt{this}^{d_e} \qquad c \cdot (\nu a)\lambda \triangleq (\nu a)c \cdot \lambda \ \ (\textit{if } a \neq c)$$

For instance, we have that if $\lambda^\downarrow$ is $\texttt{askPrice}^\downarrow?(p)$ then $c \cdot \lambda^\downarrow$ is $c\ \texttt{askPrice}^\downarrow?(p)$. The $c\cdot$ operator will be used to locate output and input transitions which are not already located. Notice that we do not locate τ and `this` labels, since τ transitions are not subject to being located, while `this` transitions require specific handling. To distinguish the sort of labels we intend to capture in the rules we introduce two predicates: one that identifies output and input labels which are already located — the *loc* predicate — and other that identifies output and input labels which are not located and defined in the $\downarrow$ direction — the *unloc* predicate. While *loc* labels are not subject to change when crossing a conversation syntactic barrier, *unloc* labels must be located in a conversation when crossing the corresponding conversation syntactic barrier.

Definition 4.3.12 (Located Transitions Predicate)
We say λ is a located output or input transition label, noted loc(λ), if there is $c, \tilde{a}, \tilde{b}, l$ such that either $\lambda = (\nu\tilde{a})c\ l^\downarrow!(\tilde{b})$ or $\lambda = c\ l^\downarrow?(\tilde{b})$.

Definition 4.3.13 (Unlocated Transitions Predicate)
We say λ is an unlocated output or input transition label, noted unloc(λ), if there is $\tilde{a}, \tilde{b}, l$ such that either $\lambda = (\nu\tilde{a})l^\downarrow!(\tilde{b})$ or $\lambda = l^\downarrow?(\tilde{b})$.

We may now present the transition relations. We start by describing informally the key rules and then present the definition. This first set of rules is standard, in the sense that the same rules are found in the π-Calculus labeled transition system (see [85]).

$$l^d!(\tilde{a}).P \xrightarrow{l^d!(\tilde{a})} P\ (\textit{Out}) \qquad l^d?(\tilde{x}).P \xrightarrow{l^d?(\tilde{a})} P\{\tilde{x}/\tilde{a}\}\ (\textit{In}) \qquad \frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad j \in I}{\Sigma_{i \in I}\ \alpha_i.P_i \xrightarrow{\lambda} Q}(\textit{Sum})$$

In rule (*Out*) an output is observed over the output-prefixed process, representing a message emission to the external environment, being the arrival state of the transition the continuation of the output-prefixed process. The symmetric rule (*In*) specifies the input observation over the input-prefixed process, representing a message reception from the external environment. The arrival state of the transition is the continuation of the input prefixed-process where the variables $\tilde{x}$ are replaced by the received names $\tilde{a}$. Rule (*Sum*) describes the behavior of the prefixed guarded choice, that can choose between any of its initial actions and evolve to the

respective continuation. The following two rules for name restriction are also standard.

$$\frac{P \xrightarrow{\lambda} Q \quad a \in out(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{(\nu a)\lambda} Q}(Open) \qquad \frac{P \xrightarrow{\lambda} Q \quad a \notin na(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{\lambda} (\boldsymbol{\nu} a)Q}(Res)$$

While rule (*Res*) describes that transitions that do not mention the restricted name transparently cross the restriction boundary, rule (*Open*) describes the case when the transition is an output action that is emitting the restricted name. In such case, the restriction scope is opened, since it will be enlarged so as to comprehend the process that will receive the name. The following rules are also standard.

$$\frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R)}{P \mid R \xrightarrow{\lambda} Q \mid R}(Par) \qquad \frac{P\{\mathcal{X}/\textbf{rec}\, \mathcal{X}.P\} \xrightarrow{\lambda} Q}{\textbf{rec}\, \mathcal{X}.P \xrightarrow{\lambda} Q}(Rec)$$

Rule (*Par*) describes that a behavior observed over one of the parallel branches is also observed at the level of the parallel composition, since both branches are simultaneously active. The side condition guarantees there is no unintended name capture. Rule (*Rec*) says the recursive process behaves as its one step unfolding. The next rule captures the synchronization of parallel processes, relying on the synchronization algebra that refines the notion of duality.

$$\frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} P' \mid Q'}(Comm)$$

Thus two processes may synchronize if they exhibit transitions which can be explained by the synchronization operation $\bullet$. If the transitions are dual then $\lambda_1 \bullet \lambda_2$ is τ, otherwise $\lambda_1 \bullet \lambda_2$ is defined over a `this` label that will then read the necessary contextual information so as to determine if the labels are dual with respect to such context. Rule (*Close*) extends rules (*Comm*) as usual, considering the output is carrying some bound names which scope is now to be closed.

$$\frac{P \xrightarrow{(\nu\tilde{a})\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \tilde{a} \# fn(Q) \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')}(Close)$$

The scope of the restricted names is thus enlarged so to contain both emitter (who already knew the restricted names) and receiver (who gains knowledge of the restricted names) and hence the scope is *close*d, provided the bound name identifiers are fresh to Q. We now turn to the rules specific to our setting that explain the behavior of conversation access pieces and of the conversation awareness primitive.

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q \quad (c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda^{\downarrow}} c \blacktriangleleft [Q]}(Here)$$

Rule (*Here*) describes that an action directed to the enclosing conversation becomes an action directed to the current conversation when crossing a conversation boundary, captured by the update of the direction. Instead, when the action pertains to the current conversation we have:

$$\frac{P \xrightarrow{\lambda} Q \quad (unloc(\lambda))}{c \blacktriangleleft [P] \xrightarrow{c \cdot \lambda} c \blacktriangleleft [Q]}(Loc)$$

which specifies that a not yet located $\downarrow$ action gets located in the conversation in which it originates. If an action is already located in some conversation then it transparently crosses the conversation boundary, as specified in the following rule:

$$\frac{P \xrightarrow{\lambda} Q \quad (loc(\lambda), c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda} c \blacktriangleleft [Q]} (\mathit{Through})$$

Likewise an internal action transparently crosses a conversation boundary, hence:

$$\frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (\mathit{Tau})$$

The conversation awareness primitive reads the identity of the current conversation by means of a $c\ \mathtt{this}^{\downarrow}$ transition, as specified in the following rule:

$$\mathbf{this}(x).P \xrightarrow{c\ \mathtt{this}^{\downarrow}} P\{x/c\}\ (\mathit{This})$$

where the variable x is replaced by the name read in the label c. Such transition may only progress if the immediate conversation boundary it finds is in fact, a c conversation access, as expressed in the following rule:

$$\frac{P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (\mathit{ThisLoc})$$

Rule ($\mathit{ThisLoc}$) also captures the case of a $c\ \mathtt{this}^{\downarrow}$ that originates from a synchronization ($\bullet$) of two labels, while $c\ \mathtt{this}^{\uparrow}$ labels are captured in rule (Here) previously described, hence:

$$\frac{P \xrightarrow{c\ \mathtt{this}^{\uparrow}} Q}{c \blacktriangleleft [P] \xrightarrow{c\ \mathtt{this}^{\downarrow}} c \blacktriangleleft [Q]}$$

The last case of transitions that are the result of a $\bullet$ is the $\mathtt{this}^{\downarrow\uparrow}$ label, captured in the rule:

$$\frac{P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q}{c \blacktriangleleft [P] \xrightarrow{c\ \mathtt{this}^{\downarrow}} c \blacktriangleleft [Q]} (\mathit{ThisHere})$$

which reads the enclosing conversation through a $c\ \mathtt{this}^{\downarrow}$ label, where c is the name of the current conversation, thus ensuring that the enclosing and current conversations are the same.

Given this informal understanding, we may now present the definition of the transition relations. For clarity, we split the presentation into two sets of rules: the one in Figure 4.2 contains the rules for the basic operators, which essentially correspond to the ones found in the π-Calculus transition relation (see [85]) and that of the π_{lab}-Calculus (see Figure 2.2), except for the duality which is here refined by the synchronization algebra ($\bullet$); the other in Figure 4.3 groups the rules specific to the Conversation Calculus.

Definition 4.3.14 (Transition Relations)
The transition relations $\{\xrightarrow{\lambda} \mid \lambda \in \mathcal{T}\}$ are defined by the rules in Figure 4.2 and Figure 4.3.

$$l^d!(\tilde{a}).P \xrightarrow{l^d!(\tilde{a})} P \; (Out) \qquad l^d?(\tilde{x}).P \xrightarrow{l^d?(\tilde{a})} P\{\tilde{x}/\tilde{a}\} \; (In) \qquad \frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad j \in I}{\Sigma_{i \in I} \; \alpha_i.P_i \xrightarrow{\lambda} Q} (Sum)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a \in out(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{(\nu a)\lambda} Q} (Open) \qquad \frac{P \xrightarrow{\lambda} Q \quad a \notin na(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{\lambda} (\boldsymbol{\nu} a)Q} (Res)$$

$$\frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \,\#\, fn(R)}{P \mid R \xrightarrow{\lambda} Q \mid R} (Par\text{-}l) \qquad \frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \,\#\, fn(R)}{R \mid P \xrightarrow{\lambda} R \mid P} (Par\text{-}r)$$

$$\frac{P \xrightarrow{(\nu\tilde{a})\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \tilde{a} \,\#\, fn(Q) \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')} (Close\text{-}l)$$

$$\frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{(\nu\tilde{a})\lambda_2} Q' \quad \tilde{a} \,\#\, fn(P) \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')} (Close\text{-}r)$$

$$\frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} P' \mid Q'} (Comm) \qquad \frac{P\{\mathcal{X}/\mathsf{rec}\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathsf{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q} (Rec)$$

Figure 4.2: Transition Rules for Basic Operators (π-Calculus).

The labelled transition system describes not only the autonomous evolution of processes, captured by τ transitions, but also the interactions with the external environment. When we focus on closed systems, not subject to interaction with the environment, we are only interested on autonomous behavior (τ transitions), which we capture by a notion of reduction.

Definition 4.3.15 (Reduction)
The relation of reduction between processes, noted $P \to Q$, is defined as $P \xrightarrow{\tau} Q$. Also, we denote by $\xrightarrow{*}$ the reflexive transitive closure of the reduction relation.

Using the behavioral descriptions captured by the labeled transition system we define the behavioral semantics of the core CC, which then precisely characterizes when two core CC systems have the same behavior, and thus correspond to the same specification from a behavioral point of view. We present the behavioral semantics of core CC systems in the next section.

4.4 Behavioral Semantics

In this section we define the behavioral semantics of the core CC which allows us to characterize a notion of semantic object, consisting in a behavioral equivalence class which groups systems that exhibit the same behavior, and report results that first corroborate our syntactically chosen constructs at the semantic level, and second illuminate on the communication model of the CC.

Building on the notion of observation over processes captured by the labelled transition system given in Definition 4.3.14, we now characterize the core CC semantic object by an observational equivalence, expressed in terms of standard notion of bisimilarity — introduced by Park [81] and since then widely used in models for concurrency, in particular the π-Calculus.

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q \quad (c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda^{\downarrow}} c \blacktriangleleft [Q]} (Here) \qquad \frac{P \xrightarrow{\lambda} Q \quad (unloc(\lambda))}{c \blacktriangleleft [P] \xrightarrow{c \cdot \lambda} c \blacktriangleleft [Q]} (Loc)$$

$$\frac{P \xrightarrow{\lambda} Q \quad (loc(\lambda), c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda} c \blacktriangleleft [Q]} (Through) \qquad \frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (Tau)$$

$$\frac{P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q}{c \blacktriangleleft [P] \stackrel{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} c \blacktriangleleft [Q]} (ThisHere) \qquad \frac{P \stackrel{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (ThisLoc)$$

$$\mathbf{this}(x).P \stackrel{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} P\{x/c\} \ (This)$$

Figure 4.3: Transition Rules for Conversation Operators.

Definition 4.4.1 (Strong Bisimulation)
A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \# fn(Q)$
then there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

Essentially two processes are related by a bisimulation if we can play a symmetric observation game over them, where the observations performed in one process can be mimicked by the other process and lead to (observationally) equivalent states. We establish an expected basic property of strong bisimulations.

Proposition 4.4.2 (Closure Under Union)
Let $\mathcal{S}$ be a set of strong bisimulations. Then $\bigcup \mathcal{S}$ is a strong bisimulation.

Proof. Follows directly from the definition of strong bisimulation (see Appendix A.2). ■

This property allows us to define strong bisimilarity as the union of all strong bisimulations. Since strong bisimulations are closed under union, strong bisimilarity is a strong bisimulation.

Definition 4.4.3 (Strong Bisimilarity)
Strong bisimilarity, noted $\sim$, is the union of all strong bisimulations.

We establish the following properties of strong bisimilarity: it is an equivalence relation; it is preserved under a standard set of structural laws (cf., π-Calculus structural congruence [85]).

Proposition 4.4.4 (Equivalence Relation)
Strong bisimilarity is an equivalence relation.

Proof. Proofs of reflexivity and symmetry are immediate. Proof of transitivity follows by coinduction on the definition of strong bisimulation (see Appendix A.2). ■

Theorem 4.4.5 (Preservation of Strong Bisimilarity Under Structural Laws)
Given processes P, Q and R, the following axioms hold:

1. $(\boldsymbol{\nu} a)\mathbf{0} \sim \mathbf{0}$.

2. $(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \sim (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P$.

3. *If* $a \notin fn(P)$ *then* $P \mid (\boldsymbol{\nu} a)Q \sim (\boldsymbol{\nu} a)(P \mid Q)$.

4. $P \mid \mathbf{0} \sim P$.

5. $P \mid (Q \mid R) \sim (P \mid Q) \mid R$.

6. $P \mid Q \sim Q \mid P$.

7. *If* $i \in I$ *if and only if* $i \in J$ *then* $\Sigma_{i \in I}\, \alpha_i.P_i \sim \Sigma_{i \in J}\, \alpha_i.P_i$.

8. $\textsf{rec}\ \mathcal{X}.P \sim P\{\mathcal{X}/\textsf{rec}\ \mathcal{X}.P\}$.

Proof. By coinduction on the definition of strong bisimulation (see Appendix A.2). ∎

The result stated in Theorem 4.4.5 can be viewed as a sanity check of the developed operational semantics, as processes that are expected to be structurally identified, according to the standard rules of structural congruence, are also expected to exhibit the same behaviors. In particular notice that $(- \mid -, \mathbf{0})$ is a commutative monoid with respect to strong bisimilarity. Also, notice the behavioral identities involving name restrictions, in particular the identity described in Theorem 4.4.5(3) which expresses the fact that the scope of a name restriction may grow, provided it does not unintendedly capture any other names in its scope.

Next we establish a congruence result for strong bisimilarity, which thus testifies that all the constructs of our calculus may be soundly interpreted as compositional semantic operators on bisimilarity equivalence classes.

Theorem 4.4.6 (Congruence)
Strong bisimilarity is a congruence.

1. *If* $P \sim Q$ *then* $l^d!(\tilde{n}).P \sim l^d!(\tilde{n}).Q$.

2. *If* $P\{\tilde{x}/\tilde{n}\} \sim Q\{\tilde{x}/\tilde{n}\}$ *for all* $\tilde{n}$ *then* $l^d?(\tilde{x}).P \sim l^d?(\tilde{x}).Q$.

3. *If* $P\{x/n\} \sim Q\{x/n\}$ *for all* n *then* $\textsf{this}(x).P \sim \textsf{this}(x).Q$.

4. *If* $\alpha_i.P_i \sim \alpha_i'.Q_i$*, for all* $i \in I$*, then* $\Sigma_{i \in I}\, \alpha_i.P_i \sim \Sigma_{i \in I}\, \alpha_i'.Q_i$.

5. *If* $P \sim Q$ *then* $n \blacktriangleleft [P] \sim n \blacktriangleleft [Q]$.

6. *If* $P \sim Q$ *then* $(\boldsymbol{\nu} a)P \sim (\boldsymbol{\nu} a)Q$.

7. *If* $P \sim Q$ *then* $P \mid R \sim Q \mid R$.

8. *If* $P\{\mathcal{X}/R\} \sim Q\{\mathcal{X}/R\}$*, for all* R*, then* $\textsf{rec}\,\mathcal{X}.P \sim \textsf{rec}\,\mathcal{X}.Q$.

Proof. By coinduction on the definition of strong bisimulation (see Appendix A.2). ∎

Notice that in Theorem 4.4.6(2) we consider the universal instantiation congruence principle for input: if $P\{x/a\} \sim Q\{x/a\}$ for all a then $l^d?(x).P \sim l^d?(x).Q$ (cf., [85] Theorem 2.2.8(2)). Like for the π-Calculus, congruence does not hold for input, if input congruence is interpreted as a

first order algebraic congruence. However, if input is to be viewed as some sort of a function, then it seems sensible to compare two such functions by testing all possible input values.

The congruence result further justifies our choice of syntactical constructs, by showing they are proper functions of behavior: given a single abstract behavior described by two different processes, placing such two processes in a core CC language context will produce two processes that again describe the same abstract behavior. Next, we show other interesting behavioral equations, that confirm basic intuitions about our conversation-based communication model.

Theorem 4.4.7 (Behavioral Identities)
Given processes P and Q, the following axioms hold:

1. $n \blacktriangleleft [\mathbf{0}] \sim \mathbf{0}$.
2. *If* $a \neq c$ *then* $(\boldsymbol{\nu} a)(c \blacktriangleleft [P]) \sim c \blacktriangleleft [(\boldsymbol{\nu} a)P]$.
3. $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \sim n \blacktriangleleft [P \mid Q]$.
4. $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]] \sim n \blacktriangleleft [o \blacktriangleleft [P]]$.
5. $n \blacktriangleleft [l^{\uparrow}!(\tilde{n}).P] \sim l^{\downarrow}!(\tilde{n}).n \blacktriangleleft [P]$.
6. *If* $n \notin \tilde{x}$ *then* $n \blacktriangleleft [l^{\uparrow}?(\tilde{x}).P] \sim l^{\downarrow}?(\tilde{x}).n \blacktriangleleft [P]$.
7. $m \blacktriangleleft [n \blacktriangleleft [l^{\downarrow}!(\tilde{n}).P]] \sim n \blacktriangleleft [l^{\downarrow}!(\tilde{n}).m \blacktriangleleft [n \blacktriangleleft [P]]]$.
8. *If* $\{m, n\} \# \tilde{x}$ *then* $m \blacktriangleleft [n \blacktriangleleft [l^{\downarrow}?(\tilde{x}).P]] \sim n \blacktriangleleft [l^{\downarrow}?(\tilde{x}).m \blacktriangleleft [n \blacktriangleleft [P]]]$.

Proof. By coinduction on the definition of strong bisimulation (see Appendix A.2). ∎

Theorem 4.4.7(3) captures the notion of conversation context as a single medium accessible through distinct pieces. Theorem 4.4.7(4) expresses the fact that processes may only interact in the conversation in which they are located and in the enclosing one (via $\uparrow$ communications).

Example 4.4.8 *Notice however that there are processes P and Q such that:*

$$n \blacktriangleleft [m \blacktriangleleft [P] \mid Q] \not\sim m \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \tag{4.4.8.1}$$

Consider processes R_1 and R_2 defined as follows:

$$R_1 \triangleq c \blacktriangleleft \left[b \blacktriangleleft \left[l^{\uparrow}!(a)\right] \mid l^{\downarrow}?(x)\right] \qquad R_2 \triangleq b \blacktriangleleft \left[l^{\uparrow}!(a)\right] \mid c \blacktriangleleft \left[l^{\downarrow}?(x)\right]$$

Since R_1 exhibits a τ transition and R_2 does not, we have that $R_1 \not\sim R_2$.

The inequation (4.4.8.1) contrasts with Theorem 4.4.7(3): the relation between a conversation and its caller must be preserved. Although the processes given in Example 4.4.8 do not exhibit the same behavior, we may place them in a specific context and obtain equivalent configurations.

Example 4.4.9 *Consider processes R_1 and R_2 of Example 4.4.8 and static context $c \blacktriangleleft [\cdot]$:*

$$c \blacktriangleleft [R_1] \triangleq c \blacktriangleleft \left[c \blacktriangleleft \left[b \blacktriangleleft \left[l^{\uparrow}!(a)\right] \mid l^{\downarrow}?(x)\right]\right] \qquad c \blacktriangleleft [R_2] \triangleq c \blacktriangleleft \left[b \blacktriangleleft \left[l^{\uparrow}!(a)\right] \mid c \blacktriangleleft \left[l^{\downarrow}?(x)\right]\right]$$

We then have that $c \blacktriangleleft [R_1] \sim c \blacktriangleleft [R_2]$, *as can be proved in more general terms as follows:*

$$
\begin{aligned}
n \blacktriangleleft [n \blacktriangleleft [m \blacktriangleleft [P] \mid Q]] &\sim && \textit{(Theorem 4.4.7(3))}\\
n \blacktriangleleft [n \blacktriangleleft [m \blacktriangleleft [P]] \mid n \blacktriangleleft [Q]] &\sim && \textit{(Theorem 4.4.7(3))}\\
n \blacktriangleleft [n \blacktriangleleft [m \blacktriangleleft [P]]] \mid n \blacktriangleleft [n \blacktriangleleft [Q]] &\sim && \textit{(Theorem 4.4.7(4))}\\
n \blacktriangleleft [m \blacktriangleleft [P]] \mid n \blacktriangleleft [n \blacktriangleleft [Q]] &\sim && \textit{(Theorem 4.4.7(3))}\\
n \blacktriangleleft [m \blacktriangleleft [P] \mid n \blacktriangleleft [Q]]
\end{aligned}
$$

Notice the caller conversation relationship is preserved: P *is running in conversation* m *and its caller conversation is* n*;* Q *is running in conversation* n *and its caller conversation is also* n.

Example 4.4.9 already hints on the fundamental role played by the synchronization algebra in guaranteeing Theorem 4.4.7(3) — the split rule — since it allows for synchronizations that depend on the external environment of processes to occur. If we were to restrict synchronizations to those of dual labels (λ_1 and λ_2 such that $\lambda_1 \bullet \lambda_2 = \tau$), then the split rule would no longer hold, as the following example shows.

Example 4.4.10 *Consider processes* R_1 *and* R_2 *defined as follows:*

$$R_1 \triangleq c \blacktriangleleft \left[l^{\downarrow}?(x)\right] \mid c \blacktriangleleft \left[c \blacktriangleleft \left[l^{\downarrow}!(a)\right]\right] \qquad\qquad R_2 \triangleq c \blacktriangleleft \left[l^{\downarrow}?(x) \mid c \blacktriangleleft \left[l^{\downarrow}!(a)\right]\right]$$

Considering the behavioral identity given by Theorem 4.4.7(3) we have that $R_1 \sim R_2$. *Both* R_1 *and* R_2 *may exhibit* τ, $c\ l^{\downarrow}!(a)$ *and* $c\ l^{\downarrow}?(b)$ *transitions, for some* b, *and arrive at states equated by Theorem 4.4.7(3). However, the* τ *transition of process* R_1 *is derived from dual transitions:*

$$\frac{c \blacktriangleleft [l^{\downarrow}?(x)] \xrightarrow{c\ l^{\downarrow}?(a)} c \blacktriangleleft [\mathbf{0}] \qquad c \blacktriangleleft [c \blacktriangleleft [l^{\downarrow}!(a)]] \xrightarrow{c\ l^{\downarrow}!(a)} c \blacktriangleleft [c \blacktriangleleft [\mathbf{0}]] \qquad c\ l^{\downarrow}?(a) \bullet c\ l^{\downarrow}!(a) = \tau}{c \blacktriangleleft [l^{\downarrow}?(x)] \mid c \blacktriangleleft [c \blacktriangleleft [l^{\downarrow}!(a)]] \xrightarrow{\tau} c \blacktriangleleft [\mathbf{0}] \mid c \blacktriangleleft [c \blacktriangleleft [\mathbf{0}]]}$$

while the τ *transition of process* R_2 *is derived from a* $c\ \mathtt{this}^{\downarrow}$ *transition:*

$$\dfrac{\dfrac{l^{\downarrow}?(x) \xrightarrow{l^{\downarrow}?(a)} \mathbf{0} \qquad c \blacktriangleleft [l^{\downarrow}!(a)] \xrightarrow{c\ l^{\downarrow}!(a)} c \blacktriangleleft [\mathbf{0}] \qquad l^{\downarrow}?(a) \bullet c\ l^{\downarrow}!(a) = c\ \mathtt{this}^{\downarrow}}{l^{\downarrow}?(x) \mid c \blacktriangleleft [l^{\downarrow}!(a)] \xrightarrow{c\ \mathtt{this}^{\downarrow}} \mathbf{0} \mid c \blacktriangleleft [\mathbf{0}]}}{c \blacktriangleleft [l^{\downarrow}?(x) \mid c \blacktriangleleft [l^{\downarrow}!(a)]] \xrightarrow{\tau} c \blacktriangleleft [\mathbf{0} \mid c \blacktriangleleft [\mathbf{0}]]}$$

The behavioral laws shown in Theorem 4.4.7 hint on the abstract spatial model of core CC processes, and pave the way for the establishment of a normal form result and of an alternative characterization of the operational semantics based on a notion of structural congruence. We describe the former in section 4.4.1 and the latter in section 4.4.2. To complete our basic study on the behavioral semantics of core CC systems, we present the standard weak variant of strong bisimilarity in Section 4.4.3.

4.4.1 Normal Form

The behavioral identities stated in Theorem 4.4.7, in particular Theorem 4.4.7(3) and Theorem 4.4.7(4), allow us to prove an interesting normal form property, that contributes to illumi-

nate the abstract spatial structure of core CC systems. In order to present the normal form result we first characterize the *process normal form* and an auxiliary notion of *active processes*.

Definition 4.4.11 (Active Process)
Active processes are defined as follows:

$$U, V \ ::= \ \Sigma_{i \in I}\, \alpha_i.P_i \ \mid \ U \mid V \ \mid \ \mathbf{0} \qquad \text{(Active Processes)}$$

Definition 4.4.12 (Process Normal Form)
We say process P is in normal form if there are a set of active processes $U_0, \ldots, U_k, V_1, \ldots, V_l$ and sets of names $\tilde{a}, \tilde{b}, \tilde{c}, \tilde{d}$, such that:

$$P = (\boldsymbol{\nu}\tilde{a})\ (U_0 \mid b_1 \blacktriangleleft [U_1] \ \mid \ \ldots \ \mid b_k \blacktriangleleft [U_k] \ \mid c_1 \blacktriangleleft [d_1 \blacktriangleleft [V_1]] \ \mid \ \ldots \ \mid c_l \blacktriangleleft [d_l \blacktriangleleft [V_l]])$$

and where names $b_1, \ldots, b_k$ and sequences $c_1 \cdot d_1, \ldots, c_l \cdot d_l$ are pairwise distinct.

An active process is a parallel composition of prefix guarded choice processes. Then a process in normal form is a process where the maximum conversation nesting of an active process is two. We introduce *well-formed processes* to exclude undesired configurations such as, e.g., $\mathbf{rec}\,\mathcal{X}.\mathcal{X}$.

Definition 4.4.13 (Well-Formed Process)
A process P is well-formed if every recursion variable which occurs in it is guarded by a prefix.

We may now state our normal form result.

Proposition 4.4.14 (Normal Form)
If P is a well-formed process then there exists a process Q in normal form such that $P \sim Q$.

Proof. By induction on the number of active processes (see Appendix A.2). ∎

Intuitively, Proposition 4.4.14 states that any process is behaviorally equivalent to a process where the maximum nesting of contexts is two. Notice that the result does not imply the normal form characterization is invariant under process evolution, since active processes may evolve to processes which fall out of the active process characterization (e.g., an active process may evolve to a conversation piece). However, Proposition 4.4.14 does imply that any process reachable from a well-formed process may be written in normal form, since the necessary well-formedness condition on processes is invariant under process evolution.

Corollary 4.4.15 (Normal Form Preservation)
If P is a well-formed process and Q is such that $P \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_k} Q$ then there is process R in normal form such that $Q \sim R$.

Proof. Direct from Proposition 4.4.14. ∎

We may interpret the normal form existence result as follows. A system is composed by several conversation contexts. The set of upward ($\uparrow$) communication paths of a system may be seen as a graph, where the nodes are processes and contexts, and arcs connect processes to their call-ancestor contexts. As each such arc is uniquely defined by its two terminal nodes, so is the

$$
\begin{array}{rcll}
P & \equiv & Q \;\; (\mathit{if}\ P \equiv_\alpha Q) & (\mathit{StructAlpha}) \\
P \mid \mathbf{0} & \equiv & P & (\mathit{StructParZero}) \\
P \mid Q & \equiv & Q \mid P & (\mathit{StructParComm}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\mathit{StructParAssoc}) \\
(\boldsymbol{\nu} a)\mathbf{0} & \equiv & \mathbf{0} & (\mathit{StructResZero}) \\
(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P & \equiv & (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P & (\mathit{StructResComm}) \\
P \mid (\boldsymbol{\nu} a)Q & \equiv & (\boldsymbol{\nu} a)(P \mid Q) \;\; (a \notin \mathit{fn}(P)) & (\mathit{StructResPar}) \\
\textsf{rec}\ \mathcal{X}.P & \equiv & P\{\mathcal{X}/\textsf{rec}\ \mathcal{X}.P\} & (\mathit{StructRec}) \\
\Sigma_{i \in I}\ \alpha_i.P_i & \equiv & \Sigma_{i \in J}\ \alpha_i.P_i \;\; (\mathit{if}\ i \in I \iff i \in J) & (\mathit{StructSum})
\end{array}
$$

Figure 4.4: Basic Structural Congruence Rules (π-Calculus).

$$
\begin{array}{rcll}
n \blacktriangleleft [\mathbf{0}] & \equiv & \mathbf{0} & (\mathit{StructCZero}) \\
(\boldsymbol{\nu} a)n \blacktriangleleft [P] & \equiv & n \blacktriangleleft [(\boldsymbol{\nu} a)P] \;\; (a \neq n) & (\mathit{StructCRes}) \\
n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] & \equiv & n \blacktriangleleft [P \mid Q] & (\mathit{StructCSplit}) \\
n \blacktriangleleft [m \blacktriangleleft [o \blacktriangleleft [P]]] & \equiv & m \blacktriangleleft [o \blacktriangleleft [P]] & (\mathit{StructCNest}) \\
n \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\uparrow}!(\tilde{m}).P_i\right] & \equiv & \Sigma_{i \in I}\ l_i^{\downarrow}!(\tilde{m}).n \blacktriangleleft [P_i] & (\mathit{StructOutUp}) \\
n \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\uparrow}?(\tilde{x}).P_i\right] & \equiv & \Sigma_{i \in I}\ l_i^{\downarrow}?(\tilde{x}).n \blacktriangleleft [P_i] \;\; (n \notin \tilde{x}) & (\mathit{StructInUp}) \\
n \blacktriangleleft \left[m \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\downarrow}!(\tilde{o}).P_i\right]\right] & \equiv & m \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\downarrow}!(\tilde{o}).n \blacktriangleleft [m \blacktriangleleft [P_i]]\right] & (\mathit{StructOutHere}) \\
n \blacktriangleleft \left[m \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\downarrow}?(\tilde{x}).P_i\right]\right] & \equiv & m \blacktriangleleft \left[\Sigma_{i \in I}\ l_i^{\downarrow}?(\tilde{x}).n \blacktriangleleft [m \blacktriangleleft [P_i]]\right] \;\; (\{n, m\} \# \tilde{x}) & \\
& & & (\mathit{StructInHere})
\end{array}
$$

Figure 4.5: Conversation Structural Congruence Rules.

communication structure of an arbitrary process defined (up to bisimilarity) by a system where the (syntactic) nesting of contexts is of at most depth two. Intuitively, the structure suggested here represents the join-subconversation relation of concurrently ongoing conversations. Then, the normal form of Proposition 4.4.14 is analogous to a flattened representation of such a graph.

4.4.2 Structural Congruence Based Operational Semantics

In this section we provide an alternative presentation of the operational semantics, based on a notion of structural congruence and a reduction relation. Structural congruence groups processes into equivalence classes, abstracting from syntactic information that does not condition the abstract behavior of processes (cf., strong bisimilarity). We thus define structural congruence for core CC processes based on the behavioral axioms stated in Theorem 4.4.7 and Theorem 4.4.5.

Definition 4.4.16 (Structural Congruence)
Structural congruence, noted $\equiv$, is the least congruence on processes that satisfies the rules in Figure 4.4 and Figure 4.5.

Notice that in the axioms (*StructOutUp*), (*StructInUp*), (*StructOutHere*) and (*StructInHere*) of Figure 4.5 we consider that prefix guarded choice processes are always defined with actions defined on the same direction and of the same type (input or output). This restriction is

$$l^d!(\tilde{a}).P + S_1 \mid l^d?(\tilde{x}).Q + S_2 \to P \mid (Q\{\tilde{x}/\tilde{a}\}) \quad (\mathit{RedComm})$$

$$c \blacktriangleleft [\textsf{this}(x).P + S] \to c \blacktriangleleft [P\{x/c\}] \quad (\mathit{RedThis})$$

$$\frac{P \to Q}{(\boldsymbol{\nu} a)P \to (\boldsymbol{\nu} a)Q}(\mathit{RedRes}) \qquad \frac{P \to Q}{P \mid R \to Q \mid R}(\mathit{RedPar})$$

$$\frac{P \to Q}{c \blacktriangleleft [P] \to c \blacktriangleleft [Q]}(\mathit{RedConv}) \qquad \frac{P \equiv P' \to Q' \equiv Q}{P \to Q}(\mathit{RedStruct})$$

Figure 4.6: Reduction Rules.

required, since the syntax of processes would not support the rewritings considered in such rules, e.g., summations between different conversations. However, it is sensible since we are excluding summations that mix actions in different conversations and conversation identity accesses altogether. To simplify presentation we also exclude the mixed summation of inputs and outputs with the same direction: however this latter case could be captured by a slight tuning of the structural axioms. For the results of this section we consider only such processes.

Convention 4.4.17 *For the remaining of Section 4.4.2 we consider only processes where all prefix guarded choices are defined on the same action type and direction.*

For the sake of presentation we separate the presentation of the structural congruence rules in Figure 4.4 and Figure 4.5. The rules shown in Figure 4.4 correspond to usual π-Calculus structural axioms, and coincide with the behavioral axioms of Theorem 4.4.5. The rules shown in Figure 4.5 describe identities specific to the core CC, in particular to the conversation access construct, and coincide with the behavioral identities stated in Theorem 4.4.7. We then have the following expected result.

Proposition 4.4.18 (Structural Congruence Included in Bisimilarity)
We have that $\equiv \subseteq \sim$.

Proof. By coinduction on the definition of strong bisimulation (see Appendix A.2). ∎

Structural congruence thus allows us to characterize the reduction relation by describing the evolution of the basic representatives of the equivalence classes, and then close the reduction relation under such equivalence classes. For describing reduction in the core CC we consider the case of a message exchange and the case when a conversation identity is read by a **this** prefix. We thus define reduction.

Definition 4.4.19 (Reduction)
The reduction relation between processes, noted $P \to Q$*, is the least relation inductively defined by the rules in Figure 4.6.*

We establish an expected result which identifies τ-transitions ($\xrightarrow{\tau}$) with reductions ($\to$). Since the labelled transition system (Definition 4.3.14) is not closed under structural congruence, in order to compare τ-transitions with reductions we must explicitly introduce the structural congruence equivalence classes in the arrival state of the τ-transition.

Notation 4.4.20 *We write $P \xrightarrow{\tau}\equiv Q$ to denote there is Q' such that $P \xrightarrow{\tau} Q'$ and $Q' \equiv Q$.*

Theorem 4.4.21 (Reductions Match τ-Transitions)
Let P, Q be processes. We have that $P \rightarrow Q$ if and only if $P \xrightarrow{\tau}\equiv Q$.

Proof. The $\Rightarrow$ and $\Leftarrow$ directions follows by induction on the length of the derivation of $P \rightarrow Q$ and of $P \xrightarrow{\tau} Q$, respectively (see Appendix A.2). ■

The structural congruence we introduced can be viewed as not standard due to the axioms involving the conversation access construct. However, the reduction relation definition given above, based on such structural congruence, is as canonical as the one found in π-Calculus presentations. Furthermore we prove that reductions defined in such canonical way coincide with τ-transitions, as expected.

4.4.3 Weak Bisimilarity

In this section we present a variant of the behavioral equivalence provided by strong bisimilarity, by introducing a notion of weak bisimilarity which abstracts away from internal actions and focuses solely on the possible interactions processes may have with the external environment. In such way, we relax our notion of abstract behavior, allowing for processes which differ solely in some internal actions to be identified. Intuitively, one may think of an external observer that does not have the capability to observe the internal actions, and can only test the behavior of systems by interacting with them. We first define weak transitions, which essentially collapse together sequences of τ transitions, possibly along with some external action.

Definition 4.4.22 (Weak Transition)
We denote by $\xrightarrow{\tau}^*$ the reflexive transitive closure of $\xrightarrow{\tau}$. Then we say P has a weak λ transition to Q, noted $P \xRightarrow{\lambda} Q$, if it can be derived from:

$$
\begin{array}{lll}
 & P \xrightarrow{\tau}^* P' \xrightarrow{\lambda} Q' \xrightarrow{\tau}^* Q & (\textit{if } \lambda \neq \tau) \\
\textit{or} & P \xrightarrow{\tau}^* Q & (\textit{if } \lambda = \tau)
\end{array}
$$

Weak transitions thus allow us to abstract away from the internal actions a process may have. Using such weak transitions we may provide with the standard definition of weak bisimulation.

Definition 4.4.23 (Weak Bisimulation)
A weak bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \# fn(Q)$
then there is Q' such that $Q \xRightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

Notice that to match a behavior of process P, process Q may freely perform a number of internal actions in the meanwhile (and symmetrically). So, two processes are related by a weak bisimulation if we can play a symmetric observation game where the observations performed in one process can be reproduced in the other process, up to some internal actions. We establish a basic property of weak bisimulations.

Proposition 4.4.24 (Closure Under Union)
Let $\mathcal{S}$ be a set of weak bisimulations. Then $\bigcup \mathcal{S}$ is a weak bisimulation.

Proof. Follows directly from the definition of weak bisimulation. ■

This property allows us to define weak bisimilarity as the union of all weak bisimulations. Since weak bisimulation is closed under union, weak bisimilarity is itself a weak bisimulation.

Definition 4.4.25 (Weak Bisimilarity)
Weak bisimilarity, noted $\approx$, is the union of all weak bisimulations.

We establish the following properties that weak bisimilarity is an equivalence relation and that it is preserved under a standard set of structural laws.

Proposition 4.4.26 (Equivalence Relation)
Weak bisimilarity is an equivalence relation.

Proof. Proof of reflexivity is immediate and of symmetry follows directly from the definition. Proof of transitivity follows by coinduction on the definition of weak bisimulation, along the lines of the proof of Proposition 4.4.4. ■

Theorem 4.4.27 (Preservation of Weak Bisimilarity Under Structural Laws)
Given processes P, Q and R, the following axioms hold:

1. $(\boldsymbol{\nu} a)\mathbf{0} \approx \mathbf{0}$.
2. $(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \approx (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P$.
3. *If* $a \notin fn(P)$ *then* $P \mid (\boldsymbol{\nu} a)Q \approx (\boldsymbol{\nu} a)(P \mid Q)$.
4. $P \mid \mathbf{0} \approx P$.
5. $P \mid (Q \mid R) \approx (P \mid Q) \mid R$.
6. $P \mid Q \approx Q \mid P$.
7. *If* $i \in I$ *if and only if* $i \in J$ *then* $\Sigma_{i \in I}\, \alpha_i.P_i \approx \Sigma_{i \in J}\, \alpha_i.P_i$.
8. $\textsf{rec}\ \mathcal{X}.P \approx P\{\mathcal{X}/\textsf{rec}\ \mathcal{X}.P\}$.

Proof. By coinduction on the definition of weak bisimulation, along the lines of the proof of Theorem 4.4.5. ■

Hence, we have that $(- \mid -, \mathbf{0})$ is a commutative monoid with respect to weak bisimilarity, and other usual structural rules, for instance the scope extrusion rule (Theorem 4.4.27(3)). As for strong bisimilarity, we also establish that weak bisimilarity is a congruence.

Theorem 4.4.28 (Congruence)
Weak bisimilarity is a congruence.

1. *If* $P \approx Q$ *then* $l^d!(\tilde{n}).P \approx l^d!(\tilde{n}).Q$.

2. *If* $P\{\tilde{x}/\tilde{n}\} \approx Q\{\tilde{x}/\tilde{n}\}$ *for all* $\tilde{n}$ *then* $l^d?(\tilde{x}).P \approx l^d?(\tilde{x}).Q$.

3. *If* $P\{x/n\} \approx Q\{x/n\}$ *for all* n *then* $\textsf{this}(x).P \approx \textsf{this}(x).Q$.

4. *If* $\alpha_i.P_i \approx \alpha'_i.Q_i$*, for all* $i \in I$*, then* $\Sigma_{i\in I}\, \alpha_i.P_i \approx \Sigma_{i\in I}\, \alpha'_i.Q_i$.

5. *If* $P \approx Q$ *then* $n \blacktriangleleft [P] \approx n \blacktriangleleft [Q]$.

6. *If* $P \approx Q$ *then* $(\boldsymbol{\nu} a)P \approx (\boldsymbol{\nu} a)Q$.

7. *If* $P \approx Q$ *then* $P \mid R \approx Q \mid R$.

8. *If* $P\{\mathcal{X}/R\} \approx Q\{\mathcal{X}/R\}$*, for all* R*, then* $\textsf{rec}\,\mathcal{X}.P \approx \textsf{rec}\,\mathcal{X}.Q$.

Proof. By coinduction on the definition of weak bisimulation, along the lines of the proof of Theorem 4.4.6. ∎

The congruence result for weak bisimilarity may seem surprising as it does not hold in the π-Calculus. We next describe, in a generic way, the typical counter-example for the result.

Example 4.4.29 *Consider processes* $\alpha.P$ *and* Q *such that* $Q \xrightarrow{\tau} \alpha.P$ *and such observation is the only one that can be performed over* Q*. Clearly, we have that* $\alpha.P \approx Q$*. Now consider a summation context* $\cdot + R$ *for which we then have, in general, that:*

$$\alpha.P + R \not\approx Q + R$$

since the process on the right hand side can evolve to $\alpha.P$*, hence* $Q+R \rightarrow \alpha.P$*, and this behavior cannot be mimicked by the process on the left hand side, as to choose the* $\alpha.P$ *branch the process would have to exhibit* α*, thus* $\alpha.P + R \not\rightarrow \alpha.P$*, neither can the process do "nothing" as it would remain with the possibility of choosing the* R *branch further along, thus* $\alpha.P + R \stackrel{\tau}{\Longrightarrow} \alpha.P + R$ *but* $\alpha.P + R \not\approx \alpha.P$.

Thus, in general, weak bisimilarity is not a congruence with respect to the choice construct. However, in the Conversation Calculus it is not possible to write such a process Q, such that $Q \xrightarrow{\tau} \alpha.P$ is the only action that can be observed in Q, and that can be placed in a summation context. More specifically, our syntax for choice defines only outputs and inputs as guards for the prefixes which cannot by themselves perform a τ transition, along with conversation awareness primitives which exhibit $c\ \mathtt{this}^{\downarrow}$ transitions, for some c. Hence it is not possible to specify a prefixed process, which is the only kind of process admissible in our prefix guarded choice, that only has internal actions and no external interactions.

Some calculi consider a τ prefix so as to explicitly denote some internal action. In such cases, the process Q of Example 4.4.29 could be of the form $\tau.\alpha.P$. The closest we get to specifying such a τ prefix in our model is given by the process $\tau.P \triangleq (\boldsymbol{\nu} c)(c \blacktriangleleft [\textsf{this}(x).P])$ but, again, such process may not be placed in a summation context.

This informal discussion completes our presentation of the study of the basic behavioral semantics of core CC systems. Our results provide with evidence that our mathematical interpretation of CC processes enjoys some fundamental properties, namely the congruence result. Given such results that demonstrate, to some extent, the theoretical soundness of our approach, we proceed to a more practical perspective so as to see the sort of systems that can be expressed, in a rather simple way, in our model.

4.5 Idioms for Service-Oriented Computing

The core CC focuses on the fundamental notions of conversation context and message-based communication. From these basic mechanisms, useful programming abstractions for service-oriented systems may be idiomatically defined, namely service definition and instantiation constructs, and the conversation join construct, which is crucial to our approach to multiparty conversations. These constructs may be embedded in a simple way in the minimal calculus, without hindering the flexibility of modeling and analysis.

We present the service-oriented idioms, along with their translation in lower level communication primitives, and informally describe their meaning. We specify the service definition idiom by $\textbf{def}\, s \Rightarrow P$, which publishes a service named s in the current conversation. Process P specifies the code that will run in the service conversation, upon service instantiation, which implements the service provider role in the conversation. The service definition idiom is implemented in basic communication primitives as follows:

$$\textbf{def}\, s \Rightarrow P \quad \triangleq \quad s^{\downarrow}?(x).x \blacktriangleleft [P] \qquad (x \notin \mathit{fv}(P))$$

Essentially, the service definition specifies a message — labeled by the name of the service s — is received, carrying the identity of the service conversation. Then, code P will run in such received conversation. Service definitions must be placed in appropriate conversation contexts (cf., methods in objects). For instance, the following code specifies `BuyService` is published in the *Seller* context:

$$\mathit{Seller} \blacktriangleleft [\, \textbf{def}\, \texttt{BuyService} \Rightarrow \mathit{SellerCode}\,]$$

Typically, services once published are persistent in the sense they can be instantiated several times. To model such persistent services we introduce the recursive variant of service definition.

$$\star\textbf{def}\, s \Rightarrow P \quad \triangleq \quad \textbf{rec}\, \mathcal{X}.s^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) \qquad (x \notin \mathit{fv}(P))$$

Persistent service definitions are specified so as to always be ready to receive a service instantiation request, handling each request in the conversation received in each service instantiation message.

The idiom that supports the instantiation of a published service is noted by $\textbf{new}\, n \cdot s \Leftarrow Q$. The **new** idiom specifies the conversation where the service is published at (n), the name of the service (s) and the code that will run on the service client side (Q). A service instantiation resulting from a synchronization from a published service **def** and an instantiation **new** results in the creation of a fresh conversation that is shared between service provider and service caller. We translate the **new** idiom in the basic primitives of the CC as follows:

$$\textbf{new}\, n \cdot s \Leftarrow Q \quad \triangleq \quad (\boldsymbol{\nu} c)(n \blacktriangleleft \left[s^{\downarrow}!(c)\right] \mid c \blacktriangleleft [Q]) \qquad (c \notin (\mathit{fn}(Q) \cup \{n\}))$$

The service instantiation is then realized by means of a message exchange in conversation n, where the service is published at, being the message labeled by the name of the service s and carrying a newly created name c that identifies the conversation where the service interaction is to take place. In parallel to the message output that instantiates the service, we find the code of the client role Q, running in the freshly created conversation c. Notice that Q is already

$$
\begin{array}{llll}
\textsf{\textbf{def}}\ s \Rightarrow P & \triangleq & s^{\downarrow}?(x).x \blacktriangleleft [P] & \text{(Service Definition)} \\
\textsf{\textbf{new}}\ n \cdot s \Leftarrow Q & \triangleq & (\boldsymbol{\nu} c)(n \blacktriangleleft [s^{\downarrow}!(c)] \mid c \blacktriangleleft [Q]) & \text{(Service Instantiation)} \\
\textsf{\textbf{join}}\ n \cdot s \Leftarrow Q & \triangleq & \textsf{\textbf{this}}(x).(n \blacktriangleleft [s^{\downarrow}!(x)] \mid Q) & \text{(Conversation Join)} \\
\star\textsf{\textbf{def}}\ s \Rightarrow P & \triangleq & \textsf{\textbf{rec}}\, \mathcal{X}.s^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) & \text{(Replicated Service Definition)}
\end{array}
$$

Figure 4.7: Service Idioms.

active, although it has to wait for the server side to pick up the service conversation identity to start interacting in conversation c, by means of $\downarrow$ directed messages. Notice also that process Q can interact in the conversation where the service instantiation request lies, using $\uparrow$ directed messages. To instantiate the `BuyService` published in the *Seller* conversation we write:

$$\textsf{\textbf{new}}\ \mathit{Seller} \cdot \texttt{BuyService} \Leftarrow \mathit{BuyerCode}$$

Service definition and instantiation idioms have a tight correspondence with session creation primitives, where also a freshly created medium is shared between two distinct parties. The difference here is that we introduce them as mere idioms of the language, using more general communication primitives to program them (π-Calculus bound name communication), while in most session-based approaches we find such session creation mechanisms as native primitives of the language. The idiom we present next, however, does not find any correspondence in session-based approaches, thus is a distinguishing feature of our approach.

In the core CC, conversation identifiers may be manipulated by processes if needed (accessed via the $\textsf{\textbf{this}}(x).P$), passed around in messages and subject to scope extrusion: this allows us to model multiparty conversations by the progressive access of multiple, dynamically determined partners, to an ongoing conversation. Joining of another partner to an ongoing conversation is a frequent programming idiom, that may be conveniently abstracted by the $\textsf{\textbf{join}}\ n \cdot s \Leftarrow Q$ construct. The semantics of the **join** expression is similar to the service instantiation construct **new**: the key difference is that while **new** creates a fresh *new* conversation, **join** allows a service s published at n to join the *current* conversation, and continue interacting as specified by Q. The join idiom is implemented using the core CC primitives as follows:

$$\textsf{\textbf{join}}\ n \cdot s \Leftarrow Q \ \triangleq\ \textsf{\textbf{this}}(x).(n \blacktriangleleft \big[s^{\downarrow}!(x)\big] \mid Q) \qquad (x \notin (\mathit{fv}(Q) \cup \{n\}))$$

Notice that the current conversation identity is accessed via the **this** primitive, and passed along in service message s exchanged in the conversation n where s is published at. Process Q continues to interact in the current conversation (the same that was accessed in the **this**). We present the definition of our service-oriented idioms.

Definition 4.5.1 (Service-Oriented Idioms)
The definition of the service-oriented idioms is given in Figure 4.7, where x and c are fresh.

Using such idioms, multiparty conversations may be progressively and dynamically formed, starting from dyadic ones created by service instantiation. In the following sections we revisit the examples described in the Introduction (Section 1.2.4): to illustrate the underlying mechanics of the idioms we describe in detail the interactions of the purchase service collaboration; then we provide a more informed description of the Finance Portal implementation.

4.5.1 Proving Interaction in the Purchase Scenario

We go back to our running example of the purchase service collaboration so as to show how the service oriented idioms work concretely, in the light of the implementations we introduced above. We next show the code — using our service idioms — for the purchase service collaboration which involves three parties: a buyer that wants to buy a product, a seller that provides a `BuyService` and a shipper that is in charge of executing the delivery, each represented idiomatically by a distinct conversation (*Buyer*, *Seller* and *Shipper*).

$$
\begin{array}{l}
\mathit{Buyer} \blacktriangleleft [\ \textbf{new}\ \mathit{Seller} \cdot \texttt{BuyService} \Leftarrow \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ] \\
\ | \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ | \\
\qquad\qquad \textbf{def}\ \texttt{BuyService} \Rightarrow \texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{join}\ \mathit{Shipper} \cdot \texttt{DeliveryService} \Leftarrow \texttt{product}^{\downarrow}!(\mathit{prod})\ ] \\
\ | \\
\mathit{Shipper} \blacktriangleleft [\ \textbf{def}\ \texttt{DeliveryService} \Rightarrow \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]
\end{array}
$$

Notice that by using the service idioms we avoid any explicit reference to the service conversation name. Thus, our service idioms allow for service conversation handling to be conveniently abstracted. Also, the distinct directions in messages, $\downarrow$ and $\uparrow$, allows us to specify messages that are to be exchanged in the implicitly defined current and enclosing conversations, respectively, where the former can then support the interactions of the service collaboration, while the second allows for accessing resources located in the service provider context, or interact with the client processes that requested the service instantiation.

Now consider the translation of the system shown above where the service idioms are replaced by their implementation in the basic communication primitives:

$$
\begin{array}{l}
\mathit{Buyer} \blacktriangleleft [\ (\boldsymbol{\nu} c)(\mathit{Seller} \blacktriangleleft [\ \texttt{BuyService}^{\downarrow}!(c)\ ]\ \ |\ c \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\ | \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}\ | \\
\qquad\qquad \texttt{BuyService}^{\downarrow}?(x).x \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad |\ \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\ ] \\
\ | \\
\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]\ ]
\end{array}
$$

This translation is already revealing on the intended behavior of the system. For starters, buyer and seller are to interact and establish a fresh conversation between them, which is achieved by the exchange of message `BuyService` in conversation *Seller*, carrying the identity of a freshly created conversation c. If we look at the individual processes we may then observe that the buyer has the following behavior:

$$
\mathit{Buyer} \blacktriangleleft [(\boldsymbol{\nu} c)(\mathit{Seller} \blacktriangleleft [\texttt{BuyService}^{\downarrow}!(c)])\ |\ (\cdots)] \xrightarrow{(\nu c)\mathit{Seller}\ \texttt{BuyService}^{\downarrow}!(c)} \mathit{Buyer} \blacktriangleleft [(\cdots)]
$$

while the seller exhibits the following transition:

$$\mathit{Seller} \blacktriangleleft \left[\texttt{BuyService}^{\downarrow}?(x).x \blacktriangleleft [(\cdots)]\right] \xrightarrow{\mathit{Seller}\ \texttt{BuyService}^{\downarrow}?(c)} \mathit{Seller} \blacktriangleleft [c \blacktriangleleft [(\cdots)]]$$

which may then synchronize (•) in an internal action (τ) through rule (*Close*) which closes the scope of c so as to now include the seller process. Thus, at this point the system has evolved to:

$$\begin{array}{l}
(\boldsymbol{\nu} c) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{Seller} \blacktriangleleft [\ \mathbf{0}\ ]\ \mid c \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB} \mid \\
\qquad\qquad c \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\quad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\quad \textbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ]\ \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\ ]) \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]\ ]
\end{array}$$

Notice that both buyer and seller now hold an access point to conversation c and can thus interact in it, starting by the exchange of message buy, explained by the synchronization of the following behaviors of the buyer and seller processes:

$$\begin{array}{r}
\mathit{Buyer} \blacktriangleleft [(\cdots) \mid c \blacktriangleleft [\texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).(\cdots)]] \xrightarrow{c\ \texttt{buy}^{\downarrow}!(\mathit{prod})} \\
\mathit{Buyer} \blacktriangleleft [(\cdots) \mid c \blacktriangleleft [\texttt{price}^{\downarrow}?(p).(\cdots)]]
\end{array}$$

and:

$$\begin{array}{r}
\mathit{Seller} \blacktriangleleft [c \blacktriangleleft [\texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}?(\mathit{prod}).(\cdots)]] \xrightarrow{c\ \texttt{buy}^{\downarrow}?(\mathit{prod})} \\
\mathit{Seller} \blacktriangleleft [c \blacktriangleleft [\texttt{askPrice}^{\uparrow}?(\mathit{prod}).(\cdots)]]
\end{array}$$

leading the system to the following configuration:

$$\begin{array}{l}
(\boldsymbol{\nu} c) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{Seller} \blacktriangleleft [\ \mathbf{0}\ ]\ \mid c \blacktriangleleft [\ \texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB} \mid \\
\qquad\qquad c \blacktriangleleft [\ \texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad\quad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad\quad \textbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ]\ \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\ ]) \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]\ ]
\end{array}$$

After that the seller service code interacts with the *PriceDB* in conversation *Seller* so as to determine the price of the indicated product. Notice that a message that is $\uparrow$ directed in conversation c will become a $\downarrow$ message once it crosses the conversation boundary:

$$\frac{\texttt{askPrice}^{\uparrow}!(\mathit{prod}).(\cdots) \xrightarrow{\texttt{askPrice}^{\uparrow}!(\mathit{prod})} (\cdots)}{c \blacktriangleleft [\texttt{askPrice}^{\uparrow}!(\mathit{prod}).(\cdots)] \xrightarrow{\texttt{askPrice}^{\downarrow}!(\mathit{prod})} c \blacktriangleleft [(\cdots)]}$$

After the interactions with the *PriceDB*, the buyer is informed of the price of the product in message `price`. The system has then evolved to:

$$
\begin{array}{l}
(\boldsymbol{\nu} c) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{Seller} \blacktriangleleft [\ \mathbf{0}\ ] \mid c \blacktriangleleft [\ \texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}' \mid \\
\qquad\qquad c \blacktriangleleft [\ \textsf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ] \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ])\ ] \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]\ ]
\end{array}
$$

At this point the **join** implementation code will allow for a third party to join in on the ongoing service collaboration. First the **this** primitive accesses the name of the current conversation:

$$
\frac{\textsf{this}(y).(\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ] \mid \cdots) \xrightarrow{c\ \texttt{this}^{\downarrow}} \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(c)\ ] \mid \cdots)}{c \blacktriangleleft [\textsf{this}(y).(\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ] \mid \cdots)] \xrightarrow{\tau} c \blacktriangleleft [\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(c)\ ] \mid \cdots)]}
$$

Then the name is sent to shipper in message `DeliveryService`, through a bound output transition (cf., the `BuyService` message exchange), which then leads the system to the configuration:

$$
\begin{array}{l}
(\boldsymbol{\nu} c) \\
(\mathit{Buyer} \blacktriangleleft [\ \mathit{Seller} \blacktriangleleft [\ \mathbf{0}\ ] \mid c \blacktriangleleft [\ \texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB}' \mid \\
\qquad\qquad c \blacktriangleleft [\ \mathit{Shipper} \blacktriangleleft [\ \mathbf{0}\ ] \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\ ] \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ c \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ])\ ]
\end{array}
$$

where the scope of the restriction on c now also includes the shipper process. Then, the remaining interactions in conversation c can take place, starting by the `product` message exchange between seller and shipper, after which message `details` is exchanged between shipper and buyer. Notice that the seller party does not lose access to conversation c after passing the name to the shipper party, and that, actually, seller and shipper get to interact in message `product` in the delegated conversation.

4.5.2 Programming a Finance Portal

In this section we revisit the Finance portal implementation so as to show how the several primitives and idioms of the language can be combined, allowing to model complex interaction patterns in a rather simple way. We model a credit request scenario, where a bank client, a bank clerk and a bank manager participate, mediated through a bank portal. The client starts

by invoking a service available in the bank portal and places the credit request, providing his identification and the desired amount. The implementation of such client in core CC is then:

$$
\begin{array}{l}
\mathit{Client} \blacktriangleleft [\\
\quad \mathit{ClientTerminal}\\
\quad |\\
\quad \textbf{new}\ \mathit{BankPortal} \cdot \texttt{CreditRequest} \Leftarrow\\
\qquad \texttt{request}^{\downarrow}!(\mathit{myId}, \mathit{amount}).\\
\qquad\quad (\texttt{requestApproved}^{\downarrow}?().\texttt{transferDate}^{\downarrow}!(\mathit{date}).\texttt{approved}^{\uparrow}!()\\
\qquad\quad +\\
\qquad\quad \texttt{requestDenied}^{\downarrow}?().\texttt{denied}^{\uparrow}!())\ \ ]
\end{array}
$$

The client code for the service instantiation specifies the messages that are to be exchanged in the service conversation by using $\downarrow$ messages. First a message `request` is sent, after which one of two messages (either `requestApproved` or `requestDenied`) informing on the decision is received. Only after receiving one of such messages is the *ClientTerminal* informed (correspondingly) of the final decision. Notice that the service code interacts with the *ClientTerminal* process by means of $\uparrow$ messages `approved` or `denied`. In fact, from the point of view of *ClientTerminal* the external interface of the service instance can be characterized by the process $\texttt{approved}^{\downarrow}!() + \texttt{denied}^{\downarrow}!()$.

Next we show the code of the `CreditRequest` service published at the *BankPortal* context. The service is persistently available, which is represented by the $\star$ annotation.

$$
\begin{array}{l}
\mathit{BankPortal} \blacktriangleleft [\\
\quad \star\ \textbf{def}\ \texttt{CreditRequest} \Rightarrow\\
\qquad \texttt{request}^{\downarrow}?(\mathit{userId}, \mathit{amount}).\\
\qquad \textbf{join}\ \mathit{Clerk} \cdot \texttt{RiskAssessment} \Leftarrow\\
\qquad\quad \texttt{assessRisk}^{\downarrow}!(\mathit{userId}, \mathit{amount}).\\
\qquad\quad \texttt{riskVal}^{\downarrow}?(\mathit{risk}).\\
\qquad\quad \textbf{if}\ \mathit{risk}\ =\ \texttt{HIGH}\ \textbf{then}\ \texttt{requestDenied}^{\downarrow}!()\\
\qquad\quad \textbf{else}\ \textbf{this}(\mathit{clientChat}).\\
\qquad\qquad \textbf{new}\ \mathit{Manager} \cdot \texttt{CreditApproval} \Leftarrow\\
\qquad\qquad\quad \texttt{requestApproval}^{\downarrow}!(\mathit{clientChat}, \mathit{userId}, \mathit{amount}, \mathit{risk})\ \ ]
\end{array}
$$

The server code specifies that, in each `CreditRequest` service conversation, a message `request` is received, then message `assessRisk` is sent and then message `riskVal` is received. The first will be exchanged with the service client, while the latter two will be exchanged with the clerk, that is asked to join the ongoing conversation through service `RiskAssessment`. After that, depending on the risk rate the clerk determined for the request, the bank portal is either able to automatically reject the request, in which case it informs the client of such decision by sending message `requestDenied`, or it has to consult the bank manager, creating a new instance of the `CreditApproval` service to that end — notice that a **new** instance is created in this case. However, since the bank manager will reply directly back to the client, the name of the client service conversation is accessed, via the **this**(*clientChat*), and passed along to the manager (in the first argument of message `requestApproval`). This pattern is similar to a **join**: the name of the current conversation is sent to the remote service provider, allowing for it to join in the conversation. The difference with respect to a join is that the remote service will only join the

client conversation to reply back to the client. In some sense, it is as if we only delegate a basic fragment of the client conversation (e.g., the final reply), instead of incorporating the whole functionality provided by `CreditApproval` in the `CreditRequest` service collaboration.

We now show the code for the `CreditApproval` service, assuming there is a *ManagerTerminal* process able to interact with the manager, similarly to the *ClientTerminal* process.

```
Manager ◀ [
    ManagerTerminal
     |
    ⋆ def CreditApproval ⇒
        requestApproval↓?(clientChat, userId, amount, risk).
        this(managerChat).showRequest↑!(managerChat, userId, amount, risk).
            (reject↓?().clientChat ◀ [ requestDenied↓!() ]
            +
            accept↓?().clientChat ◀ [
                        requestApproved↓!().
                        join BankATM · CreditTransfer ⇐
                            orderTransfer↓!(userId, amount) ] ]
```

The `CreditApproval` server code specifies the reception of a `requestApproval` message, carrying the name of the conversation where the final answer is to be given in, after which the identity of the current conversation is accessed, and passed along to *ManagerTerminal* in message `showRequest` in conversation *Manager*. This allows *ManagerTerminal* to reply directly to the "right" conversation, since several copies of the `CreditApproval` service may be running in parallel, and therefore several `showRequest` messages may have to be concurrently handled and replied to by the `ManagerTerminal`: if the replies were to be placed in the *Manager* conversation then they would also compete and be at risk of being picked up by the wrong (unrelated) service instance. The *ManagerTerminal* thus replies in the `CreditApproval` service conversation with either a `reject` message or an `accept` message. After that the credit request client is notified accordingly in the respective conversation. Also, in the case that the credit is approved, the manager asks service `CreditTransfer` published at *BankATM* to join the client conversation (the current conversation for the **join** is the client conversation), so as to place the transfer order.

We now specify the code for the *CreditTransfer* service.

```
BankATM ◀ [
    BankATMProcess
     |
    ⋆ def CreditTransfer ⇒
        orderTransfer↓?(userId, amount).
        transferDate↓?(date).
        scheduleTransfer↑!(userId, amount, date) ]
```

The `CreditTransfer` service code specifies the reception of the transfer order and of the desired date of the transfer, after which forwards the information to a local *BankATMProcess*, which will then schedule the necessary procedure. Notice that the *BankATM* party is only asked to join in the conversation under some circumstances, in such case interacting with the bank manager

in message `orderTransfer` and with the credit request client in message `transferDate`, while otherwise it does not participate at all in the service collaboration.

The system obtained by composing the described processes captures an interesting scenario where, not only the set of multiple participants in the collaboration is dynamically determined, but also the actual maximum number of participants depends on some runtime condition. Such a configuration presents some difficult challenges to analysis techniques that intend to verify properties such as conversation fidelity, since we have to statically account for all possible dynamic joins and leaves of parties to conversations.

4.6 Remarks

Various calculi have been recently proposed with the aim to capture aspects of service-oriented computation. At the root of each one, we find different motivations and methodological approaches. Some intend to model artifacts of the web services technology, in order to develop applied verification techniques (e.g., COWS [71], SOCK [51]), others were introduced in order to demonstrate analysis techniques (e.g., [20, 30]), yet others have the goal of isolating primitives for formalizing and programming service-oriented applications (SCC [12], SSCC [68], CaSPiS [13]) just to refer a few.

We distinguish between the correlation based approaches (e.g., [51, 71]) and session based approaches (e.g., [12, 13, 68]), as the latter seem more adequate to develop analysis techniques that address properties such as conversation fidelity. Regarding session-based approaches, traditionally they only support binary interaction. Only recently have there been developments on session-based approaches so as to support multiparty interaction, namely the works of Bonelli et al. [11], Honda et al. [57] and Bettini et al. [9]. To support multiparty interaction, [57] considers multiple session channels, while [9] considers a multiple indexed session channel, both resorting to multiple communication pathways. We follow an essentially different approach, by letting a single medium of interaction support concurrent multiparty interaction via labeled messages.

In [9, 57] sessions are established simultaneously between several parties through a multicast session request. As in binary sessions, session delegation is full so the number of initial participants is kept invariant. On the other hand our conversations are initially established between two parties, however delegation is partial — in the sense the delegating partner does not lose access to the delegated conversation — so the number of parties in the conversation can grow as parties keep joining in. In such way, we may model multiparty conversations by the progressive access of multiple, dynamically determined partners, to ongoing conversations. We believe we can faithfully represent (up to some weak form of equivalence) the multicast session request by a conversation bootstrap that, through conversation join, establishes the conversation between the multiple parties in a first phase. On the other hand we do not see how our conversation join can be represented in the approaches of [9, 57]. A similar comparison can be made to the approach reported in [11], where not only dynamic joining of parties to a session is not supported (in the sense above), but also there is also a more centralized communication structure, as there is only one party that can communicate with (all the) others, which does not seem fit to address decentralized service-oriented settings.

Our approach, based on a novel notion of conversation context, and on simple and flexible message-passing communication, allows us to introduce the service initiation operations as mere

idioms of the language. This contrasts with other session-based proposals for service-oriented models — in fact, it contrasts with usual session-based models — where we find service instantiation operations primitive to the language. We thus end up with a more foundational model, where we allow for such operations to be represented by more general communication primitives. Since we based ourselves in existing proposals, when we initially introduced the Conversation Calculus [95], such operations were considered primitives of the language, and only later on we realized that they could actually be defined via the elementary communication primitives [27].

At a more fine grained level of comparison, we discuss some relation of our $\uparrow$ communication primitive and previously introduced similar ones. Our up ($\uparrow$) communication primitive was introduced with the aim of expressing the interaction between nested conversation contexts, in particular, between service instances and their callers, with loose-coupling in mind. Similar primitives were previously introduced in ambient calculi, namely Seal [36], Boxed Ambients [16] and BASS [47], and also Box π [87]. Our computation model is very different from those models (which are targeted at modeling migration and mobility), as witnessed by the normal form result (Proposition 4.4.14). Hence, even if formally related to some primitives introduced in [16, 36], at least when their reaction rules are considered in isolation, our communication primitives have very different consequences at the semantic level (for example, two $\uparrow$ messages can synchronize, just as long as they originate in subcontexts of the same context).

In a comparison to the π_{lab}-Calculus, we believe that a program specified in the Conversation Calculus is much more readable, mostly due to the fact that we can define — in a completely generic way — the service-oriented idioms, allowing us to abstract away from the manipulation of service conversation identities. The Conversation Calculus supports a generic definition of such idioms due to the fact that it allows for message protocols to be defined in the implicitly defined current (and enclosing) conversation, that may then be placed in the corresponding conversation access construct. So we may write a service definition **def** $s \Rightarrow P$ where P does not need to explicitly mention the identity of the conversation in which it will run in, while a π_{lab}-Calculus process necessarily mentions the conversation in each message exchange. This ability to specify protocols with respect to some implicitly defined conversation turns out to be important to the expressivity and simplicity of our analysis techniques, since the behaviors defined for the current conversation are invariant under name substitution, thus narrowing the points in the derivation where we must merge the behaviors. In the next chapter we show how the conversation type system profits from this notion of behavior defined in the current conversation.

Chapter 5

Typing Conversations

In this chapter we present the full-fledged conversation type system introduced in [27, 28], extending the initial presentation made in Chapter 3. The full type language supports the analysis of protocols that include alternative paths, allowing for parties to choose between a determined set of possibilities offered by other parties. Also, we enrich behavioral types with the notion of directed messages, which correspond directly to the directions of Conversation Calculus communications. In such way, we are able to capture, in a minimal way, the behavioral dependencies a process has between two (nested) conversations.

In the rest of the chapter we revisit some of the main ideas behind the conversation type system already introduced in Chapter 3, and motivate the extensions presented here. We then present the full conversation type language, and the associated operations that allow us to manipulate the type structure so as to capture the typing descriptions of CC systems. Then we present the type system itself, which associates conversation types to Conversation Calculus systems, and prove type safety results, namely Corollary 5.3.10 which says that well-typed systems are free from a certain kind of runtime errors, and enjoy the conversation fidelity property: all participants in a conversation follow well-defined protocols of interaction. To demonstrate the expressiveness of our technique we show that the higher-level service oriented idioms admit typing rules mechanically derived through the typing derivations for the low-level implementations, and we type a couple of realistic service collaborations, involving an a priori undetermined number of participants. We conclude the chapter with some remarks on related work.

5.1 Analysis of Dynamic Conversations

In a distributed computing setting it is crucial to have mechanisms to certify that distributed parties can communicate in a disciplined way: if there is no common understanding of a protocol of interaction then distributed parties are unlikely to be able to collaborate so as to carry out the tasks they are intended to. In the service-oriented computing setting in particular, the challenge of disciplining multiparty interaction is of central concern, as real service-oriented applications rely on the decentralized collaboration between several partners. The task of statically certifying that systems where parties dynamically call remote services follow well-defined protocols of interaction is particularly hard, since it involves statically "predicting" all possible runtime behaviors. The setting gets even more challenging when we consider that such protocols may follow alternative paths at runtime. Such challenging scenarios are the target of the conversation type system presented in this chapter.

The main ingredients of the type language were introduced in Chapter 3: a behavioral type system that combines, at the same level in the type language, local and global behavioral descriptions. A global behavioral description captures the overall protocol of interaction of a system — a *conversation*. A local behavioral description describes the role of an individual participant in a conversation. By mixing the two, we are also able to describe the role of a part of a system in a conversation, which is crucial to support the compositional analysis of the several parts of a system involved in a conversation. Our type language characterizes the message exchanges a process has with the external environment (output `!` and input `?` actions) that capture the local behavior of processes, and characterizes internal interactions (τ) which capture message exchanges between parties that are local to the system being characterized, and ultimately allow us to specify the global protocols of interaction. The polarity p (either `!`, `?` and τ) in a message type $(p\, l^d(C))$ characterizes processes that send `!`, receive `?` or internally exchange τ a message. Message types also specify the message label l, the direction d that specifies the conversation in which the message is to be exchanged (either in the current $\downarrow$ or in the enclosing $\uparrow$), and the argument type C of what is sent in the message.

For example, the following type (where we assume some basic value types Tp and Tm):

$$?\,\texttt{buy}^{\downarrow}(Tp).!\,\texttt{askPrice}^{\uparrow}(Tp).?\,\texttt{priceVal}^{\uparrow}(Tm).!\,\texttt{price}^{\downarrow}(Tm).\tau\,\texttt{product}^{\downarrow}(Tp).!\,\texttt{details}^{\downarrow}(Tp)$$

characterizes a process that receives a message `buy` in the current $\downarrow$ conversation, then sends message `askPrice` and receives message `priceVal` in the enclosing $\uparrow$ conversation, after which sends message `price`, internally exchanges message `product` and sends message `details` in the current conversation. Notice that output and input capabilities are mixed with internal message exchanges in this behavioral description. Notice also that our types do not mention the identity of the communicating peers. This increases the flexibility of our analysis, with respect to other approaches (e.g., [9, 57]) that consider communication capabilities are annotated with the identity of the party that is supposed to perform the action. In such way, we allow for more loosely-coupled specifications, as we do not enforce a particular party to perform the communication capability, instead we are only interested in verifying that *someone* exercises the communication capability.

A key notion in our approach to support multiparty interaction is that of a dynamic join: the dynamic call to a remote party that allows for it to start participating in an ongoing conversation. Multiparty conversations may be then formed by the progressive join of several parties to a given conversation. The ability to mix global and local specifications in conversation types is crucial to characterize such runtime delegations of behavior, since parties that delegate behavior are statically typed with their individual role in the conversation together with the communication capabilities that will be delegated away — the type of a subpart of the system. To explain such behavioral combinations our type system relies on a behavioral merge, which realizes behavioral composition by merging behavioral traces. For instance, by:

$$\begin{aligned}
&?\,\texttt{buy}^{\downarrow}(Tp).!\,\texttt{askPrice}^{\uparrow}(Tp).?\,\texttt{priceVal}^{\uparrow}(Tm).!\,\texttt{price}^{\downarrow}(Tm).\tau\,\texttt{product}^{\downarrow}(Tp).!\,\texttt{details}^{\downarrow}(Tp)\\
&=\\
&?\,\texttt{buy}^{\downarrow}(Tp).!\,\texttt{askPrice}^{\uparrow}(Tp).?\,\texttt{priceVal}^{\uparrow}(Tm).!\,\texttt{price}^{\downarrow}(Tm).!\,\texttt{product}^{\downarrow}(Tp)\\
&\bowtie\\
&?\,\texttt{product}^{\downarrow}(Tp).!\,\texttt{details}^{\downarrow}(Tp)
\end{aligned}$$

we specify that the type on top is a particular behavioral combination of the types on the bottom — the resulting type synchronizes message `product`. Our behavioral merge is then used to characterize conversations which have several distributed participants — by incrementally merging the individual behaviors — and to characterize systems where parties dynamically delegate conversation fragments.

Our type language includes message prefix $M.B$, which specifies that message M is to be sent, received, or internally exchanged, after which behavior B is activated, parallel composition $B_1 \mid B_2$ which allows us to describe concurrent behavior, and recursion that supports the specification of infinite behavior. The full conversation types language also features choice and branch types, $\oplus_{i\in I}\{M_i.B_i\}$ and $\&_{i\in I}\{M_i.B_i\}$, which characterize processes that can choose between one of the $M_i.B_i$ behaviors and branch in either of the $M_i.B_i$ branches, respectively. For example:

$$\oplus\{\,!\,\mathtt{requestApproved}^{\downarrow}().\tau\,\mathtt{orderTransfer}^{\downarrow}(\mathtt{T}_1,\mathtt{T}_2).?\,\mathtt{transferDate}^{\downarrow}(\mathtt{T}_4);\; !\,\mathtt{requestDenied}^{\downarrow}()\}$$

represents a process that can send either a `requestApproved` message or a `requestDenied` message. In the former case, the process proceeds by internally exchanging message `orderTransfer`, and after which receives message `transferDate`. On the other hand the following type:

$$\&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}().!\,\mathtt{approved}^{\uparrow}();\; ?\,\mathtt{requestDenied}^{\downarrow}().!\,\mathtt{denied}^{\uparrow}()\}$$

describes a process that is ready to receive both message `requestApproved` and `requestDenied`, where in the first case the process proceeds by sending messages `transferDate` and `approved` (the latter to the enclosing conversation), while in the second case the process proceeds by sending message `denied` (to the enclosing conversation). The merge of the choice and branch types above yields type:

$$\oplus\{\tau\,\mathtt{requestApproved}^{\downarrow}().\tau\,\mathtt{orderTransfer}^{\downarrow}(\mathtt{T}_1,\mathtt{T}_2).\tau\,\mathtt{transferDate}^{\downarrow}(\mathtt{T}_4).!\,\mathtt{approved}^{\uparrow}();\; \tau\,\mathtt{requestDenied}^{\downarrow}().!\,\mathtt{denied}^{\uparrow}()\}$$

which specifies an internal choice between the two actions — the composition of the process that chooses one of such behaviors with the process that branches in all such behaviors yields a process that internally decides which is the branch of behavior that will be activated. Notice that depending on the branch taken, distinct parties may be called in to collaborate in the ongoing conversation, as conversation fragments may be delegated away in the continuations of the message prefixes that form such choices and branches.

The behavioral types capture the protocols of interaction in a single conversation. Since processes, in general, may interact in several conversations, characterizing a CC system involves describing the several protocols the process has in each conversation. Then, a typing judgment of the form:

$$P :: B \mid n : [B_n] \mid m : [B_m] \mid o : [B_o] \mid \ldots$$

specifies that the behavior of process P in conversations n, m, ... is captured by behavioral types B_n, B_m, ..., respectively. Notice that type B appears unlocated in such judgment: since CC processes may interact in the current and enclosing conversations, which identity is not known at this point as it depends on the context where P is inserted in, the typing judgment must consider

$$
\begin{array}{lcll}
n, m, o, \ldots & \in & \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
l, s \ldots & \in & \mathcal{L} & \text{(Labels)} \\
\mathcal{X}, \mathcal{Y}, \ldots & \in & \chi & \text{(Recursion Variables)}
\end{array}
$$

$$
\begin{array}{lcll}
B & ::= & B_1 \mid B_2 \;\big|\; \mathbf{0} \;\big|\; \mathtt{rec}\,\mathcal{X}.B \;\big|\; \mathcal{X} \;\big|\; \oplus_{i \in I}\{M_i.B_i\} \;\big|\; \&_{i \in I}\{M_i.B_i\} & \text{(Behavioral)} \\
M & ::= & p\, l^d(C) & \text{(Message)} \\
p & ::= & \mathtt{!} \;\big|\; \mathtt{?} \;\big|\; \tau & \text{(Polarity)} \\
C & ::= & [B] & \text{(Conversation)} \\
L & ::= & n : C \;\big|\; L_1 \mid L_2 \;\big|\; \mathbf{0} & \text{(Located)} \\
T & ::= & L \mid B & \text{(Process)}
\end{array}
$$

Figure 5.1: Conversation Types Syntax.

an unlocated behavioral type. Notice also that, in general, behavioral dependencies between conversations are not captured by conversation types. However, it is possible to represent behavioral dependencies between the current conversation and the enclosing conversation in the unlocated part of the type, since it may mix $\downarrow$ behaviors with $\uparrow$ behaviors.

We denote by T a process type that mixes an unlocated behavioral type with some located behavioral types. Then a typing judgement $P :: T$, which states that P is well-typed with type T, intuitively says that if process P is placed in a context where a process that behaves like T is expected then we obtain a safe system. The intended safety property will be formally stated in Corollary 5.3.10: it implies conversations agree to declared protocols, and the absence of certain kind of runtime errors. In the next sections we present the type language and the type system that allows us to single out type-safe CC systems.

5.2 Type Language

In this section we formally present the full conversation types language. As already motivated in the Introduction, our types specify the message protocols that flow between and within conversations. The syntax of the type language closely follows the one presented in Chapter 3, featuring behavioral types to describe the conversation protocols, and located types that associate behavioral types to their respective conversations. We define the syntax of the language.

Definition 5.2.1 (Conversation Type Language)
The syntax of the conversation type language is given in Figure 5.1.

With respect to the core type language presented in Chapter 3 there are three differences. (1) The full type language features choice and branch types that characterize processes that can choose between one of the $M_i.B_i$ behaviors and branch in either of the $M_i.B_i$ behaviors, respectively. (2) Message types are defined with a direction d, accordingly to Conversation Calculus messages, thus, message types refer to which conversation they pertain — either the current $\downarrow$ or the enclosing $\uparrow$ conversation. (3) We introduce process types that are composed by a located type and a behavioral type. Since processes may interact in several conversations, process types describe the interactions in all such conversations: on the one hand if the identity

of the conversation in which a process interacts is already known then the process type will describe such interactions in a located type; on the other hand, the process type describes the interactions of the process in the current and enclosing conversations (which identity is not yet known) in a behavioral type. Thus, typing judgments have the form $P :: T$, where T is a process type given by $L \mid B$, for some located type L and behavioral type B.

Our behavioral types include parallel composition ($B_1 \mid B_2$) to represent concurrent behavior, the inactive behavior $\mathbf{0}$, recursive behavior through the combined use of the recursive definition construct $\mathtt{rec}\,\mathcal{X}.B$ and the recursion variable $\mathcal{X}$, and also choice and branch types that capture alternative behavior: the former characterizes processes that can perform one of the $M_i.B_i$ choices, and the latter characterizes processes that can perform either one of the $M_i.B_i$ branches.

The prefix $M.B$ specifies a process that sends, receives, or internally exchanges a message M before proceeding with behavior B. Message types specify the message polarity (either output $\mathtt{!}$, input $\mathtt{?}$ and internal exchange τ), the directed label of the message l^d and the argument type C of what is communicated in the message. The argument conversation type describes the behavior that the process receiving the message will have in the conversation which identifier is passed in the message. Notice that message types specify both communications of a process with the external environment (inputs $\mathtt{?}$ and outputs $\mathtt{!}$) and internal message exchanges (τ).

Conversation types C consist of a delimited behavior. Then, located types L collect (using composition $L_1 \mid L_2$) type associations between conversation names and their types ($n : C$).

We introduce some convenient abbreviations and conventions for behavioral types.

Syntactic conventions:

$M.B_1 \mid B_2$ is to be read as $(M.B_1) \mid B_2$.

$\mathtt{rec}\,\mathcal{X}.B_1 \mid B_2$ is to be read as $(\mathtt{rec}\,\mathcal{X}.B_1) \mid B_2$.

Abbreviations:

$p\,l(C)$ stands for $p\,l^{\downarrow}(C)$.

M stands for $M.\mathbf{0}$.

$\mathtt{?}\,l(C).B$ stands for $\&\{\mathtt{?}\,l(C).B\}$.

$\mathtt{!}\,l(C).B$ stands for $\oplus\{\mathtt{!}\,l(C).B\}$.

$\tau\,l(C).B$ stands for $\oplus\{\tau\,l(C).B\}$.

$\oplus\{B_1; \ldots; B_k\}$ and $\oplus\{\tilde{B}\}$ stand for $\oplus_{i \in 1,\ldots,k}\{B_i\}$.

$\&\{B_1; \ldots; B_k\}$ and $\&\{\tilde{B}\}$ stand for $\&_{i \in 1,\ldots,k}\{B_i\}$.

$\{\tilde{B}\}$ stands for $\&\{\tilde{B}\}$ and for $\oplus\{\tilde{B}\}$.

$\star M$ stands for $\mathtt{rec}\,\mathcal{X}.M.\mathcal{X}$.

In the following sections we present the operations that manipulate the type structure so as to support the modifications in the types required to characterize Conversation Calculus processes. Most of them were already introduced in Chapter 3 but we now adapt them to the current

setting. We start by an operation which is specific to the full conversation type language: projecting a type in a determined direction. Then we introduce the types that characterize our intended recursive behaviors. After that we present the apartness predicate that allows us to identify independent behavior, adapting the previously introduced auxiliary operators to the full conversation type language. Then we present the relation between types — the subtyping relation — which is extended with some axioms specific to the current setting. Finally, we present the merge relation which we use to behaviorally combine types, which is a slight extension of the one presented in Chapter 3.

5.2.1 Projecting Types

When characterizing CC process behavior we need to project the behavioral descriptions provided by types along a direction d (either $\uparrow$ and $\downarrow$), in particular when we locate such behaviors in their respective conversations. The projection $d(B)$ in the direction d of a behavioral type B consists in the selection of all messages that have the given direction d while filtering out the ones in the other direction, offering a partial view of behavior B from the viewpoint of d.

Definition 5.2.2 (Direction Projection)
For each direction d, the projection $d(B)$ of type B along direction d is defined as follows:

$$
\begin{array}{lll}
d(\mathbf{0}) & \triangleq & \mathbf{0} \\
d(\mathcal{X}) & \triangleq & \mathcal{X} \\
d(\mathtt{rec}\,\mathcal{X}.B) & \triangleq & \mathtt{rec}\,\mathcal{X}.B \qquad (\textit{if}\, d(B) = B) \\
d(B_1 \mid B_2) & \triangleq & d(B_1) \mid d(B_2) \\
d(!\, l^{d'}(C).B) & \triangleq & d(B) \qquad (\textit{if}\, d \neq d') \\
d(\oplus_{i\in I}\{p_i\, l_i^d(C_i).B_i\}) & \triangleq & \oplus_{i\in I}\{p_i\, l_i^d(C_i).d(B_i)\} \\
d(\oplus_{i\in I}\{!\, l_i^{d'}(C_i).B_i\}) & \triangleq & \oplus_{i\in I}\{d(B_i)\} \qquad (\textit{if}\, d \neq d' \textit{ and}\, d(B_i) = \,!\, l^d(C).B) \\
d(?\, l^{d'}(C).B) & \triangleq & d(B) \qquad (\textit{if}\, d \neq d') \\
d(\&_{i\in I}\{?\, l_i^d(C_i).B_i\}) & \triangleq & \&_{i\in I}\{?\, l_i^d(C_i).d(B_i)\} \\
d(\&_{i\in I}\{?\, l_i^{d'}(C_i).B_i\}) & \triangleq & \oplus_{i\in I}\{d(B_i)\} \qquad (\textit{if}\, d \neq d' \textit{ and}\, d(B_i) = \,!\, l^d(C).B)
\end{array}
$$

Notation 5.2.3 *To lighten notation we write $\uparrow B$ instead of $\uparrow(B)$ and $\downarrow B$ instead of $\downarrow(B)$.*

Notice that the direction projection is a partial operation and some types are excluded at this level (e.g., branch/choice types where the initial actions have different directions). Informally, we sometimes refer to $\downarrow B$ as the "here interface" of B, and likewise for $\uparrow B$ as the "up interface". We illustrate the projection of choice and branch types with an example.

Example 5.2.4 *Consider type:*

$$
\begin{array}{l}
\&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}().!\,\mathtt{approved}^{\uparrow}(); \\
\quad ?\,\mathtt{requestDenied}^{\downarrow}().!\,\mathtt{denied}^{\uparrow}()\}
\end{array}
$$

which specifies a process that can receive either a `requestApproved` *message or a* `requestDenied` *message in the current conversation, and eventually sends either message* `approved` *or message* `denied` *in the enclosing conversation. Projecting the type in the* $\downarrow$ *direction results in the branch type of the two* $\downarrow$ *messages (and respective* $\downarrow$ *continuations), as follows:*

$$\begin{array}{l} \downarrow(\&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}().!\,\mathtt{approved}^{\uparrow}();\\ \quad ?\,\mathtt{requestDenied}^{\downarrow}().!\,\mathtt{denied}^{\uparrow}()\})\\ \qquad\qquad = \\ \qquad\qquad \&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}();\\ \qquad\qquad\quad ?\,\mathtt{requestDenied}^{\downarrow}()\} \end{array}$$

On the other hand, the $\uparrow$ *projection specifies that the process chooses between one of the* $\uparrow$ *behaviors of the continuations (since the branching is invisible from this view), as follows:*

$$\begin{array}{l} \uparrow(\&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}().!\,\mathtt{approved}^{\uparrow}();\\ \quad ?\,\mathtt{requestDenied}^{\downarrow}().!\,\mathtt{denied}^{\uparrow}()\})\\ \qquad\qquad = \\ \qquad\qquad \oplus\{!\,\mathtt{approved}^{\uparrow}();\ !\,\mathtt{denied}^{\uparrow}()\} \end{array}$$

We introduce an operator that changes the direction of all messages to $\downarrow$ in a behavioral type.

Definition 5.2.5 (Locating a Behavioral Type)
Locating a behavioral type B *in the current conversation, noted* $loc(B)$*, is defined as follows:*

$$\begin{array}{llllll} loc(B_1 \mid B_2) & \triangleq & loc(B_1) \mid loc(B_2) & loc(\mathbf{0}) & \triangleq & \mathbf{0} \\ loc(\mathtt{rec}\,\mathcal{X}.B) & \triangleq & \mathtt{rec}\,\mathcal{X}.loc(B) & loc(\mathcal{X}) & \triangleq & \mathcal{X} \\ loc(\oplus_{i\in I}\{M_i.B_i\}) & \triangleq & \oplus_{i\in I}\{loc(M_i.B_i)\} & loc(\&_{i\in I}\{M_i.B_i\}) & \triangleq & \&_{i\in I}\{loc(M_i.B_i)\} \\ loc(p\,l^d(C).B) & \triangleq & p\,l^{\downarrow}(C).loc(B) & & & \end{array}$$

The $loc(B)$ operation is used when we locate behaviors in their respective conversations, more specifically, when we locate in the current conversation the $\uparrow$ interactions of a process located in a nested conversation — we then use $loc()$ to redirect $\uparrow$ to $\downarrow$ messages.

5.2.2 Typing Recursive Behavior

We reinstate our distinction, for typing purposes, between shared $\mathcal{L}_\star$ and plain $\mathcal{L}_p$ labels.

Definition 5.2.6 (Plain and Shared Labels)
We denote by $\mathcal{L}_\star$ *and* $\mathcal{L}_p$ *subsets of* $\mathcal{L}$ *such that* $\mathcal{L}_\star \cap \mathcal{L}_p = \emptyset$ *and* $\mathcal{L}_\star \cup \mathcal{L}_p = \mathcal{L}$.

Messages that are to be used exponentially, typically messages regarding service instantiation, are defined on shared labels, while messages that are to be used linearly, supporting linear protocol specifications in a service interaction, are defined on plain labels. We can then characterize the interface of a persistent service definition: it consists in a number of output messages defined on shared labels, representing calls to auxiliary services. We define exponential output types.

Definition 5.2.7 (Exponential Output Type)
Exponential output message types, noted $\star M^!$*, exponential output behavioral types, noted* $\star B^!$*, and exponential output located types, noted* $\star L^!$*, are defined as follows:*

$$
\begin{array}{llll}
\star M^! & ::= & \star M \qquad (M = \,!\, l^d(C) \text{ and } l \in \mathcal{L}_\star) & \text{(Exponential Output Message Type)} \\
\star B^! & ::= & \star M^! \;\mid\; \star B_1^! \mid \star B_2^! \;\mid\; \mathbf{0} & \text{(Exponential Output Behavioral Type)} \\
\star L^! & ::= & n : [\star B^!] \;\mid\; \star L_1^! \mid \star L_2^! \;\mid\; \mathbf{0} & \text{(Exponential Output Located Type)}
\end{array}
$$

Since several copies of a persistent service definition may be concurrently active, we only allow for them to share an exponential output type, allowing for such messages that represent service calls to compete (race) for the resource — the service definition. Instead, races on linear messages are excluded by our type system. However, recursive processes may define linear message exchanges, just as long as they appear in the type prefixing the recursion variable. In such way, we are sure that such messages will never compete, as only after they have been exchanged will the recursive behavior repeat itself. We thus define $B\langle\mathcal{X}\rangle$ that represents a safe recursive behavior, where the recursion variable $\mathcal{X}$ may occur as a leaf, and all its plain labels appear in messages that prefix the recursion variable.

Definition 5.2.8 (Recursive Types)
Leaf recursive types, noted $B^l\langle\mathcal{X}\rangle$*, and recursive behavioral types, noted* $B\langle\mathcal{X}\rangle$*, are defined as:*

$$
\begin{array}{lll}
B^l\langle\mathcal{X}\rangle & ::= & \mathbf{0} \;\mid\; \mathcal{X} \;\mid\; \oplus_{i\in I}\{M_i.B_i^l\langle\mathcal{X}\rangle\} \;\mid\; \&_{i\in I}\{M_i.B_i^l\langle\mathcal{X}\rangle\} \;\mid\; B^l\langle\mathcal{X}\rangle \mid \star B^! \quad (\textit{if } B^l\langle\mathcal{X}\rangle \,\#\, \star B^!) \\
B\langle\mathcal{X}\rangle & ::= & \mathbf{0} \;\mid\; \oplus_{i\in I}\{M_i.B_i^l\langle\mathcal{X}\rangle\} \;\mid\; \&_{i\in I}\{M_i.B_i^l\langle\mathcal{X}\rangle\} \;\mid\; B\langle\mathcal{X}\rangle \mid \star B^! \quad (\textit{if } B\langle\mathcal{X}\rangle \,\#\, \star B^!)
\end{array}
$$

Type $B\langle\mathcal{X}\rangle$ thus characterizes recursive processes that can safely have several active concurrent instances, where by safely we intend that the concurrent instances share only some exponential behavior, since the linear behavior will be sequentially activated. Notice that some of the messages that are to be sequentially activated may be defined on shared labels, so not all messages prefixing the recursion variable have to be linear. Notice also that the recursion variable may occur as a leaf in all branches and choices of branch and choice types.

Such types guarantee, by definition, there is no undesired interference in between the several runs of a recursive process. To identify, in general, when two types characterize systems that do not interfere in an undesired way, we introduce the apartness predicate.

5.2.3 Apartness

A crucial operation in our type system is type apartness: systems characterized by types that are apart are ensured not to exhibit undesired interferences. To define this notion we must distinguish between behavior in shared messages and behavior in plain messages, as the intended notion of interference varies between the two cases. Plain messages are to be used linearly, thus systems that exhibit the same behavior on some plain messages can interfere — the messages can race — and thus such systems are not apart. On the other hand, systems that exhibit the same behavior on shared messages do not interfere, just as long such messages are consistently used — if they are defined on the same argument type. Although systems that may synchronize, in either linear or shared messages, are safe systems, they are not independent and hence are not

apart — such synchronizations will be explained by the behavioral merge (Definition 5.2.18).

We introduce some auxiliary operations in order to define type apartness. We define the set of message types $Msg_{\mathcal{L}}(B)$ and the set of directed message labels $dLab_{\mathcal{L}}(B)$ of a behavioral type B. Both sets are indexed by a label set $\mathcal{L}$, which can be refined to either $\mathcal{L}_\star$ and $\mathcal{L}_p$ so as to allow, e.g., the distinction of the message sets with labels from shared labels (from $\mathcal{L}_\star$) and those from plain labels (from $\mathcal{L}_p$).

Definition 5.2.9 (Message Set)
We denote by $Msg_{\mathcal{L}}(B)$ the set of message types defined with labels in $\mathcal{L}$ of a behavioral type B, defined as follows:

$$\begin{array}{lcl}
Msg_{\mathcal{L}}(\mathbf{0}) & \triangleq & \emptyset \\
Msg_{\mathcal{L}}(B_1 \mid B_2) & \triangleq & Msg_{\mathcal{L}}(B_1) \cup Msg_{\mathcal{L}}(B_2) \\
Msg_{\mathcal{L}}(\mathcal{X}) & \triangleq & \emptyset \\
Msg_{\mathcal{L}}(\mathtt{rec}\,\mathcal{X}.B) & \triangleq & Msg_{\mathcal{L}}(B) \\
Msg_{\mathcal{L}}(p\,l^d(C).B) & \triangleq & \{(p\,l^d(C)) \mid l \in \mathcal{L}\} \cup Msg_{\mathcal{L}}(B) \\
Msg_{\mathcal{L}}(\oplus_{i\in I}\{M_i.B_i\}) & \triangleq & \bigcup_{i\in I} Msg_{\mathcal{L}}(M_i.B_i) \\
Msg_{\mathcal{L}}(\&_{i\in I}\{M_i.B_i\}) & \triangleq & \bigcup_{i\in I} Msg_{\mathcal{L}}(M_i.B_i)
\end{array}$$

For example, given some behavioral type B, $Msg_{\mathcal{L}_p}(B)$ is the set of all plain (in $\mathcal{L}_p$) message types $(p\,l^d(C))$ occurring in B, leaving out message types defined on shared labels (those belonging to $\mathcal{L}_\star$). Using the message set, we define the set of directed labels of a behavioral type.

Definition 5.2.10 (Set of Directed Labels)
We denote by $dLab_{\mathcal{L}}(B)$ the set of directed labels from $\mathcal{L}$ of a behavioral type B, defined as:

$$dLab_{\mathcal{L}}(T) \triangleq \{l^d \mid (p\,l^d(C)) \in Msg_{\mathcal{L}}(B)\}$$

Using the set of directed labels information we may already characterize when two types are apart with respect to plain messages: their plain directed label sets must be disjoint. However, to determine apartness with respect to shared messages we require another operation: conformance.

Definition 5.2.11 (Conformance)
We say two behavioral types B_1, B_2 are conformant, noted $B_1 \asymp B_2$, if for any two message types $p_1\,l^d(C_1)$ and $p_2\,l^d(C_2)$ such that:

$$(p_1\,l^d(C_1)) \in Msg_{\mathcal{L}_\star}(B_1) \qquad \text{and} \qquad (p_2\,l^d(C_2)) \in Msg_{\mathcal{L}_\star}(B_2)$$

then $C_1 = C_2$ and if $p_i = \mathtt{?}$ then $p_j = \tau$ for $\{i,j\} = \{1,2\}$.

Two behavioral types are compatible on shared messages if they specify messages defined on shared labels with identical argument types and also if they are defined on determined polarities. For instance, two message types defined on shared labels and polarity `!` are conformant as they represent compatible calls to the same service. We exclude the cases of messages presenting dual polarities (`!` and `?`) and when both messages present `?` polarities: the former will later

be used to force such messages to synchronize, which means a τ will be introduced in the type to represent the possible synchronization; the latter is used to check the compatibility of two service definitions — the combination of such types is explained by the behavioral merge (Definition 5.2.18). Using the label set and conformance, we may define type apartness.

Definition 5.2.12 (Apartness)
We say two behavioral types B_1, B_2 are apart, noted $B_1 \# B_2$, if their sets of plain directed labels are disjoint ($dLab_{\mathcal{L}_p}(B_1) \# dLab_{\mathcal{L}_p}(B_2)$) and they are conformant ($B_1 \asymp B_2$).

Essentially, apartness ensures disjointness of plain ("linear") types, and consistency of shared ("exponential") types (cf. [64]). The notion of independent behavior provided by apartness is used by the relation between types — the subtyping relation — to identify the cases when it is possible to manipulate the type structure so as to allow more flexible behavioral characterizations. In the following section we present the subtyping relation.

5.2.4 Subtyping

Types are related by the subtyping relation we now present. Intuitively, we say type T_1 is a subtype of type T_2, noted $T_1 <: T_2$, when a process of type T_1 can safely be used in a context where a process of type T_2 is expected. Subtyping provides a way to generalize the typing characterization of processes, by its use in a subsumption rule of the form:

$$\frac{P :: T_1 \quad T_1 <: T_2}{P :: T_2}$$

which then allows to characterize a process by a more general type (the supertype) given the characterization of the process by a more specific type (the subtype).

We start by informally discussing the key rules and then present the formal definition. Our subtyping rules express expected structural relationships of types, namely that $(- \mid -, \mathbf{0})$ is a commutative monoid with respect to subtyping. We also specify that branches and choices can be freely commuted in branch and choice types, respectively:

$$\oplus\{\tilde{B}; \tilde{B}'\} \equiv \oplus\{\tilde{B}'; \tilde{B}\} \qquad \&\{\tilde{B}; \tilde{B}'\} \equiv \&\{\tilde{B}'; \tilde{B}\}$$

where $T_1 \equiv T_2$ abbreviates $T_1 <: T_2$ and $T_2 <: T_1$. We adopt an equi-recursive approach to recursive types [83], based on simple unfoldings of recursive type terms:

$$\texttt{rec}\,\mathcal{X}.B \equiv B\{\mathcal{X}/\texttt{rec}\,\mathcal{X}.B\}$$

We also have a split rule:

$$n : [B_1 \mid B_2] \equiv n : [B_1] \mid n : [B_2]$$

which captures the notion that the behavior in a single conversation can be described through distinct pieces. We embed the subtyping relation with reflexivity and with transitivity. We also

specify the congruence closure of the relation in all contexts of the type language:

$$\frac{T_1 <: T_2}{T_3 \mid T_1 <: T_3 \mid T_2} \qquad \frac{B_1 <: B_2}{n : [B_1] <: n : [B_2]} \qquad \frac{B_1 <: B_2}{\texttt{rec}\,\mathcal{X}.B_1 <: \texttt{rec}\,\mathcal{X}.B_2}$$

$$\frac{\forall_{i \in I}\ B_i <: B'_i}{\oplus_{i \in I}\{M_i.B_i\} <: \oplus_{i \in I}\{M_i.B'_i\}} \qquad \frac{\forall_{i \in I}\ B_i <: B'_i}{\&_{i \in I}\{M_i.B_i\} <: \&_{i \in I}\{M'_i.B'_i\}}$$

The following rules allow us to generalize our typing characterizations. A key subtyping rule that introduces some flexibility at the level of protocol specification is the following:

$$M.(B_1 \mid B_2) <: M.B_1 \mid B_2 \ \ (M \# B_2, \mathit{fv}(B_2) = \emptyset)$$

which then allows for sequential protocols to export a more general concurrent interface, provided the behaviors specified in parallel are apart $M \# B_2$. The intuition is that if a process performs action M and after which exhibits behavior B_2 then it can safely be used in a context that expects a process that exhibits simultaneously action M and behavior B_2.

The next rule expresses a contraction principle for exponential output message types $\star M^!$:

$$\star M^! \mid \star M^! <: \star M^!$$

which describes that a process that sends two (each infinitely often) messages ($\star M^!$) can be safely used in a context where a process that sends such message just once (infinitely often) is expected. Also regarding exponential output types we introduce a weakening principle by rule:

$$M <: \star M^! \ \ (M = \,!\, l^d(C), l \in \mathcal{L}_\star)$$

which specifies that a process which outputs once a message defined on a shared label can be used in a context where a process that outputs the message an unbounded number of times is expected. This allows us to always expect an exponential output type which already specifies an unbounded number of usages, for instance, in the premises of typing rules, and also in the following rule where we allow recursive types to export such exponential interface separately:

$$\texttt{rec}\,\mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\mathcal{X}\rangle) <: \star M_1^! \mid \ldots \mid \star M_k^! \mid \texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle$$

The rule then allows for a process that specifies a number of exponential output messages in between its repeated executions to be characterized by the type that separately specifies the exponential interface and the recursive behavior.

The following rule expresses a crucial subtyping principle, where we allow a behavioral type to be decomposed (in the supertype) in its two projections according to the message directions:

$$B <: \downarrow B \mid \uparrow B$$

The rule characterizes that a process that specifies some, possibly interleaved, behavior in the current and enclosing conversations, can safely be used in a context where a process that exhibits such behaviors independently is expected. We define subtyping for process types.

$$T_1 \mid T_2 \equiv T_2 \mid T_1 \;\; (\mathit{SubParComm}) \qquad T_1 \mid (T_2 \mid T_3) \equiv (T_1 \mid T_2) \mid T_3 \;\; (\mathit{SubParAssoc})$$

$$T \mid \mathbf{0} \equiv T \;\; (\mathit{SubParZero}) \qquad \mathtt{rec}\, \mathcal{X}.B \equiv B\{\mathcal{X}/\mathtt{rec}\, \mathcal{X}.B\} \;\; (\mathit{SubRecUnfold})$$

$$\oplus\{\tilde{B}; \tilde{B}'\} \equiv \oplus\{\tilde{B}'; \tilde{B}\} \;\; (\mathit{SubChoiceComm}) \qquad \&\{\tilde{B}; \tilde{B}'\} \equiv \&\{\tilde{B}'; \tilde{B}\} \;\; (\mathit{SubBranchComm})$$

$$n : [B_1 \mid B_2] \equiv n : [B_1] \;\mid n : [B_2] \;\; (\mathit{SubLocSplit})$$

$$T <: T \;\; (\mathit{SubReflex}) \qquad \frac{T_1 <: T_3 \quad T_3 <: T_2}{T_1 <: T_2}(\mathit{SubTrans})$$

$$\frac{T_1 <: T_2}{T_3 \mid T_1 <: T_3 \mid T_2}(\mathit{SubPar}) \quad \frac{B_1 <: B_2}{n : [B_1] <: n : [B_2]}(\mathit{SubProc}) \quad \frac{B_1 <: B_2}{\mathtt{rec}\, \mathcal{X}.B_1 <: \mathtt{rec}\, \mathcal{X}.B_2}(\mathit{SubRec})$$

$$\frac{\forall_{i \in I}\; B_i <: B_i'}{\&_{i \in I}\{M_i.B_i\} <: \&_{i \in I}\{M_i.B_i'\}}(\mathit{SubBranch}) \qquad \frac{\forall_{i \in I}\; B_i <: B_i'}{\oplus_{i \in I}\{M_i.B_i\} <: \oplus_{i \in I}\{M_i.B_i'\}}(\mathit{SubChoice})$$

$$\mathtt{rec}\; \mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle \mathcal{X} \rangle) <: \star M_1^! \mid \ldots \mid \star M_k^! \mid \mathtt{rec}\; \mathcal{X}.B\langle \mathcal{X} \rangle \;\; (\mathit{SubExpRec})$$

$$\star M^! \mid \star M^! <: \star M^! \;\; (\mathit{SubExpFold}) \qquad B <: \downarrow B \mid \uparrow B \;\; (\mathit{fv}(B) = \emptyset) \;\; (\mathit{SubProject})$$

$$M.(B_1 \mid B_2) <: M.B_1 \mid B_2 \;\; (M \;\#\; B_2, \mathit{fv}(B_2) = \emptyset) \;\; (\mathit{SubParPref})$$

$$M <: \star M^! \;\; (M = \,!\, l^d(C), l \in \mathcal{L}_\star) \;\; (\mathit{SubExpWeak})$$

Figure 5.2: Process Types Subtyping Rules.

Definition 5.2.13 (Process Types Subtyping and Equivalence)
We say process type T_1 is a subtype of process type T_2, noted $T_1 <: T_2$, if $T_1 <: T_2$ is derivable by the rules shown in Figure 5.2. Also, we say T_1 and T_2 are equivalent, noted $T_1 \equiv T_2$, if $T_1 <: T_2$ and $T_2 <: T_1$.

Next we present the operation used to behaviorally combine types, that thus explains the behavioral composition of processes.

5.2.5 Merge Relation

Our typing characterizations crucially depend on the operation that behaviorally combines types: the merge relation. The merge relation allows us to characterize the behavior of composite processes, by merging the individual behaviors of the several subprocesses in a compositional way. We thus write $B = B_1 \bowtie B_2$ to say that B is a particular (in general not unique) behavioral combination of the types B_1 and B_2. The merge of two independent — apart — types yields the independent composition of the two types. However, when the types specify behaviors that may synchronize, then the merge relation introduces an internal message exchange τ in the type to represent such synchronization potential. Thus, the merge of two behaviors is defined not only in terms of spatial separation, but also, and crucially, in terms of merging behavioral "traces". Notice that it is not always the case that there is B such that $B = B_1 \bowtie B_2$. On the other hand, if some such B exists, we use $B_1 \bowtie B_2$ to non-deterministically denote any such B.

Notation 5.2.14 *We use $B_1 \bowtie B_2$ to represent B such that $B = B_1 \bowtie B_2$.*

Intuitively, $B = B_1 \bowtie B_2$ holds if B_1 and B_2 may safely synchronize or interleave so as to produce behavioral type B. So, the merge of two types is defined when either the types are apart, or when the non-apart messages may be synchronized, otherwise the merge is undefined.

Before defining the merge relation we introduce some auxiliary operations: one that collects the set of directed labels of actions immediately active in a behavioral type B, noted $\mathcal{I}(B)$, and another that replaces all occurrences of a message type by another, noted $B\{M_1/M_2\}$.

Definition 5.2.15 (Set of Initial Directed Labels)
The set of initial directed labels of a behavioral type B, noted $\mathcal{I}(B)$, is defined as follows:

$$
\begin{array}{lll@{\qquad}lll}
\mathcal{I}(\mathbf{0}) & \triangleq & \emptyset & \mathcal{I}(B_1 \mid B_2) & \triangleq & \mathcal{I}(B_1) \cup \mathcal{I}(B_2) \\
\mathcal{I}(\mathcal{X}) & \triangleq & \emptyset & \mathcal{I}(\mathtt{rec}\,\mathcal{X}.B) & \triangleq & \mathcal{I}(B) \\
\mathcal{I}(\oplus_{i\in I}\{M_i.B_i\}) & \triangleq & \bigcup_{i\in I}\mathcal{I}(M_i.B_i) & \mathcal{I}(\&_{i\in I}\{M_i.B_i\}) & \triangleq & \bigcup_{i\in I}\mathcal{I}(M_i.B_i) \\
\mathcal{I}(p\, l^d(C).B) & \triangleq & \{l^d\} & & &
\end{array}
$$

Definition 5.2.16 (Message Type Substitution)
We denote by $B\{M_1/M_2\}$ the type obtained by replacing all occurrences of message type M_1 with message type M_2 in type B, defined inductively in the structure of types as follows:

$$
\begin{array}{lll}
\mathbf{0}\{M_1/M_2\} & \triangleq & \mathbf{0} \\
(B_1 \mid B_2)\{M_1/M_2\} & \triangleq & (B_1\{M_1/M_2\}) \mid (B_2\{M_1/M_2\}) \\
\mathcal{X}\{M_1/M_2\} & \triangleq & \mathcal{X} \\
(\mathtt{rec}\,\mathcal{X}.B)\{M_1/M_2\} & \triangleq & \mathtt{rec}\,\mathcal{X}.(B\{M_1/M_2\}) \\
(M_1.B)\{M_1/M_2\} & \triangleq & M_2.(B\{M_1/M_2\}) \\
(M.B)\{M_1/M_2\} & \triangleq & M.(B\{M_1/M_2\}) \qquad (\textit{if } M \neq M_1) \\
(\oplus_{i\in I}\{M_i.B_i\})\{M/M'\} & \triangleq & \oplus_{i\in I}\{(M_i.B_i)\{M/M'\}\} \\
(\&_{i\in I}\{M_i.B_i\})\{M/M'\} & \triangleq & \&_{i\in I}\{(M_i.B_i)\{M/M'\}\}
\end{array}
$$

Given these basic operations we may now present the merge relation, starting by an informal description of the key rules. There are two key rules that explain type synchronizations, one for messages defined on plain labels, and the other for messages defined on shared labels. For plain messages synchronization we have the following rule:

$$
\frac{\forall_{i\in I}.(B_i = B_i^- \bowtie B_i^+ \qquad l_i \in \mathcal{L}_p)}{\oplus_{i\in I}\{\tau\, l_i^{\downarrow}(C_i).B_i\} = \&_{i\in I}\{?\, l_i^{\downarrow}(C_i).B_i^-\} \bowtie \oplus_{i\in I}\{!\, l_i^{\downarrow}(C_i).B_i^+\}}(\mathit{Plain})
$$

that merges dual branch and choice types in an internal choice — a choice between a set of internal (τ) message exchanges. The continuations of the internal actions are given by the merge of the respective continuations of the branches and choices. Merge $\bowtie$ thus allows $\tau\, l^{\downarrow}$ plain message types ("here" internal interactions) to be separated into send ! and receive ? capabilities in respective choice and branch constructs.

Shared message synchronization is captured by rule:

$$\frac{B \asymp \,!\, l^d(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l^d(C)/\tau\ l^d(C)\} \mid \star\,?\, l^d(C) = B \bowtie \star ?\, l^d(C)}(\mathit{Shared})$$

which synchronizes a persistently available input message type with all corresponding output message types. The resulting merge is then the type obtained replacing all $!\ l^d(C)$ message types with $\tau\, l^d(C)$ in B, in parallel with the persistent input message type: this allows for shared labeled inputs to synchronize and still expect further outputs from the external environment, leaving open the possibility for further synchronizations. A τ shared message type thus represents that there is (at least one) persistently available input that matches the output, while a τ plain message type characterizes the uniquely determined synchronization on that plain label.

The following rule ensures compatibility of persistent shared input specifications:

$$\frac{l \in \mathcal{L}_\star}{\star ?\, l^d(C) = \star ?\, l^d(C) \bowtie \star ?\, l^d(C)}(\mathit{SharedInp})$$

Thus, two persistent shared inputs may be merged if they are characterized by exactly the same type. The following rule allows for the merge to interleave a message prefix:

$$\frac{M \# B_2 \qquad B' \mid B'' \equiv B_1 \bowtie B_2 \qquad M \# B'' \qquad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \qquad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2)}{M.B' \mid B'' = M.B_1 \bowtie B_2}(\mathit{Shuffle})$$

Rule (*Shuffle*) explains the composition of behaviors $M.B_1$ and B_2 by first composing B_1 and B_2 (since M is apart from B_2 it does not interfere with B_2) and second by placing the message prefix $M.B'$ so as to maintain (some of) the sequentiality information originally specified in $M.B_1$. On the one hand, such merge does not allow for M to prefix behaviors it did not originally prefix ($\mathcal{I}(B') \subseteq \mathcal{I}(B)$). On the other hand the behavior B'' which is specified in parallel to M has its initial actions defined by a subset of the ones specified by B_2, so no behaviors that occurred only in the continuation of M will be exposed in parallel. In such way, we allow for type synchronizations to occur in the continuation of message prefixes. Notice also we only allow shuffling on simple message prefixes ($M.B$), and not in choice and branch types, with respect to which we show the following motivating example.

Example 5.2.17 *Consider the following specification:*

$$\&\{?\,\texttt{requestApproved}^{\downarrow}().!\,\texttt{transferDate}^{\downarrow}();$$
$$?\,\texttt{requestDenied}^{\downarrow}()\}$$

which characterizes a process that may branch in either one of two behaviors, depending if it receives message `requestApproved` *or message* `requestDenied`*, and, in the former case, proceeds by sending message* `transferDate`*. Now consider type:*

$$?\,\texttt{transferDate}^{\downarrow}()$$

which characterizes a process that receives message `transferDate`*. At this point we may ask if these two types can be merged, as there is a possible synchronization in message* `transferDate`*, but such synchronization is conditioned by the chosen branch. If* `transferDate` *is received by a*

party that does not influence this choice then we risk that the process receiving the `transferDate` *message will get stuck if the chosen branch is* `requestDenied`*. We thus disallow such merges, and allow for message shuffling only in simple message prefixes (unary choice and branch types).*

We may now define the behavioral types merge relation.

Definition 5.2.18 (Behavioral Types Merge Relation)
We say type behavioral type B is the merge of behavioral types B_1 and B_2, noted $B = B_1 \bowtie B_2$, if $B = B_1 \bowtie B_2$ is derivable by the rules shown in Figure 5.3.

We state some properties of the behavioral types merge relation.

Proposition 5.2.19 (Behavioral Type Merge Relation Properties)
The behavioral types merge relation is commutative and associative:

(1). If $B = B_1 \bowtie B_2$ then $B = B_2 \bowtie B_1$.

(2). If $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ then there is B'' such that $B'' = B_2 \bowtie B_3$ and $B = B_1 \bowtie B''$.

Proof. (*1*) follows immediately from the definition, while (*2*) follows by induction on the derivation of $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ (See Appendix A.3). ∎

We show an example that illustrates the behavioral combination of two types via merge.

Example 5.2.20 *Consider type:*

$$!\,\mathtt{request}^{\downarrow}().\,\&\{?\,\mathtt{requestApproved}^{\downarrow}().!\,\mathtt{transferDate}^{\downarrow}();\\ ?\,\mathtt{requestDenied}^{\downarrow}()\}$$

which specifies a process that sends message `request`*, and after which branches in either one of two behaviors, depending on the reception of either message* `requestApproved` *or message* `requestDenied`*, proceeding, in the former case, by sending message* `transferDate`*. When merged with the type:*

$$?\,\mathtt{request}^{\downarrow}().\tau\,\mathtt{assessRisk}^{\downarrow}().\tau\,\mathtt{riskVal}^{\downarrow}().\\ \oplus\{!\,\mathtt{requestApproved}^{\downarrow}().\tau\,\mathtt{orderTransfer}^{\downarrow}().?\,\mathtt{transferDate}^{\downarrow}();\\ !\,\mathtt{requestDenied}^{\downarrow}()\}$$

which specifies the dual polarity on message `request` *and the dual choice type, over messages* `requestApproved` *and* `requestDenied`*, and the dual polarity on message* `transferDate`*, along with some other internal message exchanges, it yields type:*

$$\tau\,\mathtt{request}^{\downarrow}().\tau\,\mathtt{assessRisk}^{\downarrow}().\tau\,\mathtt{riskVal}^{\downarrow}().\\ \oplus\{\tau\,\mathtt{requestApproved}^{\downarrow}().\tau\,\mathtt{orderTransfer}^{\downarrow}().\tau\,\mathtt{transferDate}^{\downarrow}();\\ \tau\,\mathtt{requestDenied}^{\downarrow}()\}$$

which specifies the composite behavior consisting on some internal message exchanges, including an internal choice.

$$\frac{l \in \mathcal{L}_\star}{\star ?\, l^d(C) = \star ?\, l^d(C) \bowtie \star ?\, l^d(C)}(\mathit{SharedInp})$$

$$\frac{B \asymp\, !\, l^d(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l^d(C)/\tau\ l^d(C)\} \mid \star\, ?\, l^d(C) = B \bowtie \star ?\, l^d(C)}(\mathit{Shared\text{-}r})$$

$$\frac{B \asymp\, !\, l^d(C) \qquad l \in \mathcal{L}_\star}{B\{!\ l^d(C)/\tau\ l^d(C)\} \mid \star\, ?\, l^d(C) = \star ?\, l^d(C) \bowtie B}(\mathit{Shared\text{-}l})$$

$$\frac{\forall_{i\in I}(B_i = B_i^- \bowtie B_i^+ \qquad l_i \in \mathcal{L}_p)}{\oplus_{i\in I}\{\tau\, l_i^{\downarrow}(C_i).B_i\} = \oplus_{i\in I}\{!\, l_i^{\downarrow}(C_i).B_i^+\} \bowtie \&_{i\in I}\{?\, l_i^{\downarrow}(C_i).B_i^-\}}(\mathit{Plain\text{-}r})$$

$$\frac{\forall_{i\in I}(B_i = B_i^- \bowtie B_i^+ \qquad l_i \in \mathcal{L}_p)}{\oplus_{i\in I}\{\tau\, l_i^{\downarrow}(C_i).B_i\} = \&_{i\in I}\{?\, l_i^{\downarrow}(C_i).B_i^-\} \bowtie \oplus_{i\in I}\{!\, l_i^{\downarrow}(C_i).B_i^+\}}(\mathit{Plain\text{-}l})$$

$$\frac{M \# B_1 \qquad B' \mid B'' \equiv B_1 \bowtie B_2 \qquad M \# B' \qquad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2) \qquad \mathcal{I}(B') \subseteq \mathcal{I}(B_1)}{B' \mid M.B'' = B_1 \bowtie M.B_2}(\mathit{Shuffle\text{-}r})$$

$$\frac{M \# B_2 \qquad B' \mid B'' \equiv B_1 \bowtie B_2 \qquad M \# B'' \qquad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \qquad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2)}{M.B' \mid B'' = M.B_1 \bowtie B_2}(\mathit{Shuffle\text{-}l})$$

$$\frac{B_1 \not\# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_3}{B = B_1 \bowtie B_2 \mid B_3}(\mathit{ShufflePar\text{-}r})$$

$$\frac{B_1 \not\# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_2}{B = B_1 \bowtie B_3 \mid B_2}(\mathit{ShufflePar\text{-}rr})$$

$$\frac{B_2 \not\# B_3 \qquad B' = B_2 \bowtie B_3 \qquad B = B_1 \bowtie B'}{B = B_1 \mid B_2 \bowtie B_3}(\mathit{ShufflePar\text{-}l})$$

$$\frac{B_2 \not\# B_3 \qquad B' = B_2 \bowtie B_3 \qquad B = B_1 \bowtie B'}{B = B_2 \mid B_1 \bowtie B_3}(\mathit{ShufflePar\text{-}ll})$$

$$\frac{B = B_1 \bowtie B_2}{\mathtt{rec}\,\mathcal{X}.B = \mathtt{rec}\,\mathcal{X}.B_1 \bowtie \mathtt{rec}\,\mathcal{X}.B_2}(\mathit{Rec}) \qquad \frac{}{\mathcal{X} = \mathcal{X} \bowtie \mathcal{X}}(\mathit{Var}) \qquad \frac{B_1 \# B_2}{B_1 \mid B_2 = B_1 \bowtie B_2}(\mathit{Apart})$$

$$\frac{B' = B_1 \bowtie B_2 \qquad B'' = B_3 \bowtie B_4 \qquad B' \# B''}{B' \mid B'' = (B_1 \mid B_3) \bowtie (B_2 \mid B_4)}(\mathit{Par\text{-}1})$$

$$\frac{B' = B_1 \bowtie B_4 \qquad B'' = B_3 \bowtie B_2 \qquad B' \# B''}{B' \mid B'' = (B_1 \mid B_3) \bowtie (B_2 \mid B_4)}(\mathit{Par\text{-}2})$$

Figure 5.3: Behavioral Type Merge Relation Rules.

$$\frac{B = B_1 \bowtie B_2}{n : [B] = n : [B_1] \bowtie n : [B_2]}(MergeLoc) \qquad T = T \bowtie \mathbf{0} \;\; (MergeZero\text{-}r)$$

$$\frac{B = B_1 \bowtie B_2 \quad L = L_1 \bowtie L_2}{L \mid B = L_1 \mid B_1 \bowtie L_2 \mid B_2}(MergeProc) \qquad T = \mathbf{0} \bowtie T \;\; (MergeZero\text{-}l)$$

$$\frac{\forall_{i \in 1,2}\; L_i = L_i^+ \bowtie L_i^- \quad dom(L_1) \,\#\, dom(L_2)}{L_1 \mid L_2 = L_1^+ \mid L_2^+ \bowtie L_1^- \mid L_2^-}(MergeLocPar) \quad \frac{dom(L_1) \,\#\, dom(L_2)}{L_1 \mid L_2 = L_1 \bowtie L_2}(MergeApart)$$

Figure 5.4: Process Types Merge Relation Rules.

The type system relies on a merge relation between process types, which we define by lifting the merge between behavioral types. We first define the domain of a process type.

Definition 5.2.21 (Domain of a Process Type)
The domain of a process type T, noted $dom(T)$, is defined as follows:

$$dom(T) \triangleq \{n \;\mid\; T \equiv T' \mid n : C\}$$

Definition 5.2.22 (Process Types Merge Relation)
We say process type T is the merge of process types T_1 and T_2, noted $T = T_1 \bowtie T_2$, if $T = T_1 \bowtie T_2$ is derivable by the rules given in Figure 5.4.

We are thus able to characterize the behavioral composition of processes, explained by the merge of their types. When such merges lead to a type that only specifies internal actions, we say that the type is *closed*, since it has no open external dependencies.

Definition 5.2.23 (Closed Types)
We say type B is closed, noted $closed(B)$, if for any message type $(p\, l^d(C))$ such that $(p\, l^d(C)) \in Msg_{\mathcal{L}}(B)$ then either $p = \tau$, or $p = ?$ and $l \in \mathcal{L}_\star$ and $B <: B' \mid \star\, ?\, l^d(C)$. We say type T is closed, noted $closed(T)$, if for any type B such that $T \equiv T' \mid n : [B]$ we have $closed(B)$.

Closed behavioral types characterize processes that have matching receives for all sends. Closed types are thus defined exclusively on messages of polarity τ, except for shared input $?$ message types that are still open to match with further outputs.

In the next section we show how the conversation type language and associated operations, characterize systems specified in the Conversation Calculus, so as to guarantee that processes follow the conversation protocols prescribed by the types.

5.3 Type System

In this section we present the type system that associates conversation types to Conversation Calculus systems. Using such type system we prove that if it is possible to type a CC system by our rules, then such system enjoys some safety properties, namely it is free from a certain kind of runtime errors, and also that its processes, at runtime, follow the protocols prescribed by the types. At the end of the section we present the mechanically derived rules for the service-oriented

idioms, showing that the derived typings actually correspond to the expected characterizations, and type an example so as to illustrate how the rules work in a concrete setting.

A typing judgment has the form $P :: T$, which says that process P is well-typed with process type T. This means that there is a derivation using our type rules for which $P :: T$ is the conclusion. We start by informally describing our typing rules, one for each construct of the Conversation Calculus language, that allow us to generically characterize CC specifications.

The rule for parallel composition:

$$\frac{P :: T_1 \quad Q :: T_2}{P \mid Q :: T_1 \bowtie T_2}$$

says the composition is well typed with the merge of the types of the branches. Recall the merge explains the composition of two processes by synchronizing behavioral traces. The rule for the inactive process:

$$\overline{\mathbf{0} :: \star L^!}$$

specifies that the process is well-typed with any exponential output located type, in such way allowing for the process to be used in a context where a process that performs some service calls is expected (cf., subtyping). The rule for the recursion variable also specifies such an exponential output located type:

$$\overline{\mathcal{X} :: \star L^! \mid \mathcal{X}}$$

but also, and crucially, introduces a recursion variable at the level of the current conversation. This recursion variable may then form the leaf of a (non-located) behavioral type, which is captured by a recursive type $B\langle\mathcal{X}\rangle$ in the premise of the recursion definition rule:

$$\frac{P :: \star L^! \mid B\langle\mathcal{X}\rangle}{\mathbf{rec}\,\mathcal{X}.P :: \star L^! \mid \mathtt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle}$$

The recursive type will then specify some linear behavior which is to be sequentially activated along with some exponential behavior. The body of the recursion may also specify an exponential output located type, allowing for the process to perform calls to remote services. Recursive processes specify the intended shared behavior using shared messages, in such way allowing several instances of the (shared part) of the recursive process to be concurrently active — the types of these several instances must be apart (#). The rule:

$$\frac{P :: T \mid a : [B] \quad (\mathit{closed}(B), a \notin \mathit{dom}(T))}{(\boldsymbol{\nu} a)P :: T}$$

types the name restriction by checking the behavioral type of the restricted conversation is closed, in such case eliding it in the conclusion of the rule. The type associated to a process only describes the behaviors in the visible conversations, and the closed condition avoids hiding a conversation name where unmatched communications still persist (necessary to ensure progress).

The following rule types a (piece of a) conversation:

$$\frac{P :: L \mid B \quad \mathit{fv}(B) = \emptyset}{n \blacktriangleleft [P] :: (L \bowtie n : [\downarrow B]) \mid \mathit{loc}(\uparrow B)}$$

The process located in the conversation piece P defines behavior located in conversations L, and

some behavior B in the current conversation. Then, the type in the conclusion is obtained by merging the process type L with a type that describes the behavior of the new conversation piece, in parallel with the type of the toplevel conversation, the now current conversation. Essentially the type of each projection (along the two directions) is collected appropriately: the "here" behavior projection $\downarrow B$ is the behavior in conversation n, and the "up" behavior projection $\uparrow$ of P becomes the "here" behavior at the toplevel conversation, via $loc(\uparrow B)$. Rule:

$$\frac{P :: L \mid B_1 \mid x : [B_2] \quad (x \notin dom(L))}{\textbf{this}(x).P :: L \mid (B_1 \bowtie B_2)}$$

gives the typing of the conversation awareness primitive, specifying that behavior B_2 of conversation x is a separate (in general, just partial) view of the current conversation. This allows to bind the current conversation to name x, and possibly send it to other parties that may need to join it. Names are sent in output action prefixes which are typed by the following rule:

$$\frac{P :: L \mid B}{l^d!(n).P :: (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\}}$$

The continuation process P defines some located behavior L and some unlocated behavior B. Then, the output prefix is typed by the merge of the delegated conversation fragment $n : C$ with the located behavior L, along with a choice type that includes the output action specified in the prefix with respective continuation. Notice that the conversation fragment piece that is delegated away is actually a separate $\bowtie$ view of conversation n, which means that the type being sent may actually be some separate part of the type of the conversation. This mechanism is crucial to allow external partners to join in on ongoing conversations in a disciplined way. The behavioral interface of the output prefixed process is a choice type, as the process can choose the specified action from any set of choices that contains it. Notice also that the continuation is given by the unlocated type in the premise (B), instead of being determined by a partial view of the type of some conversation (as required in the typing of the output in the π_{lab}-Calculus — see Figure 3.6). In such way, the analysis is simplified as the rule does not depend on the merge and, more importantly, the static sequentiality information introduced in the rule can always be reproduced, even in the presence of name substitution (in the sense that if $P :: L \mid B$ then $P\{n/m\} :: L' \mid B$).

The rule that types the input guarded summation is the following:

$$\frac{\forall_{i \in I}(P_i :: L \mid B_i \mid x_i : C_i \quad (x_i \notin dom(L),\ \forall_{j \neq i}\, l_i \neq l_j))}{\Sigma_{i \in I}\, l_i^d?(x_i).P_i :: L \mid \&_{i \in I}\{?\, l_i^d(C_i).B_i\}}$$

where the premises (one for each $i \in I$) state that processes P_i specify some located behavior L, some current conversation behavior B_i, and some behavior C_i at conversation x_i. Then, the conclusion states that the input summation process is well-typed with type L, with the behavior interface becoming the branch of the input message types $?\, l_i^d(C_i)$, where the argument type is the type of the respective x_i variable, with respective continuations. Notice that the located behavior L must be common to all continuation processes (P_i), so as to allow for the external interface of the guarded sum to be uniquely determined. Introducing some exponential behaviors in the typing derivation at the level of the axioms (for inactive process and recursion variable) can be useful here, as it allows for some branches that do not really use such exponential behavior

$$\frac{P :: T_1 \quad Q :: T_2}{P \mid Q :: T_1 \bowtie T_2}(Par) \qquad \frac{}{\mathbf{0} :: \star L^!}(Stop)$$

$$\frac{P :: \star L^! \mid B\langle \mathcal{X} \rangle}{\mathbf{rec}\, \mathcal{X}.P :: \star L^! \mid \mathtt{rec}\, \mathcal{X}.B\langle \mathcal{X} \rangle}(Rec) \qquad \frac{}{\mathcal{X} :: \star L^! \mid \mathcal{X}}(RecVar)$$

$$\frac{P :: T \mid a : [B] \quad (closed(B), a \notin dom(T))}{(\boldsymbol{\nu} a)P :: T}(Res)$$

$$\frac{P :: L \mid B \quad (fv(B) = \emptyset)}{n \blacktriangleleft [P] :: (L \bowtie n : [\downarrow B]) \mid loc(\uparrow B)}(Piece)$$

$$\frac{P :: L \mid B_1 \mid x : [B_2] \quad (x \notin dom(L))}{\mathbf{this}(x).P :: L \mid (B_1 \bowtie B_2)}(This)$$

$$\frac{P :: L \mid B}{l^d!(n).P :: (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\}}(Output)$$

$$\frac{\forall_{i \in I}(P_i :: L \mid B_i \mid x_i : C_i \quad (x_i \notin dom(L),\ \forall_{j \neq i}\ l_i \neq l_j))}{\Sigma_{i \in I}\ l_i^d?(x_i).P_i :: L \mid \&_{i \in I}\{?\, l_i^d(C_i).B_i\}}(Input)$$

$$\frac{P :: T_1 \quad T_1 <: T_2}{P :: T_2}(Sub)$$

Figure 5.5: Typing Rules.

to share a common interface with branches that do make use of such exponential behavior.

Notice there is some asymmetry between the output and input rules: while we can safely consider that the process can choose a given output in between any other choices, we cannot forget any branches in the branch type, as this would allow undesired matches between choice and branch types. If a process does not fully reveal all the behaviors if offers then, when composed with a process that can actually choose the "forgotten" action, unexpected behaviors may arise, i.e., behaviors not described by the type (cf., [35] where a similar problem arises in contract compliance). Such flexibility of choice types is usually introduced at the level of width subtyping for choice types. The reason we introduce the choice directly in the output rule instead of specifying such a width subtyping axiom is due to the merge relation: as discussed in Example 5.2.17 we do not allow shuffling in choice prefixes while we do allow it in simple message prefixes, so we specify a priori the "final" type so as to avoid interference between subtyping and merge. Subtyping characterizations are introduced by the subsumption rule:

$$\frac{P :: T_1 \quad T_1 <: T_2}{P :: T_2}$$

We may now define our typing relation between CC systems and conversation types.

Definition 5.3.1 (Well-Typed CC Process)
We say a CC process P is well typed if there is process type T such that $P :: T$ can be derived by the rules given in Figure 5.5.

Next we show our results that prove that CC systems that get past our rules are free from a certain kind of runtime errors, entailing that the prescribed protocols will be met at runtime.

$$\tau\, l^d(C).B \to B \ (\mathit{RedTau}) \qquad \frac{B \to B'}{\oplus\{B; \tilde{B}\} \to B'}(\mathit{RedChoice})$$

$$\frac{B_1 \to B_2}{n : [B_1] \to n : [B_2]}(\mathit{RedPiece}) \qquad \frac{T_1 \to T_2}{T_1 \mid T_3 \to T_2 \mid T_3}(\mathit{RedPar})$$

$$T \to T \ \ (\mathit{RedReflex}) \qquad \frac{T_1 \equiv T_1' \to T_2' \equiv T_2}{T_1 \to T_2}(\mathit{RedEquiv})$$

Figure 5.6: Process Types Reduction Rules.

5.3.1 Type Safety

In this section we show our type safety results that prove well-typed systems are free from a specific class of errors, and that well-typedness is invariant under reduction, which thus entail well-typed systems always evolve to error free configurations. Preservation of typing under reduction — Subject Reduction — is defined using a notion of reduction on types, since each reduction step at the process level may require a modification in the typing.

Definition 5.3.2 (Process Types Reduction)
We say process type T_1 reduces to process type T_2, noted $T_1 \to T_2$, if $T_1 \to T_2$ can be derived by the rules given in Figure 5.6.

We can then explain a reduction in a process by reducing a τ message type, i.e., by activating the continuation of a τ message type. Since some behavior may not visible in the type, namely message exchanges in restricted conversations, we close the type reduction under reflexivity so as to capture such silent evolutions. We may now state our soundness results, starting by a weakening property of our typing associations.

Proposition 5.3.3 (Weakening)
Let P be a well-typed process such that $P :: T$. If exponential output located type $\star L^!$ is such that T and $\star L^!$ are apart, hence $T \# \star L^!$, then $P :: T \mid \star L^!$.

Proof. Follows by induction on the length of the derivation of $P :: T$ in expected lines. Essentially $\star L^!$ is introduced at the level of the axioms (*Stop*) and (*RecVar*), and pushed down in the derivation separately, being the independence guaranteed by $T \# \star L^!$ (see Appendix A.3). ∎

We now state a Substitution Lemma, main auxiliary result to Subject Reduction.

Lemma 5.3.4 (Substitution Lemma)
Let P be a well typed process such that $P :: T \mid x : C$ and $x \notin dom(T)$. If there is T' such that $T' = T \bowtie a : C$ then $P\{x/a\} :: T'$.

Proof. By induction on the length of the derivation of $P :: T \mid x : C$ (see Appendix A.3). ∎

The substitution lemma plays a crucial role in the proof of subject reduction. Synchronizations allow for names to be passed around, so we need to make sure that once a variable is instantiated by a name that the resulting system is still well-typed. The substitution lemma guarantees this, provided the type of the delegated conversation fragment is "mergeable" with the overall type,

a condition that is ensured by the process that is sending the name. We may now state our subject reduction result.

Theorem 5.3.5 (Subject Reduction)
Let P be a well-typed process such that $P :: T$. If $P \to Q$ then there is $T \to T'$ such that $Q :: T'$.

Proof. By induction on length of the derivation of $P \to Q$ (see Appendix A.3). ∎

Subject reduction thus guarantees that well-typedness is invariant under process reduction. Moreover, each reduction in a process is explained by a reduction in the type.

Our type safety result asserts that certain error processes are unreachable from well-typed processes. To define error processes we introduce static process contexts and an auxiliary notion that characterizes single output or input prefixed processes.

Definition 5.3.6 (CC Static Context)
CC static process contexts, noted $\mathcal{C}[\cdot]$, are defined as follows:

$$\mathcal{C}[\cdot] \ ::= \ (\boldsymbol{\nu} a)\mathcal{C}[\cdot] \ \big| \ P \mid \mathcal{C}[\cdot] \ \big| \ c \blacktriangleleft [\mathcal{C}[\cdot]] \ \big| \ \cdot \qquad \text{(CC Static Context)}$$

Definition 5.3.7 (CC Prefix Process)
We denote by $l^d\!-\!.P$ a prefix process that is ready to communicate on l^d, defined as follows:

$$l^d\!-\!.P \ ::= \ l^d!(\tilde{a}).P \ \big| \ l^d?(\tilde{x}).P \qquad \text{(CC Prefix Process)}$$

We consider an error process to be a configuration where there is an active race on a linear message, which means two processes are willing to send or are waiting to receive the same message. We thus characterize error processes by observing that two distinct parts of the process can synchronize with the same message prefix.

Definition 5.3.8 (Error Process)
We say P is an error process if there are contexts $\mathcal{C}, \mathcal{C}'$, processes Q, R with $P = \mathcal{C}[Q \mid R]$, direction d, and labels $l, \texttt{flag}$ with $l \in \mathcal{L}_p$ and $\texttt{flag} \notin \mathit{fl}(P)$ such that:

$$\mathcal{C}[Q \mid \mathcal{C}'[l^d\!-\!.\texttt{flag}^{\downarrow}!()]] \to \mathcal{C}''[\texttt{flag}^{\downarrow}!()] \quad \textit{and} \quad \mathcal{C}[R \mid \mathcal{C}'[l^d\!-\!.\texttt{flag}^{\downarrow}!()]] \to \mathcal{C}''[\texttt{flag}^{\downarrow}!()]$$

This characterization of error processes may be viewed, informally, as observational, since we are testing the behavior of the process by allowing it to interact with a specially crafted context (much like the proof methods used in barbed equivalences, cf., [85]). Thus, a process is not an error only if for each possible immediate interaction in a plain message there is at most a single sender and a single receiver. We now assert that well-typed systems are error free.

Proposition 5.3.9 (Error Freeness)
If P is a well-typed process then P is not an error process.

Proof. Follows by auxiliary results and by definition of merge (see Appendix A.3). ∎

We thus conclude that any process reachable from a well-typed process $P :: T$ is not an error.

Corollary 5.3.10 (Type Safety)
Let P be a well-typed process. If there is Q such that $P \xrightarrow{*} Q$, then Q is not an error process.

Proof. Immediate from Theorem 5.3.5 and Proposition 5.3.9. ∎

Our type safety result ensures that, in any reduction sequence arising from a well-typed process, for each plain-labeled message ready to communicate there is always at most a unique input/output outstanding synchronization. More: arbitrary interactions in shared labels do not invalidate this invariant. Another consequence of subject reduction (Theorem 5.3.5) is that any message exchange inside the process must be explained by a τM prefix in the related conversation type (via type reduction), thus implying conversation fidelity, i.e., all conversations follow the prescribed protocols.

Corollary 5.3.11 (Conversation Fidelity)
Let P be a process such that $P :: T$ for some T. Then all conversations in P follow the protocols prescribed by T.

Example 5.3.12 *Consider for instance the typing for the purchase conversation presented in the Introduction (Section 1.2.4):*

$$\tau\,\mathtt{buy}(Tp).\tau\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)$$

Such a (closed) type characterizes the global interaction scheme (the choreography*) of the interaction between parties Buyer, Seller and Shipper. In the light of Theorem 5.3.5 we then have that each interaction in the process is explained by a reduction of a τ message type. For instance, when message* `buy` *is exchanged between Buyer and Seller, such synchronization in the process is explained by the following reduction in the type:*

$$\begin{array}{l}\tau\,\mathtt{buy}(Tp).\tau\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)\\ \rightarrow\\ \tau\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)\end{array}$$

after which the synchronization in message `price` *is explained by the following type reduction:*

$$\begin{array}{l}\tau\,\mathtt{price}(Tm).\tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)\\ \rightarrow\\ \tau\,\mathtt{product}(Tp).\tau\,\mathtt{details}(Td)\end{array}$$

and so on and so forth in the successive synchronizations. Thus, the type reductions actually capture the evolution of the choreographies throughout system execution.

In the expected polyadic extension of core CC and type system, considering also basic values and basic types, we would also exclude arity mismatch and type mismatch errors.

Remark 5.3.13 *Extending the conversation type language with basic types at the level of argument message types (e.g., $C ::= [B] \mid \mathtt{Int} \mid \mathtt{String} \mid \ldots$) would directly allow us to exclude systems where message argument types mismatch (via conformance $\asymp$ and merge $\bowtie$). To type*

identifiers which carry basic types we would impose a conformance check — such identifiers are always used with the same type — and exclude their use as conversation names.

Remark 5.3.14 *A result which is frequently found in related approaches is Subject Congruence: well-typedness is invariant under structural congruence. Since our operational semantics does not rely on structural congruence we do not prove this result. In fact, under the structural rules introduced in Section 4.4.2 and the typing analysis presented here, where we exploit the syntactic information about the current conversation, the result does not hold.*

An essential property of any type system is the ability to automate the type checking procedure. Although we have not yet fully addressed the implementation issues, we may already state a crucial property that asserts the existence of such a type checking procedure.

Theorem 5.3.15 (Decidability of Type Checking)
Let P be a process where all bound names are type annotated. Then checking if $P :: T$ for some T is decidable.

Proof. By induction on the derivation of $P :: T$, following expected lines (See Appendix A.3). ∎

We prove decidability of our system, if binders are type annotated. This is an expected result, since our typing rules are syntax-directed, our merge relation is finitary, and typability is witnessed by a proof tree (as usual).

5.3.2 Derived Typings for Service Idioms

In this section we present the typing for the service-oriented idioms. Since such idioms are defined using lower-level communication primitives, the typing rules for the idioms are not primitive, instead they may be mechanically derived from the typing of the idioms lower-level implementation. We recall the implementations (defined in Section 4.5) and show the derived typing rules. The service definition is implemented as follows:

$$\mathsf{\textbf{def}}\, s \Rightarrow P \quad \triangleq \quad s^{\downarrow}?(x).x \blacktriangleleft [P] \qquad\qquad (x \notin fv(P))$$

we then have the following typing derivation using rules (*Piece*) and (*Input*):

$$\dfrac{\dfrac{P :: L \mid B \quad (x \notin dom(L))}{x \blacktriangleleft [P] :: L \mid x : [\downarrow B] \mid loc(\uparrow B)}}{s^{\downarrow}?(x).x \blacktriangleleft [P] :: L \mid ?\, s^{\downarrow}([\downarrow B]).(loc(\uparrow B))}$$

where we may assume, without any loss of generality, that x is not in the domain of L since it is not a free variable of P. We thus have the following typing for the service definition:

$$\dfrac{P :: L \mid B}{\mathsf{\textbf{def}}\ \mathsf{s} \Rightarrow P :: L \mid ?\, s^{\downarrow}([\downarrow B]).(loc(\uparrow B))}$$

Ignoring the continuation $loc(\uparrow B)$, which describes the access to resources local to the context where the service is published at, the type of a service definition **def** has the form $?\, s([B])$, where B describes the service provider behavior in the service collaboration.

The implementation for the persistent service definition is as follows:

$$\star\textbf{def}\ s \Rightarrow P \ \triangleq\ \textbf{rec}\,\mathcal{X}.s^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) \qquad (x \notin fv(P))$$

The typing rule for the persistent service definition is the following:

$$\frac{\textbf{def}\ s \Rightarrow P :: \star L^{!} \mid \texttt{?}\,s^{\downarrow}(C).(\star B^{!})}{\star\textbf{def}\ s \Rightarrow P :: \star L^{!} \mid \texttt{rec}\,\mathcal{X}.\texttt{?}\,s^{\downarrow}(C).(\star B^{!} \mid \mathcal{X})}$$

which is mechanically derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{X} :: \mathcal{X}} \qquad \dfrac{P :: L \mid B \quad (x \notin dom(L))}{x \blacktriangleleft [P] :: L \mid x : [\downarrow B] \mid loc(\uparrow B)}}{\mathcal{X} \mid x \blacktriangleleft [P] :: L \mid x : [\downarrow B] \mid loc(\uparrow B) \mid \mathcal{X}}}{s^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) :: L \mid \texttt{?}\,s^{\downarrow}([\downarrow B]).(loc(\uparrow B) \mid \mathcal{X})} \qquad (\star B^{!} = loc(\uparrow B), \star L^{!} = L)}{\textbf{rec}\,\mathcal{X}.s^{\downarrow}?(x).(\mathcal{X} \mid x \blacktriangleleft [P]) :: \star L^{!} \mid \texttt{rec}\,\mathcal{X}.\texttt{?}\,s^{\downarrow}([\downarrow B]).(\star B^{!} \mid \mathcal{X})}$$

Notice that $\texttt{rec}\,\mathcal{X}.\texttt{?}\,s^{\downarrow}(C).(\star B^{!} \mid \mathcal{X}) <: \star B^{!} \mid \star\,\texttt{?}\,s^{\downarrow}(C)$. Then, when we place such a persistent service in its respective conversation we may obtain the following typing characterization:

$$\frac{\star\textbf{def}\ s \Rightarrow P :: \star L^{!} \mid \star\,B^{!} \mid \star\,\texttt{?}\,s^{\downarrow}(C)}{n \blacktriangleleft [\star\textbf{def}\ s \Rightarrow P] :: \star L^{!} \bowtie (n : [\star B^{!} \mid \star\,\texttt{?}\,s^{\downarrow}(C)])}$$

The merge in the conclusion thus allows, for instance, to capture the case of a service that calls itself, in which case the merge introduces a τ in the service call specified in $\star L^{!}$, as the result of merging it with $n : [\star\texttt{?}\,s^{\downarrow}(C)]$. Also, the compatibility (conformance $\asymp$) between the service calls in $\star L^{!}$ and in $n : \left[\star B^{!}\right]$ is checked via the merge.

For the service instantiation idiom we have the following implementation:

$$\textbf{new}\ n \cdot s \Leftarrow P \ \triangleq\ (\boldsymbol{\nu} c)(n \blacktriangleleft \left[s^{\downarrow}!(c)\right] \mid c \blacktriangleleft [P]) \qquad (c \notin (fn(P) \cup \{n\}))$$

for which we have the following typing derivation (by rules (*Output*), (*Piece*), (*Par*) and (*Res*)):

$$\dfrac{\dfrac{\dfrac{s^{\downarrow}!(c) :: c : [B_1] \mid \texttt{!}\ s^{\downarrow}([B_1])}{n \blacktriangleleft [s^{\downarrow}!(c)] :: c : [B_1] \mid n : [\texttt{!}\ s^{\downarrow}([B_1])]} \qquad \dfrac{P :: L \mid B_2}{c \blacktriangleleft [P] :: L \mid c : [\downarrow B_2] \mid loc(\uparrow B_2)}}{n \blacktriangleleft [s^{\downarrow}!(c)] \mid c \blacktriangleleft [P] :: (L \bowtie n : [\texttt{!}\ s^{\downarrow}([B_1])]) \mid c : [B_1 \bowtie\downarrow B_2] \mid loc(\uparrow B_2) \quad closed(B_1 \bowtie\downarrow B_2)}}{(\boldsymbol{\nu} c)(n \blacktriangleleft [s^{\downarrow}!(c)] \mid c \blacktriangleleft [P]) :: (L \bowtie n : [\texttt{!}\ s^{\downarrow}([B_1])]) \mid loc(\uparrow B_2)}$$

Thus, the service instantiation primitive is typed as follows:

$$\frac{P :: L \mid B_2 \quad (closed(\downarrow B_2 \bowtie B_1))}{\textbf{new}\ n \cdot s \Rightarrow P :: (L \bowtie n : [\texttt{!}\ s^{\downarrow}([B_1])]) \mid loc(\uparrow B_2)}$$

The type for a service instantiation is of the form $\texttt{!}\ s([B])$, where B describes the compatible server behavior. However, such type must be located at some context n, cf. the semantics of **new**. Notice that the output on the service message at n is merged with the located type L. This is necessary since P may specify behavior at n, in particular it may specify other calls to service s which, via $\bowtie$, are checked to be conformant $\asymp$. Also, P itself may publish the required service, in which case a possible synchronization is captured in the merge — P is actually running in parallel to the service instantiation request. Although this is not the intended

$$\frac{P :: L \mid B}{\textbf{def}\ \mathtt{s} \Rightarrow P :: L \mid ?\, s^{\downarrow}([\downarrow B]).(loc(\uparrow B))}(Def)$$

$$\frac{\textbf{def}\ s \Rightarrow P :: \star L^{!} \mid ?\, s^{\downarrow}(C).(\star B^{!})}{\star\textbf{def}\ s \Rightarrow P :: \star L^{!} \mid \mathtt{rec}\, \mathcal{X}.?\, s^{\downarrow}(C).(\star B^{!} \mid \mathcal{X})}(RepDef)$$

$$\frac{P :: L \mid B_2 \quad (closed(\downarrow B_2 \bowtie B_1))}{\textbf{new}\ n \cdot s \Rightarrow P :: (L \bowtie n : [!\ s^{\downarrow}([B_1])]) \mid loc(\uparrow B_2)}(New)$$

$$\frac{P :: L \mid B_2}{\textbf{join}\ n \cdot s \Leftarrow P :: (L \bowtie n : [!\ s^{\downarrow}([B_1])]) \mid B_1 \bowtie B_2}(Join)$$

Figure 5.7: Service Idioms Derived Typing Rules.

usage, it is not excluded by the typing, but it could be easily be so by an extra condition (e.g., $n : [!\ s^{\downarrow}(B_1)] \# L$). The implementation of the conversation join is given as follows:

$$\textbf{join}\ n \cdot s \Leftarrow P \ \triangleq\ \textbf{this}(x).(n \blacktriangleleft \left[s^{\downarrow}!(x)\right] \mid P) \qquad (x \notin (fv(P) \cup \{n\}))$$

We have the following typing for the conversation join idiom:

$$\frac{P :: L \mid B_2}{\textbf{join}\ n \cdot s \Leftarrow P :: (L \bowtie n : [!\ s^{\downarrow}([B_1])]) \mid B_1 \bowtie B_2}$$

which is derived as follows (using rules (*Output*), (*Piece*), (*Par*) and (*This*)):

$$\cfrac{\cfrac{\cfrac{s^{\downarrow}!(x) :: x : [B_1] \mid !\ s^{\downarrow}([B_1])}{n \blacktriangleleft [s^{\downarrow}!(x)] :: x : [B_1] \mid n : [!\ s^{\downarrow}([B_1])]} \qquad P :: L \mid B_2}{n \blacktriangleleft [s^{\downarrow}!(x)] \mid P :: (L \bowtie n : [!\ s^{\downarrow}([B_1])]) \mid x : [B_1] \mid B_2}}{\textbf{this}(x).(n \blacktriangleleft [s^{\downarrow}!(x)] \mid P) :: (L \bowtie n : [!\ s^{\downarrow}([B_1])]) \mid B_1 \bowtie B_2}$$

The typing for **join** clearly displays the partial delegation of a conversation type fragment: B_1 represents the conversation type defining the participation of the incoming partner, while B_2 specifies the residual that remains owned by the current process.

We group the rules in Figure 5.7. We have thus shown how these rules can be mechanically derived from the typings of the encodings. Remarkably, the typings of these idiomatic constructs, defined from the small set of primitives in the core CC, admit the intended high level typings.

5.3.3 Typing Examples

Finance Portal

In order to show how conversation types deal with complex realistic scenarios we show the typings for the Finance Portal example, which involves an a priori undetermined number of parties. In the example we model a credit request scenario, involving several parties that collaborate so as to grant or not a credit to a costumer. The interaction is mediated by a bank portal, which asks a bank clerk to join the ongoing interaction at some point, after which it is either able to authorize the credit automatically or delegates the final decision to a bank manager.

In the following we assume the extension of the language and type system with basic values and basic types, respectively, and we represent the latter by a $\mathtt{T}$ ($\mathtt{T}_1, \mathtt{T}_2, \ldots$). We recall the

implementation given for the *CreditTransfer* service, which is ultimately called in the credit request service collaboration so as to schedule the money transfer.

$$\begin{array}{l} \mathit{BankATMProcess} \triangleq \\ \quad \mathit{BankATM} \blacktriangleleft [\ \mathit{BankATMServer} \\ \qquad \mid \star\ \textbf{def}\ \texttt{CreditTransfer} \Rightarrow \\ \mathit{CreditTransferBody} \triangleq \begin{cases} \texttt{orderTransfer}^{\downarrow}?(\mathit{userId}, \mathit{amount}). \\ \texttt{transferDate}^{\downarrow}?(\mathit{date}). \\ \texttt{scheduleTransfer}^{\uparrow}!(\mathit{userId}, \mathit{amount}, \mathit{date})\ \ ] \end{cases} \end{array}$$

The body of the persistent service definition is typed as follows:

$$\begin{array}{l} \mathit{CreditTransferBody} :: \\ \quad ?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4).!\,\texttt{scheduleTransfer}^{\uparrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \end{array}$$

for some basic types $\texttt{T}_1$, $\texttt{T}_2$ and $\texttt{T}_4$. The `CreditTransfer` service will then be typed with the $\downarrow$ projection of the above type, namely:

$$?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)$$

while the interface of the service with the enclosing conversation is typed by the $\uparrow$ projection:

$$!\,\texttt{scheduleTransfer}^{\uparrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4)$$

Since `scheduleTransfer` is specified in the interface of a persistent service it must be the case that it is a shared label (from $\mathcal{L}_\star$), otherwise we would not be able to type it, thus we have:

$$!\,\texttt{scheduleTransfer}^{\uparrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) <: \star!\,\texttt{scheduleTransfer}^{\uparrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4)$$

We may then type the service definition as follows:

$$\begin{array}{l} \star\,\textbf{def}\ \texttt{CreditTransfer} \Rightarrow \mathit{CreditTransferBody}\ :: \\ \quad \star?\,\texttt{CreditTransfer}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)) \\ \quad \mid \star!\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \end{array}$$

Notice that `scheduleTransfer` is now directed to the current $\downarrow$ conversation — the up $\uparrow$ interface of the body of a service definition is the here $\downarrow$ interface of the service definition (service message aside). The above typing makes use of subtyping rules (*SubParPref*) and (*SubExpRec*), so as to allow for the exponential output interface of the recursive type to be specified separately:

$$\begin{array}{l} \texttt{rec}\,\mathcal{X}.?\,\texttt{CreditTransfer}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)). \\ \quad (\star!\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \mid \mathcal{X}) \\ <: \\ \texttt{rec}\,\mathcal{X}.(?\star!\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \\ \quad \mid \texttt{CreditTransfer}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)).\mathcal{X}) \\ <: \\ \texttt{rec}\,\mathcal{X}.?\,\texttt{CreditTransfer}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)).\mathcal{X} \\ \mid \star!\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \end{array}$$

Considering *BankATMServer* — which implementation we abstract away from — is a process ready to persistently receive scheduleTransfer messages, as characterized by: *BankATMServer* :: $\star ?\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4)$ then, we have the following typing for the *BankATMProcess*:

$$\begin{array}{l} \textit{BankATMProcess} :: \\ \quad \textit{BankATM} : [\star ?\,\texttt{CreditTransfer}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)) \\ \quad\quad \mid \star\tau\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4) \mid \star ?\,\texttt{scheduleTransfer}^{\downarrow}(\texttt{T}_1, \texttt{T}_2, \texttt{T}_4)] \end{array}$$

where the τ scheduleTransfer results from the merge of the output ! and input ? polarities. Since the message is shared, the merge leaves open the possibility for the input to match with other outputs (notice the input ? message type is still present). We thus have that the *BankATMProcess* publishes service CreditTransfer at conversation *BankATM*, which, once instantiated, will receive messages orderTransfer and transferDate in the service conversation. Also, the *BankATMProcess* internally exchanges messages scheduleTransfer in the *BankATM* conversation, being open to receive further scheduleTransfer messages.

We now show the code for the CreditApproval service, where we consider that process *ManagerTerminal* is somehow able to interact with the manager, e.g., via a computer terminal.

$$\begin{array}{l} \textit{ManagerProcess} \triangleq \\ \quad \textit{Manager} \blacktriangleleft [\textit{ManagerTerminal} \\ \quad\quad \mid \star\ \textbf{def}\ \texttt{CreditApproval} \Rightarrow \\ \quad\quad\quad \texttt{requestApproval}^{\downarrow}?(\textit{clientChat}, \textit{userId}, \textit{amount}, \textit{risk}). \\ \quad\quad\quad \textbf{this}(\textit{managerChat}).\texttt{showRequest}^{\uparrow}!(\textit{managerChat}, \textit{userId}, \textit{amount}, \textit{risk}). \\ \quad\quad\quad\quad (\texttt{reject}^{\downarrow}?().\textit{clientChat} \blacktriangleleft [\ \texttt{requestDenied}^{\downarrow}!()\ ] \\ \quad\quad\quad\quad +\texttt{accept}^{\downarrow}?().\textit{clientChat} \blacktriangleleft [\\ \quad\quad\quad\quad\quad\quad \texttt{requestApproved}^{\downarrow}!(). \\ \quad\quad\quad\quad\quad\quad \textbf{join}\ \textit{BankATM} \cdot \texttt{CreditTransfer} \Leftarrow \\ \quad\quad\quad\quad\quad\quad\quad \texttt{orderTransfer}^{\downarrow}!(\textit{userId}, \textit{amount})\ ]\ ] \end{array}$$

The code implements a service which handles the final acceptance of a credit request by a bank manager. The service code specifies that message requestApproval is received, containing some data of the credit request and also the name of the conversation in which the bank client will be notified of the manager decision (*clientChat*). Then, message showRequest is sent to the *ManagerTerminal* process, containing the name of the current conversation (*managerChat*), accessed via the **this**. In such way, the *ManagerTerminal* process may reply (by sending either message reject or accept) directly in the respective CreditApproval service instance conversation. Although several showRequest messages may race in conversation *Manager*, as there may be several copies of the service running concurrently, we do not intend for the replies to such messages to compete in conversation *Manager* as they would be possibly be picked up by some unrelated service instance. Thus, to guarantee that such replies are handled by their respective service instances, they are directly placed in the respective conversations.

After receiving the decision of the manager, the service instance then informs the client on the final decision by sending either message requestDenied or message requestApproved, in the conversation received in the requestApproval message identified by *clientChat*. In the latter case, the *BankATM* · CreditTransfer service is called in to collaborate in the *clientChat*

conversation, and an `orderTransfer` message is sent to place the order for the money transfer. Thus, the role of the manager in the client conversation is characterized by the following type:

$$\begin{aligned}&\mathit{managerRole} \triangleq\\ &\qquad \oplus\{\,!\,\texttt{requestApproved}^{\downarrow}().\tau\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4);\\ &\qquad\quad !\,\texttt{requestDenied}^{\downarrow}()\}\end{aligned}$$

where, in the case of the `requestApproved` choice, the behavior results from the merge of the rest of the manager role in the client conversation with the type of the `CreditTransfer` service (at the level of the typing of the **this** used in the implementation of the **join**):

$$!\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\bowtie ?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4)$$

The `CreditApproval` service is characterized by the type:

$$\begin{aligned}&\mathit{creditApprovalB} \triangleq\\ &\qquad ?\,\texttt{requestApproval}^{\downarrow}(\mathit{managerRole},\texttt{T}_1,\texttt{T}_2,\texttt{T}_3).\oplus\{\tau\,\texttt{reject}^{\downarrow}();\ \tau\,\texttt{accept}^{\downarrow}()\}\end{aligned}$$

which specifies the reception of a `requestApproval` message, carrying a conversation identifier in which the manager behaves as specified by *managerRole*, after which proceeding as the internal choice between messages `reject` and `accept`. We may then type the *ManagerProcess* as follows:

$$\begin{aligned}&\mathit{ManagerProcess} ::\\ &\quad \mathit{Manager} : [\star ?\,\texttt{CreditApproval}^{\downarrow}(\mathit{creditApprovalB})\\ &\qquad\qquad \mid \star ?\,\texttt{showRequest}^{\downarrow}(\oplus\{\,!\,\texttt{reject}^{\downarrow}();\ !\,\texttt{accept}^{\downarrow}()\},\texttt{T}_1,\texttt{T}_2,\texttt{T}_3)\\ &\qquad\qquad \mid \star\tau\,\texttt{showRequest}^{\downarrow}(\oplus\{\,!\,\texttt{reject}^{\downarrow}();\ !\,\texttt{accept}^{\downarrow}()\},\texttt{T}_1,\texttt{T}_2,\texttt{T}_3)]\\ &\quad \mid \mathit{BankATM} : [\star !\,\texttt{CreditTransfer}^{\downarrow}(?\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).?\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4))]\end{aligned}$$

which specifies that `CreditApproval` service is published (`?`) at conversation *Manager*, and that service `CreditTransfer` is required (`!`) at conversation *BankATM*. The type also specifies that some *showRequest* messages may be exchanged in conversation *Manager*, carrying a conversation identifier in which the delegated behavior is characterized by the choice type $\oplus\{\,!\,\texttt{reject}^{\downarrow}();\ !\,\texttt{accept}^{\downarrow}()\}$ — characterizing the role of the *ManagerTerminal* process in the `CreditApproval` conversation.

Next we show the code of the `CreditRequest` service published at the *BankPortal* context.

$$\begin{aligned}&\mathit{BankProcess} \triangleq\\ &\quad \mathit{BankPortal} \blacktriangleleft [\\ &\qquad \star\ \textbf{def}\ \texttt{CreditRequest} \Rightarrow\\ &\qquad\quad \texttt{request}^{\downarrow}?(\mathit{userId},\mathit{amount}).\\ &\qquad\quad \textbf{join}\ \mathit{Clerk}\cdot\texttt{RiskAssessment} \Leftarrow\\ &\qquad\qquad \texttt{assessRisk}^{\downarrow}!(\mathit{userId},\mathit{amount}).\\ &\qquad\qquad \texttt{riskVal}^{\downarrow}?(\mathit{risk}).\\ &\qquad\qquad \textbf{if}\ \mathit{risk} = \texttt{HIGH}\ \textbf{then}\ \texttt{requestDenied}^{\downarrow}!()\\ &\qquad\qquad \textbf{else this}(\mathit{clientChat}).\\ &\qquad\qquad\quad \textbf{new}\ \mathit{Manager}\cdot\texttt{CreditApproval} \Leftarrow\\ &\qquad\qquad\qquad \texttt{requestApproval}^{\downarrow}!(\mathit{clientChat},\mathit{userId},\mathit{amount},\mathit{risk})\ \ ]\end{aligned}$$

The CreditRequest service mediates the interaction between the bank client, the bank clerk and the bank manager. After receiving the credit request data in message request, the bank clerk is asked to join the conversation, through the join of service RiskAssessment, so as to determine the risk factor of the credit request. The clerk collaboration is achieved by means of the exchange of messages assessRisk and riskVal — we abstract away from the implementation of the RiskAssessment service, and assume its behavior is characterized by type $?\,\mathtt{assessRisk}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2).!\,\mathtt{riskVal}^{\downarrow}(\mathtt{T}_3)$. Depending on the risk factor, the CreditRequest service may either decide automatically to reject the credit request, or it delegates the decision to the bank manager, via the instantiation of service CreditApproval. We assume the extension of the typing rules with the following rule for the **if** statement:

$$\frac{P :: T \quad Q :: T \quad cond : \mathtt{bool}}{\textbf{if } cond \textbf{ then } P \textbf{ else } Q :: T}$$

Thus, to type the particular **if** statement in the service code, both branches must have the same type. We may type the **then** process as follows:

$$\begin{array}{l} \mathtt{requestDenied}^{\downarrow}!() :: \\ \quad \oplus\{!\,\mathtt{requestDenied}^{\downarrow}(); \\ \quad\quad !\,\mathtt{requestApproved}^{\downarrow}().\tau\,\mathtt{orderTransfer}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2).?\,\mathtt{transferDate}^{\downarrow}(\mathtt{T}_4)\} \end{array}$$

which is the type we identified with *managerRole*, hence $\mathtt{requestDenied}^{\downarrow}!() :: managerRole$ (recall that the output is typed in any choice that contains the given output). We then have the following typing, considering the weakening property (Proposition 5.3.3):

$$\mathtt{requestDenied}^{\downarrow}!() :: managerRole \mid Manager : [\star!\,\mathtt{CreditApproval}^{\downarrow}(creditApprovalB)]$$

The **else** process is typed via the rule for the **this** construct. We first show the typing for the continuation process which instantiates the CreditApproval service:

$$\frac{\begin{array}{l} \mathtt{requestApproval}^{\downarrow}!(clientChat, userId, amount, risk) :: \\ \quad !\,\mathtt{requestApproval}^{\downarrow}(managerRole, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3)) \mid clientChat : [managerRole] \\ closed(creditApprovalB \bowtie !\,\mathtt{requestApproval}^{\downarrow}(managerRole, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3)) \end{array}}{\begin{array}{c} \textbf{new}\ Manager \cdot \mathtt{CreditApproval} \Leftarrow \mathtt{requestApproval}^{\downarrow}!(clientChat, userId, amount, risk) :: \\ Manager : [\star!\,\mathtt{CreditApproval}^{\downarrow}(creditApprovalB)] \mid clientChat : [managerRole] \end{array}}$$

where we have that the CreditApproval service conversation is characterized by a closed type, and that the *clientChat* conversation fragment delegated in the requestApproval is characterized by type *managerRole*. We then have the following typing for the **this**:

$$\frac{\begin{array}{c} \textbf{new}\ Manager \cdot \mathtt{CreditApproval} \Leftarrow \mathtt{requestApproval}^{\downarrow}!(clientChat, userId, amount, risk) :: \\ Manager : [\star!\,\mathtt{CreditApproval}^{\downarrow}(creditApprovalB)] \mid clientChat : [managerRole] \end{array}}{\textbf{this}(clientChat).(\cdots) :: Manager : [\star!\,\mathtt{CreditApproval}^{\downarrow}(creditApprovalB)] \mid managerRole}$$

where the type in the current conversation in the conclusion (*managerRole*) results from the merge of the behavior specified for the *clientChat* conversation and the behavior in the current conversation in the premise (in this case *managerRole* and **0** respectively, hence $managerRole \bowtie \mathbf{0}$). We thus have that the type of the **if** statement is the type of the **then** and **else** branches.

Considering the type of service RiskAssessment is $?\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).!\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3)$ we have the following merge at the level of the **join** of the RiskAssessment service:

$$\begin{aligned}&\tau\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\tau\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3).\mathit{managerRole} =\\ &\quad ?\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).!\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3) \bowtie !\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).?\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3).\mathit{managerRole}\end{aligned}$$

We then have that the behavior of the CreditRequest service is captured by the following type:

$$\mathit{creditRequestB} \triangleq ?\,\texttt{request}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\tau\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\tau\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3).\mathit{managerRole}$$

and we have the following typing for the *BankProcess*:

$$\begin{aligned}\mathit{BankProcess} \;::\; & \mathit{BankPortal} : [\star ?\,\texttt{CreditRequest}^{\downarrow}(\mathit{creditRequestB})]\\ & \mid \mathit{Clerk} : [\star !\,\texttt{RiskAssessment}^{\downarrow}(?\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).!\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3))]\\ & \mid \mathit{Manager} : [\star !\,\texttt{CreditApproval}^{\downarrow}(\mathit{creditApprovalB})]\end{aligned}$$

which specifies that the *BankProcess* publishes service CreditRequest, with type *creditRequestB*, at conversation *BankPortal*, and requires services RiskAssessment and CreditApproval to be available at conversations *Clerk* and *Manager*, respectively.

Finally we recall the implementation of the bank client:

$$\begin{aligned}&\mathit{ClientProcess} \triangleq\\ &\quad \mathit{Client} \blacktriangleleft [\\ &\qquad \mathit{ClientTerminal}\\ &\qquad \mid\\ &\qquad \textbf{new}\ \mathit{BankPortal} \cdot \texttt{CreditRequest} \Leftarrow\\ &\mathit{CreditRequestBody}\left\{\begin{array}{l}\texttt{request}^{\downarrow}!(\mathit{myId}, \mathit{amount}).\\ \quad (\texttt{requestApproved}^{\downarrow}?().\texttt{transferDate}^{\downarrow}!(\mathit{date}).\texttt{approved}^{\uparrow}!()\\ \quad +\\ \quad \texttt{requestDenied}^{\downarrow}?().\texttt{denied}^{\uparrow}!())\ \ ]\end{array}\right.\end{aligned}$$

The code specifies the instantiation of service CreditRequest, at conversation *BankPortal*, after which a request message is sent and a notification of the bank's decision is received (either message requestApproved or requestDenied). Afterwards the *ClientTerminal* is notified of the credit request acceptance or rejection (↑ directed messages approved and denied). In case the credit is accepted, and before notifying the *ClientTerminal* process, the service instance code specifies that a transferDate message is sent containing the date of the money transfer. The typing for the body of the CreditRequest service instantiation is as follows:

$$\begin{aligned}&\mathit{CreditRequestBody} ::\\ &\quad !\,\texttt{request}^{\downarrow}().\\ &\qquad \&\{?\,\texttt{requestApproved}^{\downarrow}().!\,\texttt{transferDate}^{\downarrow}().!\,\texttt{approved}^{\uparrow}();\\ &\qquad\quad ?\,\texttt{requestDenied}^{\downarrow}().!\,\texttt{denied}^{\uparrow}()\}\end{aligned}$$

which specifies the output of message request, then a branching that depends upon the reception of message requestApproved or requestDenied, where the process proceeds in the former case by sending messages transferDate and approved, and in the latter case by sending message

`denied`. Notice that `approved` and `denied` messages are directed to the enclosing conversation and will be exchanged with the *ClientTerminal* process. So to determine the client role in the service conversation we must project the type in the $\downarrow$ direction, which yields type:

$$\texttt{!}\,\texttt{request}^{\downarrow}().\ \&\{\texttt{?}\,\texttt{requestApproved}^{\downarrow}().\texttt{!}\,\texttt{transferDate}^{\downarrow}();\ \texttt{?}\,\texttt{requestDenied}^{\downarrow}()\}$$

Instead to characterize the interaction with the *ClientTerminal* process we must project the *CreditRequestBody* type in the $\uparrow$ direction, which yields type: $\oplus\{\texttt{!}\,\texttt{approved}^{\uparrow}();\ \texttt{!}\,\texttt{denied}^{\uparrow}()\}$. So, the service instance actually supplies the external environment (the *Client* conversation) with a choice of one out of two actions that inform on the credit acceptance.

The `CreditRequest` service conversation is characterized by the type obtained from the merge of the service type with the type of the client's role in the interaction. The service type:

$$\begin{aligned}&\mathit{creditRequestB} \triangleq\\ &\quad \texttt{?}\,\texttt{request}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\tau\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\tau\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3).\\ &\qquad \oplus\{\texttt{!}\,\texttt{requestApproved}^{\downarrow}().\tau\,\texttt{orderTransfer}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\texttt{?}\,\texttt{transferDate}^{\downarrow}(\texttt{T}_4);\\ &\qquad\quad \texttt{!}\,\texttt{requestDenied}^{\downarrow}()\}\end{aligned}$$

merged with the client's role in the interaction:

$$\texttt{!}\,\texttt{request}^{\downarrow}().\ \&\{\texttt{?}\,\texttt{requestApproved}^{\downarrow}().\texttt{!}\,\texttt{transferDate}^{\downarrow}();\ \texttt{?}\,\texttt{requestDenied}^{\downarrow}()\}$$

yields the following characterization of the service conversation, a *closed* type:

$$\begin{aligned}&\mathit{creditConversation} \triangleq\\ &\quad \tau\,\texttt{request}^{\downarrow}().\tau\,\texttt{assessRisk}^{\downarrow}().\tau\,\texttt{riskVal}^{\downarrow}().\\ &\qquad \oplus\{\tau\,\texttt{requestApproved}^{\downarrow}().\tau\,\texttt{orderTransfer}^{\downarrow}().\tau\,\texttt{transferDate}^{\downarrow}();\\ &\qquad\quad \tau\,\texttt{requestDenied}^{\downarrow}()\}\end{aligned}$$

which leads to the following typing of the *ClientProcess*:

$$\begin{aligned}&\mathit{ClientProcess} ::\\ &\quad \mathit{Client} : [\oplus\{\tau\,\texttt{approved}^{\downarrow}();\ \tau\,\texttt{denied}^{\downarrow}()\}]\\ &\quad \mid \mathit{BankPortal} : [\texttt{!}\,\texttt{CreditRequest}^{\downarrow}(\mathit{creditRequestB})]\end{aligned}$$

which then specifies the internal choice in the *Client* conversation — assuming the type of the *ClientTerminal* process is $\&\{\texttt{?}\,\texttt{approved}^{\downarrow}();\texttt{?}\,\texttt{denied}^{\downarrow}()\}$ — and the `CreditRequest` service instantiation at conversation *BankPortal*.

Composing the several processes involved in the system we obtain the typing given in Figure 5.8, which characterizes the several message exchanges internal to the system (where the shared input message types are left open to match with further outputs from the external environment). Notice that the type is not *closed* since an unmatched `RiskAssessement` service call is still present. If we were to place this system in parallel with a *ClerkProcess* typed as:

$$\begin{aligned}&\mathit{ClerkProcess} ::\\ &\quad \mathit{Clerk} : [\star\texttt{?}\,\texttt{RiskAssessment}^{\downarrow}(\texttt{?}\,\texttt{assessRisk}^{\downarrow}(\texttt{T}_1,\texttt{T}_2).\texttt{!}\,\texttt{riskVal}^{\downarrow}(\texttt{T}_3))]\end{aligned}$$

$$
\begin{array}{l}
\mathit{ClientProcess} \mid \mathit{BankProcess} \mid \mathit{ManagerProcess} \mid \mathit{BankATMProcess} :: \\
\quad \mathit{Client} : [\oplus\{\tau\, \mathtt{approved}^{\downarrow}();\ \tau\, \mathtt{denied}^{\downarrow}()\}] \\
\quad \mid \mathit{BankPortal} : [\tau\, \mathtt{CreditRequest}^{\downarrow}(\mathit{creditRequestB})] \\
\quad \mid \mathit{BankPortal} : [\star ?\, \mathtt{CreditRequest}^{\downarrow}(\mathit{creditRequestB})] \\
\quad \mid \mathit{Clerk} : [\star !\, \mathtt{RiskAssessment}^{\downarrow}(?\, \mathtt{assessRisk}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2).!\, \mathtt{riskVal}^{\downarrow}(\mathtt{T}_3))] \\
\quad \mid \mathit{Manager} : [\star\tau\, \mathtt{CreditApproval}^{\downarrow}(\mathit{creditApprovalB})] \\
\quad \mid \mathit{Manager} : [\star ?\, \mathtt{CreditApproval}^{\downarrow}(\mathit{creditApprovalB}) \\
\qquad\qquad \mid \star ?\, \mathtt{showRequest}^{\downarrow}(\oplus\{!\, \mathtt{reject}^{\downarrow}();\ !\, \mathtt{accept}^{\downarrow}()\}, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3) \\
\qquad\qquad \mid \star\tau\, \mathtt{showRequest}^{\downarrow}(\oplus\{!\, \mathtt{reject}^{\downarrow}();\ !\, \mathtt{accept}^{\downarrow}()\}, \mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_3)] \\
\quad \mid \mathit{BankATM} : [\star\tau\, \mathtt{CreditTransfer}^{\downarrow}(?\, \mathtt{orderTransfer}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2).?\, \mathtt{transferDate}^{\downarrow}(\mathtt{T}_4))] \\
\quad \mid \mathit{BankATM} : [\star ?\, \mathtt{CreditTransfer}(?\, \mathtt{orderTransfer}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2).?\, \mathtt{transferDate}^{\downarrow}(\mathtt{T}_4)) \\
\qquad\qquad \mid \star\tau\, \mathtt{scheduleTransfer}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_4) \mid \star ?\, \mathtt{scheduleTransfer}^{\downarrow}(\mathtt{T}_1, \mathtt{T}_2, \mathtt{T}_4)]
\end{array}
$$

Figure 5.8: Credit Request Example Typing.

$$
\begin{array}{l}
\mathit{Client} \blacktriangleleft [\\
\quad \textbf{new}\ \mathit{NewsSite} \cdot \mathtt{Newsfeed} \Leftarrow \\
\qquad \textbf{rec}\ \mathcal{X}.\mathrm{post}?(\mathit{info}).\mathcal{X}\ \ ] \\
\mid \\
\mathit{NewsSite} \blacktriangleleft [\\
\quad \star\textbf{def}\ \mathtt{Newsfeed} \Rightarrow \\
\qquad \textbf{rec}\ \mathcal{X}.\textbf{join}\ \mathit{NewsPortal} \cdot \mathtt{NewsService} \Leftarrow \mathcal{X}\ \ ] \\
\mid \\
\mathit{BBC} \blacktriangleleft [\\
\quad \mathit{NewsPortal} \blacktriangleleft [\\
\qquad \star\textbf{def}\ \mathtt{NewsService} \Rightarrow \mathtt{post}!(\mathit{info})\ \ ]\ ] \\
\mid \\
\mathit{CNN} \blacktriangleleft [\\
\quad \mathit{NewsPortal} \blacktriangleleft [\\
\qquad \star\textbf{def}\ \mathtt{NewsService} \Rightarrow \mathtt{post}!(\mathit{info})\ \ ]\ ]
\end{array}
$$

Figure 5.9: The Newsfeed Conversation CC Code.

we would then obtain a system characterized by a *closed* type.

The Newsfeed Conversation

Our next example shows a scenario where an unbounded number of parties may join a single conversation. We consider a `Newsfeed` service that, upon instantiation, asks an undetermined number of news service providers to join the conversation. Each one of the news service providers that join the conversation send a message `post` (containing some news information) that is picked up by the `Newsfeed` service client.

The CC implementation of this scenario is given in Figure 5.9. We define two particular news service providers (*BBC* and *CNN*) but the system is open to an unbounded number of such news providers. Notice that the `Newsfeed` service code continuously calls external news services to join in the conversation, and, in particular, countless copies of *BBC* and *CNN* news services may get to join the conversation. Notice also that the `Newsfeed` service client is continuously able to receive `post` messages, regardless of who is sending them.

The conversation types that capture the `Newsfeed` interaction are shown in Figure 5.10. The

$$
\begin{array}{lcl}
\mathit{NewsfeedConversationT} & \triangleq & \star ?\,\texttt{post}(\mathit{infoT}) \mid \star \tau\,\texttt{post}(\mathit{infoT}) \\
\mathit{NewsfeedClientT} & \triangleq & \star ?\,\texttt{post}(\mathit{infoT}) \\
\mathit{NewsfeedServiceT} & \triangleq & \star !\,\texttt{post}(\mathit{infoT}) \\
\mathit{NewsServiceT} & \triangleq & !\,\texttt{post}(\mathit{infoT})
\end{array}
$$

$$\mathit{Client} :: \mathit{NewsSite} : [!\,\texttt{Newsfeed}([\mathit{NewsfeedServiceT}])]$$

$$
\begin{array}{l}
\mathit{NewsSite} :: \mathit{NewsSite} : [\star ?\,\texttt{Newsfeed}([\mathit{NewsfeedServiceT}])] \\
\qquad\qquad \mid \mathit{NewsPortal} : [\star !\,\texttt{NewsService}([\mathit{NewsServiceT}])]
\end{array}
$$

$$\mathit{BBC} :: \mathit{NewsPortal} : [\star ?\,\texttt{NewsService}([\mathit{NewsServiceT}])]$$

$$\mathit{CNN} :: \mathit{NewsPortal} : [\star ?\,\texttt{NewsService}([\mathit{NewsServiceT}])]$$

$$
\begin{array}{l}
\mathit{NewsfeedSystem} \\
:: \\
\mathit{NewsSite} : [\tau\,\texttt{Newsfeed}([\mathit{NewsfeedServiceT}]) \mid \star ?\,\texttt{Newsfeed}([\mathit{NewsfeedServiceT}])] \\
\mid \\
\mathit{NewsPortal} : [\star \tau\,\texttt{NewsService}([\mathit{NewsServiceT}]) \mid \star ?\,\texttt{NewsService}([\mathit{NewsServiceT}])]
\end{array}
$$

Figure 5.10: The Newsfeed System Typing.

type of the `Newsfeed` conversation (given by *NewsfeedConversationT*) says that infinitely many `post` messages are exchanged, and that the system is still open to receive further `post` messages. Type *NewsfeedConversationT* is split in the types of the `Newsfeed` client and provider, where the first specifies the reception and the second the emission (both infinitely many times) of message `post`. The contribution of each `NewsService` is characterized by type *NewsServiceT* which specifies the output of a single `post` message.

The typings of the four participants individually specify that: the *Client* expects a `Newsfeed` service is available at conversation *NewsSite*; the *NewsSite* publishes a `Newsfeed` service and uses (an infinite number of times) service `NewsService` available at conversation *NewsPortal*; both *BBC* and *CNN* publish `NewsService` in conversation *NewsPortal*. The typing for the whole system (*NewsfeedSystem*) specifies the interactions in services `Newsfeed` and `NewsService` in conversations *NewsSite* and *NewsPortal*, respectively.

5.4 Remarks

Our work shares the same goals of recent developments on multiparty session types, namely the works of Honda et al. [57], Bettini et al. [9] and of Bonelli et al. [11]. To support multiparty interaction, [57] considers multiple session channels, while [9] considers a multiple indexed session channel, both resorting to multiple communication pathways. We follow an essentially different approach, by letting a single medium of interaction support concurrent multiparty interaction via labeled messages. In [9, 57] sessions are established simultaneously between several parties through a multicast session request. As in binary sessions, session delegation is full — a delegating party loses access to the delegated session — so the number of initial participants is kept invariant, unlike in conversations where parties can keep joining in.

The approach of [9, 57] builds on two-level descriptions of service collaborations (global and

local types), first introduced in a theory of endpoint projection [30]. The global types mention the identities of the communicating partners, being the types of the individual participants projections of the global type with respect to these annotations. Our merge operation $\bowtie$ is inspired by the idea of projection [30], but we follow a different approach where "global" and "local" types are treated at the same level in the type language and types do not explicitly mention the participants identities, so that each given protocol may be realized by different sets of participants, provided that the composition of the types of the several participants produces (via $\bowtie$) the appropriate invariant. Our approach thus supports conversations with a dynamically changing number of partners and ensures a higher degree of loose-coupling. We do not see how this could be encoded in the approaches of [9, 57], neither in the approach of [11] to which we can draw a comparison in similar lines and that, moreover, does not seem fit to address decentralized settings since it relies on a one-to-all interaction pattern. On the other hand, we believe that core CC with conversation types can express the same kind of systems as [9, 11, 57].

We view our approach as being more fundamental, with respect to most session-types presentations (including the cited multiparty works), since we are able to accommodate service creation operations as mere programming idioms in the model and avoid their introduction as primitive operations. Such service-oriented operations, implemented via lower-level message passing, admit typing characterizations mechanically derived from the typings of the lower-level implementations, that closely correspond to the expected typings. We are also able to implement and type the conversation join idiom, which we do not see how to represent in existing session-type approaches. We do not claim that our particular implementations for the service-oriented idioms are the best possible implementations, however, we believe that they serve as a proof of concept that such idioms can be represented and typed in a more general setting.

As most behavioral type systems (see [37, 58]), we describe a conversation behavior by some kind of abstract process. However, fundamental ideas behind the conversation type structure, in particular the composition/decomposition of behaviors via merge, as captured, e.g., in the typing rule for parallel composition, and used to model delegation of conversation fragments, have not been explored before. The work of Igarashi et al. [58], in particular, introduces a generic typing framework for π-Calculus systems. Although we do not see how π-Calculus specifications can support (linearly paired) multiparty interactions, we nevertheless comment on some connections. The approach is based on a (parametric) notion of an *ok* predicate that typings must respect, namely the type of a restricted name, which must be invariant under reduction. It may be argued that our *closed* predicate resembles such an *ok* predicate, being also invariant under reduction. Conceivably, there might be instances of the generic framework of [58] (although we are aware of none) that can address the sort of properties we address, not considering the merge of behaviors as we do, and keeping a "raw" type structure that is subject to some posterior analysis to check all the synchronizations are *closed*. However, it does not seem an easy task to develop an *ok* predicate which captures this notion, while being invariant under reduction. To substantiate this claim, consider the following process:

$$
\begin{array}{l}
(\boldsymbol{\nu} session) \\
(\ \mathtt{speak}!(session) \mid \mathtt{listen}!(session)\ ) \\
\ \mid \\
\mathtt{speak}?(x).\,\mathtt{listen}?(y).\,x \blacktriangleleft [\,\mathtt{hi}!().\,y \blacktriangleleft [\,\mathtt{hi}?()\,]\,]
\end{array}
$$

The processes on top are willing to send the same channel name *session* in messages `speak` and `listen`. The process on the bottom specifies the reception of two names in such messages, and then specifies an output over the conversation which name is received first (x), and afterwards an input over the conversation which name is received second (y). The names received are typed as $x : [!\,\texttt{hi}()]$ and $y : [?\,\texttt{hi}()]$. Therefore, we have the following typing for conversation *session* when analyzing the processes on top:

$$\texttt{speak}!(session) \mid \texttt{listen}!(session) :: session : [\tau\,\texttt{hi}()] \mid \ldots$$

where the τ results from the merge $!\,\texttt{hi}() \bowtie ?\,\texttt{hi}()$. Instead, adopting a typing strategy such as the one of [58] we type *session* with $!\,\texttt{hi}() \mid ?\,\texttt{hi}()$. A possible *ok* predicate might then say "type $!\,\texttt{hi}() \mid ?\,\texttt{hi}()$ is *ok* since we can synchronize the actions and reduce it to the inaction type". However, the system evolves, in two steps, to the configuration:

$$session \blacktriangleleft [\, \texttt{hi}!().\, session \blacktriangleleft [\, \texttt{hi}?()\,]\,]$$

where we recover the same conversation type as before via the merge in the conclusion of rule (*Piece*):

$$\frac{\texttt{hi}!().\, session \blacktriangleleft [\, \texttt{hi}?()\,] :: !\,\texttt{hi}() \mid session : [?\,\texttt{hi}()]}{session \blacktriangleleft [\, \texttt{hi}!().\, session \blacktriangleleft [\, \texttt{hi}?()\,]\,] :: session : [!\,\texttt{hi}()] \bowtie session : [?\,\texttt{hi}()]}$$

thus allowing for *closed* to still hold at the level of the restriction, while in the typing strategy of [58] we obtain a type that specifies the two actions in sequence, e.g.,:

$$session \blacktriangleleft [\, \texttt{hi}!().\, session \blacktriangleleft [\, \texttt{hi}?()\,]\,] :: session : [!\,\texttt{hi}().?\,\texttt{hi}()]$$

where the type of session is not *ok* as it does not reduce to inaction (it does not reduce at all), so this informally described *ok* predicate does not seem invariant under reduction. Although the typing description that best fits the process is perhaps the second alternative (the process is indeed deadlocked), this example illustrates the flexibility introduced by the merge in our typing characterizations. In fact, similar kind of examples are found in the session types literature to justify heavy restrictions to analysis techniques that address progress properties (namely, the continuation of an input uses exclusively the received name — see [40, 80]), showing that this sort of scenarios are, in general, not easy to deal with.

Without this flexibility we would not be able to address some of the challenging scenarios discussed, where conversation fragments are passed around in messages in a flexible way, and therefore we do not see how our approach could be reproduced in an instance of [58]. Naturally, we would prefer to exclude such deadlocked systems in the type system directly, but not at the cost of losing our generality. To address the deadlock absence issue in particular, we introduce a complementary technique to the conversation type system that singles out CC systems that enjoy a progress property, described in the next chapter.

Chapter 6

Proving Progress of Conversations

In this section we present the progress proof system, an analysis technique we introduced in [27, 28] so as to guarantee systems enjoy a progress property. While the conversation type system allows us to guarantee that conversations follow the prescribed protocols, it is not enough to guarantee that the systems do not get stuck, due to, for instance, communication dependencies between distinct conversations. We start by presenting the main ideas behind our analysis technique that allow us to address challenging configurations that fall out of scope of other approaches, then we present the technical artifacts that instantiate these ideas so as to characterize systems that have a lock-free communication structure — namely, event orderings. We proceed by showing a proof system that associates Conversation Calculus processes to event orderings that then allows us to characterize their communication dependency structure, and show the results that can be proved for processes characterized by such event orderings. For the sake of illustration we show an example derivation in our proof system. Also, we show how our technique can be used over systems modeled in π_{lab}-Calculus, so as to demonstrate the generality of the approach. At the end of the chapter we present some remarks on related work.

6.1 Progress of Dynamic Conversations

In this section, we present the main ideas behind our proof system, developed so as to enforce progress properties on systems. As most traditional deadlock detection methods (e.g., see [40, 63, 73, 98]), we build on the construction of a well-founded ordering on events.

Consider the following example of a stuck system:

$$chatA \blacktriangleleft [\, \mathtt{hello}!().\, chatB \blacktriangleleft [\, \mathtt{hello}!() \,]\,] \mid chatB \blacktriangleleft [\, \mathtt{hello}?().\, chatA \blacktriangleleft [\, \mathtt{hello}?() \,]\,]$$

Although the conversation protocols — captured by the conversation type $\tau\,\mathtt{hello}()$ for both *chatA* and *chatB* — are followed, the system is stuck since the dependencies between the two conversations are exercised inversely by the two parallel processes. By looking at the process on the left hand side we conclude that the exchange of message `hello` in conversation *chatB* can only take place after the exchange of message `hello` in conversation *chatA*. On the other hand, the process on the right hand side tells us the exact inverse order: the exchange of message `hello` in conversation *chatA* can only take place after the exchange of message `hello` in conversation *chatB*. Therefore, we cannot order the message exchanges — *events* — in a well-founded way.

If we were to state that the exchange of message `hello` in conversation *chatA* comes first, and

only afterwards the exchange of message *hello* in conversation *chatB* can take place, which we capture by saying event $chatA.\mathtt{hello}$ is smaller than event $chatB.\mathtt{hello}$, then we would exclude the process on the right hand side (and consequently, we would also exclude the whole system) as it does not respect the predefined ordering of events. The key process construct where we must verify such orderings are respected is the prefixed process, since the action prefix is blocking all the actions in its continuation, and therefore inducing an ordering. Intuitively, we need to make sure that if a prefix is blocking some action A then it better be the case that action A is not blocking the dual action of the prefix, otherwise the system is stuck. In other words, we need to verify that the events specified in the continuation of a prefix are of greater rank with respect to the event relative to the prefix itself. In the example above, considering the ordering $chatA.\mathtt{hello} \prec chatB.\mathtt{hello}$, we can verify that the process:

$$chatA \blacktriangleleft [\, \mathtt{hello}!().\, chatB \blacktriangleleft [\, \mathtt{hello}!() \,]\,]$$

is well ordered since the event in the continuation ($chatB.\mathtt{hello}$) is of greater rank with respect to the event relative to the prefix ($chatA.\mathtt{hello}$). On the other hand, the process:

$$chatB \blacktriangleleft [\, \mathtt{hello}?().\, chatA \blacktriangleleft [\, \mathtt{hello}?() \,]\,]$$

is not well ordered as the event in the continuation ($chatA.\mathtt{hello}$) is of lesser rank with respect to the event relative to the prefix ($chatB.\mathtt{hello}$).

This example already allowed us to introduce the basic idea behind our approach, which is to verify the events in a process can be ordered in a well-founded way. Since we are typically interested in more challenging configurations, let us look at the purchase service collaboration (considering the translation of the service-oriented idioms to their lower-level implementations):

$$
\begin{array}{l}
Buyer \blacktriangleleft [\, (\boldsymbol{\nu} c)(Seller \blacktriangleleft [\, \mathtt{BuyService}^{\downarrow}!(c) \,] \mid c \blacktriangleleft [\, \mathtt{buy}^{\downarrow}!(prod).\mathtt{price}^{\downarrow}?(p).\mathtt{details}^{\downarrow}?(d) \,]\,] \\
\mid \\
Seller \blacktriangleleft [\, PriceDB \mid \\
\qquad \mathtt{BuyService}^{\downarrow}?(x).x \blacktriangleleft [\, \mathtt{buy}^{\downarrow}?(prod).\mathtt{askPrice}^{\uparrow}!(prod). \\
\qquad\qquad \mathtt{priceVal}^{\uparrow}?(p).\mathtt{price}^{\downarrow}!(p). \\
\qquad\qquad \mathbf{this}(y).(\, Shipper \blacktriangleleft [\, \mathtt{DeliveryService}^{\downarrow}!(y) \,] \\
\qquad\qquad\quad \mid \mathtt{product}^{\downarrow}!(prod) \,)\,]\,] \\
\mid \\
Shipper \blacktriangleleft [\, \mathtt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\, \mathtt{product}^{\downarrow}?(p).\mathtt{details}^{\downarrow}!(data) \,]\,]
\end{array}
$$

The code models a three-party purchase collaboration, where the seller party actually interleaves the service conversation (x) with conversation *Seller*, to access the resource database, and with conversation *Shipper* so as to allow for it to join the ongoing service conversation. The challenge here is how to statically account for the orderings of events on conversations which will only be dynamically instantiated. For instance, the dependency between events $x.\mathtt{buy}$ and $Seller.\mathtt{askPrice}$ and between events $x.\mathtt{price}$ and $Shipper.\mathtt{DeliveryService}$ depends on the name that will instantiate x upon `BuyService` instantiation. In other words, how do we account for the ordering of received names?

Previous approaches on progress for session types do not cope with such configurations,

namely the approaches of Dezani-Ciancaglini et al. [40] and of Bettini et al. [9], where this kind of systems is excluded in their analysis, i.e., in their approach it is not possible to address systems where processes interleave received names. However, such interleaving seems to be crucial to allow for service instances to access local resources or to call external services.

To solve this problem we attach to our events a notion of prescribed ordering: the ordering that captures the event ordering expected by the receiving process, that the emitted name will have to comply to. In such way, we are able to statically determine the orderings followed by the processes at "runtime", through propagation of orderings in the analysis of message exchanges that carry conversation identifiers. Technically, we proceed by developing a notion of event and of event ordering that allow us to verify that CC processes can be ordered in a well-founded way, including when conversation references are passed around. In the next section we describe the formalisms that instantiate these ideas.

6.2 Event Ordering

In this section we present the formalisms that allow us to characterize the ordering of events in processes. In the case of CC processes, events are message synchronizations taking place in conversations. Thus, an event is identified by a conversation name and a message label. Also, since messages carry conversation references, an event also accounts for the *prescribed* ordering for the name being passed in the message.

Definition 6.2.1 (Event orderings, Parameterized Event Ordering and Events) *We say relation Γ between events is an event ordering if it is a well-founded partial order of events. We denote by $(x)\Gamma$ an event ordering which is parameterized by x, where $x \in \mathcal{V}$. Events, noted e, are defined as follows:*

$$e, e_1, \ldots ::= \mathsf{n}.l.(x)\Gamma \quad \text{(Event)}$$

where $\mathsf{n} \in \Lambda \cup \mathcal{V} \cup \{\mathsf{here}, \mathsf{up}\}$ and $l \in \mathcal{L}$ and $(x)\Gamma$ is a parameterized event ordering.

Event orderings capture the overall ordering of events. Parameterized event orderings are used to capture the prescribed ordering of conversation fragments that are passed in messages. Events describe a message exchange by identifying the conversation (by its name, variable, or by special identifiers here and up that represent the anonymous top level conversations), the label of the message, and the prescribed ordering for the conversation name passed in the message.

To order our events we consider the tuple (conversation identifier,message label), which allows us to cope with systems with multiple interleaved conversations, and back and forth communications between two or more conversations in the same thread. To order the conversation references that are passed in message synchronizations, each event in the ordering also informs on the ordering associated to the conversation which is to be communicated in the message. These ingredients will then allow us to check that all events in the continuation of a prefix are of greater rank than the event of the prefix, thus guaranteeing the event dependencies are acyclic.

In the next section we show some operations that manipulate event orderings that will be used when we characterize the ordering of events in CC processes, along with some auxiliary notions used to order recursive processes and to locate events in CC systems.

6.2.1 Event Ordering Operators

We define some useful operations over event orderings. The two main operations allow us to explain name hiding, and verify the ranks of the communications in the continuations of prefixes. For starters, we introduce some notation helpful to describe how two events in particular are related by a given ordering, an abbreviation to denote events defined on a determined name and a notation that elides empty event orderings in events.

Notation 6.2.2 *We introduce the following notation conventions:*

- *By* $e_1 \prec_\Gamma e_2$ *we denote that* e_1 *is smaller than* e_2 *in* Γ.
- *By* $e_1 \prec^*_\Gamma e_k$ *we denote that there is* $e_2, \ldots, e_{k-1}$ *such that* $e_1 \prec_\Gamma e_2 \prec_\Gamma \ldots \prec_\Gamma e_{k-1} \prec_\Gamma e_k$.
- *By* $e(\mathsf{n})$ *we denote an event of conversation* n*, hence of the form* $\mathsf{n}.l.(x)\Gamma$*, for some* $l, (x)\Gamma$.
- *We use* $\mathsf{n}.l$ *to abbreviate* $\mathsf{n}.l.(x)\Gamma$ *where* $\Gamma = \emptyset$.

We now define some basic operators over event orderings and events, namely: the *domain* of an event ordering, the set of *free conversation identifiers*, *free names* and *free variables* of events and event orderings, and *conversation identifier substitution* over event orderings and events.

Definition 6.2.3 (Event Ordering Domain)
The domain of an event ordering Γ*, noted* $dom(\Gamma)$*, is the set of events which are related by* Γ*, defined as follows:*

$$dom(\Gamma) \triangleq \{e \mid \exists e'.(e \prec_\Gamma e') \text{ or } (e' \prec_\Gamma e)\}$$

Definition 6.2.4 (Event Ordering Free Conversation Identifiers)
We denote by $fids(\Gamma)$ *and by* $fids(e)$ *the sets of free conversation identifiers of event ordering* Γ *and of event* e*, respectively, defined as follows:*

$$\begin{array}{rcl} fids(\Gamma) & \triangleq & \{\mathsf{n} \mid \mathsf{n} \in fids(e) \wedge e \in dom(\Gamma)\} \\ fids(\mathsf{m}.l.(x)\Gamma') & \triangleq & \{\mathsf{m}\} \cup \{\mathsf{n} \mid \mathsf{n} \in fids(\Gamma') \wedge \mathsf{n} \neq x\} \end{array}$$

Definition 6.2.5 (Event Ordering Free Names)
We denote by $fn(\Gamma)$ *the set of free names of event ordering* Γ *defined as* $fn(\Gamma) \triangleq fids(\Gamma) \cap \Lambda$.

Definition 6.2.6 (Event Ordering Free Variables)
We denote by $fv(\Gamma)$ *the set of free variables of event ordering* Γ *defined as* $fv(\Gamma) \triangleq fids(\Gamma) \cap \mathcal{V}$.

Definition 6.2.7 (Event Ordering Conversation Identifier Substitution)
We denote by $\Gamma\{\mathsf{n}/\mathsf{m}\}$ *the event ordering and by* $e\{\mathsf{n}/\mathsf{m}\}$ *the event obtained by replacing all free occurrences of conversation identifier* n *by conversation identifier* m *in event ordering* Γ *and in*

event e, respectively, defined as follows:

$$
\begin{array}{rcl}
\Gamma\{\mathsf{n}/\mathsf{m}\} & \triangleq & \{e_1\{\mathsf{n}/\mathsf{m}\} \prec e_2\{\mathsf{n}/\mathsf{m}\} \mid e_1 \prec_\Gamma e_2\} \\
(\mathsf{n}.l.(x)\Gamma)\{\mathsf{n}/\mathsf{m}\} & \triangleq & \mathsf{m}.l.(x)\Gamma' \\
& & (\mathit{where}\ \Gamma' = \{e_1\{\mathsf{n}/\mathsf{m}\} \prec e_2\{\mathsf{n}/\mathsf{m}\} \mid e_1 \prec_\Gamma e_2\}\ \mathit{and}\ \mathsf{n} \neq x \neq \mathsf{m}) \\
(\mathsf{o}.l.(x)\Gamma)\{\mathsf{n}/\mathsf{m}\} & \triangleq & \mathsf{o}.l.(x)\Gamma' \\
& & (\mathit{where}\ \mathsf{o} \neq \mathsf{n}\ \mathit{and}\ \Gamma' = \{e_1\{\mathsf{n}/\mathsf{m}\} \prec e_2\{\mathsf{n}/\mathsf{m}\} \mid e_1 \prec_\Gamma e_2\}\ \mathit{and}\ \mathsf{n} \neq x \neq \mathsf{m})
\end{array}
$$

Next we introduce the name hiding operation that removes from an ordering Γ all events relative to some given conversation n, noted $\Gamma \setminus \mathsf{n}$. This operation will be useful to characterize name restriction in CC processes, since our orderings only talk about the order of communications in visible conversations.

Definition 6.2.8 (Event Ordering Conversation Identifier Hiding)
We denote by $\Gamma \setminus \mathsf{n}$ the event ordering obtained by removing all events defined on n in event ordering Γ, defined as follows:

$$
\begin{array}{rcl}
\Gamma \setminus \mathsf{n} & \triangleq & \{(e(\mathsf{m}) \prec e(\mathsf{o})) \mid (e(\mathsf{m}) \prec_\Gamma e(\mathsf{o})) \wedge \mathsf{m} \neq \mathsf{n} \neq \mathsf{o}\} \cup \\
& & \{(e(\mathsf{m}) \prec e(\mathsf{o})) \mid (e(\mathsf{m}) \prec_\Gamma e_1(\mathsf{n}) \prec_\Gamma \ldots \prec_\Gamma e_k(\mathsf{n}) \prec_\Gamma e(\mathsf{o})) \wedge \mathsf{m} \neq \mathsf{n} \neq \mathsf{o}\}
\end{array}
$$

We consider $\Gamma \setminus \mathsf{n}$ to be undefined when $\mathsf{n} \in \mathit{fids}(\Gamma \setminus \mathsf{n})$.

Essentially, the name hiding of n in event ordering Γ relates all events already in the initial ordering Γ, except those defined on the name to hide (n), closing, by transitivity, the ordering chains where events defined on n occur. The set of events related by $\Gamma \setminus \mathsf{n}$, its domain, is then the domain of Γ minus the events that are defined on n.

The operation we introduce next allows us to characterize the subrelation minored by a given event e, which will then be used in the rules for action prefixes to check that continuations specify interactions in events of greater rank than the event associated to the prefix.

Definition 6.2.9 (Event Ordering Subrelation Minored by e)
Given event e and event ordering Γ such that $e \in \mathit{dom}(\Gamma)$ we define $e \perp \Gamma$ as the subrelation of Γ where all events are greater than e, as follows:

$$
e \perp \Gamma \triangleq \{(e_1 \prec e_2) \mid (e_1 \prec_\Gamma e_2) \wedge (e \prec_\Gamma^* e_1)\}
$$

We consider $e \perp \Gamma$ to be undefined when $e \notin \mathit{dom}(\Gamma)$.

Intuitively, if we are to view orderings as trees, then the subrelation operation corresponds to taking the tree under a specific node in the tree — the minor event. Notice the minoring element is required to belong to the domain of the event ordering, otherwise the subrelation is undefined.

6.2.2 Ordering Recursive Behavior

To represent the ordering of recursive processes our system makes use of an auxiliary environment that associates recursion variables to event orderings. To motivate the introduction of such

environment consider the following specification of a recursive process:

$$a \blacktriangleleft \left[\textbf{rec}\, \mathcal{X}.\, \texttt{hello}^{\downarrow}!().\, \texttt{bye}^{\downarrow}!().\mathcal{X}\right]$$

where we assume that event $a.\texttt{hello}$ is smaller than $a.\texttt{bye}$. If such ordering is respected throughout the system, then no process waiting on `bye` will only after receiving on `bye` activate an input over `hello`. Also, the process will always respect such ordering in each of its iterations. However, if we are to look at the one-step unfolding of the process:

$$a \blacktriangleleft \left[\texttt{hello}^{\downarrow}!().\, \texttt{bye}^{\downarrow}!().\textbf{rec}\, \mathcal{X}.\, \texttt{hello}^{\downarrow}!().\, \texttt{bye}^{\downarrow}!().\mathcal{X}\right]$$

we verify that it is no longer well ordered under the principle that actions in the continuation of a prefix are of greater rank with respect to the prefix. To support such safe specifications we consider that unfoldings are annotated $\textbf{rec}_t$ so as to identify them as unfoldings. Hence, we write the process above as:

$$a \blacktriangleleft \left[\texttt{hello}^{\downarrow}!().\, \texttt{bye}^{\downarrow}!().\textbf{rec}_t\, \mathcal{X}.\, \texttt{hello}^{\downarrow}!().\, \texttt{bye}^{\downarrow}!().\mathcal{X}\right]$$

Then, we will verify that such unfoldings are well-ordered with respect to the ordering initially considered for the **rec** process. To capture this, we introduce a notion of recursion environment which is a set of associations of recursion variables and event orderings, defined next.

Definition 6.2.10 (Recursion Environment)
A recursion environment, noted Δ, is a set of associations between recursion variables and event orderings defined as follows:

$$\Delta \triangleq \mathcal{X}_1 \to \Gamma_1, \ldots, \mathcal{X}_k \to \Gamma_k$$

We introduce an abbreviation useful to simplify presentation, define the domain of a recursion environment and lift the name hiding operation to recursion environments.

Notation 6.2.11 *We denote by $\Delta(\mathcal{X}) = \Gamma$ that recursion environment Δ is such that:*

$$\Delta = \mathcal{X}_1 \to \Gamma_1, \ldots, \mathcal{X} \to \Gamma, \ldots, \mathcal{X}_k \to \Gamma_k$$

Definition 6.2.12 (Recursion Environment Domain)
We denote by $dom(\Delta)$ the domain of the recursion environment Δ, defined as follows:

$$dom(\Delta) \triangleq \{\mathcal{X}_1, \ldots, \mathcal{X}_k\} \quad (\textit{if } \Delta = \mathcal{X}_1 \to \Gamma_1, \ldots, \mathcal{X}_k \to \Gamma_k)$$

Definition 6.2.13 (Recursion Environment Conversation Identifier Hiding)
We denote by $\Delta \setminus \mathsf{n}$ the recursion environment obtained by applying the name hiding to all event orderings in Δ, defined as follows:

$$\Delta \setminus \mathsf{n} \triangleq \mathcal{X}_1 \to (\Gamma_1 \setminus \mathsf{n}), \ldots, \mathcal{X}_k \to (\Gamma_k \setminus \mathsf{n}) \quad (\textit{if } \Delta = \mathcal{X}_1 \to \Gamma_1, \ldots, \mathcal{X}_k \to \Gamma_k)$$

We introduce some notation useful to simplify presentation and define a predicate that certifies that the combination of an event ordering and a recursion environment is well founded.

Notation 6.2.14 *We denote by* $\bigcup \Delta$ *the event ordering obtained by the union of all event orderings that occur in* Δ*, hence:* $\bigcup \Delta \triangleq \Gamma_1 \cup \ldots \cup \Gamma_k$ *for* $\Delta = \mathcal{X}_1 \rightarrow \Gamma_1, \ldots, \mathcal{X}_k \rightarrow \Gamma_k$*.*

Definition 6.2.15 (Well-Founded Event Ordering and Recursion Environment)
We say that the combination of an event ordering Γ *and a recursion environment is well founded, noted* $wf(\Gamma, \Delta)$*, if* $\Gamma \cup \bigcup \Delta$ *is a well-founded relation.*

Such predicate will be used in our rules so as to ensure that our judgements always rely on a (overall) well-founded ordering.

6.2.3 Locating Events

When characterizing the ordering of CC processes we need to keep track of the identities of the current and enclosing conversations. To carry such information we define the notion of locator (ℓ) and introduce some notation useful to simplify presentation.

Definition 6.2.16 (Locator)
We denote by ℓ *a pair of conversation identifiers* (n, m) *where* $\mathsf{n}, \mathsf{m} \in \Lambda \cup \mathcal{V} \cup \{\mathsf{here}, \mathsf{up}\}$*.*

A locator ℓ then identifies the current and enclosing conversations, through their names or variables, and also through the here and up identifiers that represent the special cases of process top level, where the $\downarrow$ and $\uparrow$ conversations are anonymous.

Notation 6.2.17 *If* $\ell = (\mathsf{n}, \mathsf{m})$ *then we use* $\ell(\uparrow)$ *and* $\ell(\downarrow)$ *to refer to* n *and* m*, respectively.*

Given these artifacts and operations over events, we may present our proof system that singles out Conversation Calculus systems that enjoy a well-founded ordering of events.

6.3 Progress Proof System

In this section we present the progress proof system that associates event orderings to CC processes, so as to guarantee the communication structure of processes is well founded. The proof system is presented by means of judgments of the form $\Gamma; \Delta \vdash_\ell P$. The judgment $\Gamma; \Delta \vdash_\ell P$ states that the communications of process P follow a well determined order, given by event ordering Γ and recursion environment Δ, where ℓ keeps track of the identities of the current and enclosing conversations of P. We start by informally presenting the rules and then formally present the proof system.

The rule for parallel composition is as follows:

$$\frac{\Gamma; \Delta \vdash_\ell P \quad \Gamma; \Delta \vdash_\ell Q}{\Gamma; \Delta \vdash_\ell P \mid Q}$$

which specifies that the parallel composition is well ordered by the ordering that orders both branches of the parallel composition. Since the parallel branches are simultaneously active, the

ordering for parallel composition needs only to make sure that the ordering for the two branches is consistent, hence, the same. The rule for the inactive process:

$$\frac{wf(\Gamma, \Delta)}{\Gamma; \Delta \vdash_\ell \mathbf{0}}$$

specifies that any (well-founded) event ordering/recursion environment $\Gamma; \Delta$ orders such process. The next rule explains the ordering for name restriction:

$$\frac{\Gamma; \Delta \vdash_\ell P}{\Gamma \setminus a; \Delta \setminus a \vdash_\ell (\boldsymbol{\nu} a)P}$$

Essentially, the rule says that if the scope of the name restriction is well-ordered by $\Gamma; \Delta$, then we can hide a in $\Gamma; \Delta$ in the conclusion of the rule, since no other communication dependencies will have to be verified for events defined over a in the outer environment, given that the name is restricted — communication of the name to the outer environment is handled at the level of the output. The rule for the conversation piece is as follows:

$$\frac{\Gamma; \Delta \vdash_{(\ell(\downarrow), n)} P}{\Gamma; \Delta \vdash_\ell n \blacktriangleleft [P]}$$

where the premise specifies that n is the current conversation and $\ell(\downarrow)$ is the enclosing conversation, which is the current conversation in the conclusion.

The following rules for input and output prefixes ensure that communications originating in the continuations, including the ones in the conversation being received/sent, are of a greater rank. The rule for input prefix is specified as follows:

$$\frac{\forall_{i \in I}((\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y_i/x_i\}; \Delta \vdash_\ell P_i \quad \Gamma'_i \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma)) \quad wf(\Gamma, \Delta)}{\Gamma; \Delta \vdash_\ell \Sigma_{i \in I}\, l_i^d?(x_i).P_i}$$

The rule takes the events associated to the input prefixes $\ell(d).l_i.(y_i)\Gamma'_i$ (for each $i \in I$), where $\ell(d)$ gives the identity of the conversation in which the message exchange will take place, and checks that the continuation of the prefix (P_i) is well ordered. The continuations are checked to be ordered by the relation that is minored by the event $\ell(d).l_i.(y_i)\Gamma'_i$, denoted by $\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma$, combined with the ordering prescribed for the name which is to be received in the input (Γ'_i), where the event ordering parameter (y_i) is replaced by the variable bound in the input (x_i), and recursion environment Δ. Extending the ordering with $\Gamma'_i\{y_i/x_i\}$ thus allows for the continuation to participate in events in the received name x_i, accordingly to the prescribed ordering.

If this is the case for all input prefixes and $\Gamma; \Delta$ is well founded, then the input guarded summation process is well-ordered by $\Gamma; \Delta$. Notice that the events $\ell(d).l_i.(y_i)\Gamma'_i$ are in the domain of Γ, otherwise the subrelations $\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma$ are undefined. Notice also that the parameterized event orderings $(y_i)\Gamma'_i$ are contained in the ordering $\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma$, excluding the events specific to the parameter y_i. The next rule orders the output prefix:

$$\frac{(\ell(d).l.(x)\Gamma' \perp \Gamma); \Delta \vdash_\ell P \quad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \perp \Gamma) \quad wf(\Gamma, \Delta)}{\Gamma; \Delta \vdash_\ell l^d!(n).P}$$

Analogously to the rule for the input prefix, we take the event associated to the prefix $\ell(d).l.(x)\Gamma'$ and verify that the continuation is ordered by events greater than $\ell(d).l.(x)\Gamma'$. Also, and cru-

cially, we check that the name being passed in the output complies with the ordering prescribed in the event $(x)\Gamma'$ by verifying that such prescribed ordering, where the variable x is replaced by the name to be sent n, is contained in the ordering of events greater than $\ell(d).l.(x)\Gamma'$. In such way, we can be sure that after the name is passed the overall ordering is still respected.

The following rule orders the conversation awareness primitive:

$$\frac{\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{x/\ell(\downarrow)\} \prec_\Gamma e_2\{x/\ell(\downarrow)\})\}; \Delta \vdash_\ell P \quad (\ell(\downarrow) \neq \textsf{here})}{\Gamma; \Delta \vdash_\ell \textbf{this}(x).P}$$

The rule ensures interactions in conversations x follow the ordering defined for the current conversation ($\ell(\downarrow)$). The event ordering considered in the premise is enlarged with events defined on conversation x, according to the ordering specified for events of conversation $\ell(\downarrow)$. Condition $\ell(\downarrow) \neq$ here guarantees the **this** is inside some conversation piece (not at toplevel) as otherwise the process would be stuck — cf., the semantics of **this**.

The next rules order the recursive process definition and recursion variable:

$$\frac{\mathcal{X} \notin \mathit{dom}(\Delta) \quad \Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P}{\Gamma; \Delta \vdash_\ell \textbf{rec}\, \mathcal{X}.P} \qquad \frac{\mathcal{X} \in \mathit{dom}(\Delta) \quad \mathit{wf}(\Gamma, \Delta)}{\Gamma; \Delta \vdash_\ell \mathcal{X}}$$

$$\frac{\Delta(\mathcal{X}); \Delta \vdash_\ell P \quad \mathit{fv}(\Delta(\mathcal{X})) = \emptyset \quad \mathit{wf}(\Gamma, \Delta)}{\Gamma; \Delta \vdash_\ell \textbf{rec}_t\, \mathcal{X}.P}$$

The rule for the recursion definition **rec** says that the process is well-ordered if the body of the recursion is well-ordered by the same ordering, but where the recursion environment considers an extra association of the current recursion variable with the current event ordering. Then, we may order the body of the recursion up to the point a recursion variable is found, at which stage we verify that such variable is in the domain of the recursion environment and ensure that the Γ and Δ mentioned in the conclusion are well founded. Finally, to order the unfolded recursion definition, we ensure that the body is well-ordered by the event ordering associated to the respective recursion variable, and ensure that the combination of Γ and Δ mentioned in the conclusion is well founded. Given this basic intuition over the rules that characterize the ordering of CC processes, we may now define the notion of well-ordered process.

Definition 6.3.1 (Well-Ordered Process)
We say process P is well-ordered by Γ and Δ, noted $\Gamma; \Delta \vdash_\ell P$, if $\Gamma; \Delta \vdash_\ell P$ can be derived using the rules given in Figure 6.1.

Next we show our results that guarantee that well-ordered processes enjoy a progress property.

6.3.1 Progress Property

In this section we present our progress results, which guarantee that the well-ordered property of processes is invariant under process reduction, and that well-ordered (and well-typed) processes never reach deadlocked configurations. We first characterize a *well-formedness* condition, which filters out some undesired configurations.

Definition 6.3.2 (Well-Formed Process)
We say process P is well-formed if for any occurrence of a recursive process **rec** $\mathcal{X}.Q$ *in P then*

$$\frac{\Gamma;\Delta \vdash_\ell P \quad \Gamma;\Delta \vdash_\ell Q}{\Gamma;\Delta \vdash_\ell P \mid Q}(Par) \qquad \frac{wf(\Gamma,\Delta)}{\Gamma;\Delta \vdash_\ell \mathbf{0}}(Stop) \qquad \frac{\Gamma;\Delta \vdash_\ell P}{\Gamma \setminus a;\Delta \setminus a \vdash_\ell (\boldsymbol{\nu} a)P}(Res)$$

$$\frac{\mathcal{X} \notin dom(\Delta) \quad \Gamma;(\Delta,\mathcal{X}\to\Gamma) \vdash_\ell P}{\Gamma;\Delta \vdash_\ell \mathbf{rec}\,\mathcal{X}.P}(Rec) \qquad \frac{\mathcal{X} \in dom(\Delta) \quad wf(\Gamma,\Delta)}{\Gamma;\Delta \vdash_\ell \mathcal{X}}(RecVar)$$

$$\frac{\Delta(\mathcal{X});\Delta \vdash_\ell P \quad fv(\Delta(\mathcal{X})) = \emptyset \quad wf(\Gamma,\Delta)}{\Gamma;\Delta \vdash_\ell \mathbf{rec}_t\,\mathcal{X}.P}(RecTerm) \qquad \frac{\Gamma;\Delta \vdash_{(\ell(\downarrow),n)} P}{\Gamma;\Delta \vdash_\ell n \blacktriangleleft [P]}(Piece)$$

$$\frac{\forall_{i\in I}((\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y_i/x_i\};\Delta \vdash_\ell P_i \quad \Gamma'_i \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma)) \quad wf(\Gamma,\Delta)}{\Gamma;\Delta \vdash_\ell \Sigma_{i\in I}\, l_i^d?(x_i).P_i}(Input)$$

$$\frac{(\ell(d).l.(x)\Gamma' \perp \Gamma);\Delta \vdash_\ell P \quad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \perp \Gamma) \quad wf(\Gamma,\Delta)}{\Gamma;\Delta \vdash_\ell l^d!(n).P}(Output)$$

$$\frac{\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{x/\ell(\downarrow)\} \prec_\Gamma e_2\{x/\ell(\downarrow)\})\};\Delta \vdash_\ell P \quad (\ell(\downarrow) \neq \mathsf{here})}{\Gamma;\Delta \vdash_\ell \mathbf{this}(x).P}(This)$$

Figure 6.1: Progress Proof Rules.

no recursive process occurs in Q, the occurrences of $\mathcal{X}$ in Q are guarded by a prefix, and such variables $\mathcal{X}$ do not occur in parallel processes, neither within the scope of a conversation piece.

Remark 6.3.3 *We may show that well-typed processes are well-formed processes, however we introduce the well-formedness condition so as to leave the preservation of the ordering as an independent result with respect to the conversation type system.*

Convention 6.3.4 *To simplify presentation we consider that transition rule (Rec) of Figure 4.2 introduces $\mathbf{rec}_t$ in the unfolded recursive process, and such $\mathbf{rec}_t$ is afterwards conservatively kept:*

$$\frac{P\{\mathcal{X}/\mathbf{rec}_t\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathbf{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q} \qquad \frac{P\{\mathcal{X}/\mathbf{rec}_t\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathbf{rec}_t\,\mathcal{X}.P \xrightarrow{\lambda} Q}$$

Also, we consider all recursion variables are distinct and that the recursion variable associated to a $\mathbf{rec}_t$ is not subject to α-conversion.

Our first result states a weakening property that captures the fact that a process well ordered by a given event ordering is also well ordered by any event ordering that contains the initial one.

Proposition 6.3.5 (Event Ordering Weakening)
Let P be a process such that $\Gamma;\Delta \vdash_\ell P$. If $wf(\Gamma',\Delta')$ and $\Gamma \subseteq \Gamma'$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$ then $\Gamma';\Delta' \vdash_\ell P$.

Proof. By induction on the length of the derivation of $\Gamma;\Delta \vdash_\ell P$. Intuitively, if $\Gamma;\Delta$ already prove that events are well ordered in P, then $\Gamma';\Delta'$ describe extra events that do not pertain to P and hence do not interfere in verifying the event ordering of P (see Appendix A.4). ■

We proceed to the result that asserts that well-ordered property of process is invariant under process reduction — Theorem 6.3.7 — but first we state its auxiliary substitution lemma.

Lemma 6.3.6 (Substitution)
Let P be such that $\Gamma; \Delta \vdash_\ell P$. If $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ then $\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} P\{x/n\}$.

Proof. By induction on the derivation of $\Gamma; \Delta \vdash_\ell P$. Essentially, condition $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ ensures the ordering prescribed for n in Γ copes with the ordering required for x (see Appendix A.4). ■

Theorem 6.3.7 (Preservation of Event Ordering)
Let process P be well-formed and $\Gamma; \Delta \vdash_\ell P$. If $P \to Q$ then there are Γ' and Δ' such that $\Gamma'; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the length of the derivation of $P \to Q$ (see Appendix A.4). ■

An immediate consequence of Theorem 6.3.7 is that processes that do not contain $\textbf{rec}_t$, and hence are ordered by $\Gamma; \emptyset$ for some Γ, preserve Γ as overall ordering throughout their evolution.

Corollary 6.3.8 (Initial Ordering)
Let process P be well-formed and $\Gamma; \emptyset \vdash_\ell P$. If $P \xrightarrow{*} Q$ then there are Γ' and Δ' such that $\Gamma'; \Delta' \vdash_\ell Q$ and $\Gamma' \cup \bigcup \Delta' \subseteq \Gamma$.

Proof. Immediate from Theorem 6.3.7. ■

Theorem 6.3.7 thus asserts that well-ordering is preserved under process reduction. Before presenting our main progress result where we exclude processes that are stuck, we first need to distinguish between finished processes and stuck processes.

Definition 6.3.9 (Finished Process)
We say P is a finished process if for any static context $\mathcal{C}$ and process Q such that $P = \mathcal{C}[Q]$ and $Q \xrightarrow{\lambda} Q'$ then either $\lambda = c \cdot l?(a)$ or $\lambda = l^d?(a)$, and $l \in \mathcal{L}_\star$.

The only immediate observations that can be made over finished processes are inputs defined on shared labels. Intuitively, finished processes can be viewed as collections of service definitions, where there are no pending service calls, neither linear protocols to be fulfilled. We then consider finished processes to be in a stable state, despite the fact they have no reductions. We may now present our main progress result that ensures well-ordered systems never run into stuck states.

Theorem 6.3.10 (Lock Freeness)
Let P be a well-formed process such that $P :: T$, where closed(T), and $\Gamma; \Delta \vdash_\ell P$, where $\ell = (\mathsf{up}, \mathsf{here})$, then either P is a finished process or there is $P \to Q$.

Proof. Follows from auxiliary results (see Appendix A.4). ■

An immediate consequence of Theorem 6.3.10 is that stuck processes are unreachable from well-ordered and well-typed processes. Differently from other approaches (e.g., [9]) our analysis focuses on systems where all communications are matched (*closed* type condition). We could also express the statement so as to consider the process and a context which type merge is *closed*.

Corollary 6.3.11 (Progress)
Let P be a well-formed process such that $P :: T$, where closed(T), and $\Gamma; \emptyset \vdash_\ell P$, where $\ell = (\mathsf{up}, \mathsf{here})$. If there is Q such that $P \xrightarrow{*} Q$ then either Q is a finished process or there is $Q \to Q'$.

$$
\begin{array}{l}
\mathit{PurchaseSystem} \\
\triangleq \\
\mathit{Buyer} \blacktriangleleft [\ (\boldsymbol{\nu} c)(\mathit{Seller} \blacktriangleleft [\ \texttt{BuyService}^{\downarrow}!(c)\ ]\ \mid c \blacktriangleleft [\ \texttt{buy}^{\downarrow}!(\mathit{prod}).\texttt{price}^{\downarrow}?(p).\texttt{details}^{\downarrow}?(d)\ ]\ ] \\
\mid \\
\mathit{Seller} \blacktriangleleft [\ \mathit{PriceDB} \mid \\
\qquad \texttt{BuyService}^{\downarrow}?(x).x \blacktriangleleft [\ \texttt{buy}^{\downarrow}?(\mathit{prod}).\texttt{askPrice}^{\uparrow}!(\mathit{prod}). \\
\qquad\qquad \texttt{priceVal}^{\uparrow}?(p).\texttt{price}^{\downarrow}!(p). \\
\qquad\qquad \textbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ] \\
\qquad\qquad\qquad \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\ ] \\
\mid \\
\mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}?(z).z \blacktriangleleft [\ \texttt{product}^{\downarrow}?(p).\texttt{details}^{\downarrow}!(\mathit{data})\ ]\ ]
\end{array}
$$

Figure 6.2: Code of the Purchase Service Collaboration Example.

Proof. Immediate from Theorem 6.3.7, Theorem 5.3.5 and Theorem 6.3.10. ∎

Corollary 6.3.11 thus states that the progress property is invariant throughout process evolution. This property entails services are always available upon request and protocols involving interleaving conversations never get stuck.

In the following section we illustrate an example derivation in our proof system, that will then allow us to assert the progress property for the purchase scenario implementation.

6.3.2 Proving Progress in the Purchase Example

In this section we prove our purchase scenario implementation is well ordered. The (low-level) code for the purchase service collaboration is given in Figure 6.2. Let us consider Γ such that:

$$
\begin{array}{l}
\mathit{Seller}.\texttt{BuyService}.(x_1)\Gamma_1 \prec_{\Gamma} \mathit{Seller}.\texttt{askPrice} \prec_{\Gamma} \\
\qquad \mathit{Seller}.\texttt{priceVal} \prec_{\Gamma} \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2
\end{array}
$$

where Γ_1 describes the ordering of the conversation which is passed in the `BuyService` service instantiation and Γ_2 describes the ordering of the conversation which is passed in the `DeliveryService` service instantiation (we omit the prescribed event orderings associated to events when they are empty, e.g., in events *Seller*.`askPrice` and *Seller*.`priceVal`).

We then have that Γ_2 is such that:

$$x_2.\texttt{product} \prec_{\Gamma_2} x_2.\texttt{details}$$

which captures the ordering of the events in which *Shipper* participates, and Γ_1 is such that:

$$
\begin{array}{l}
x_1.\texttt{buy} \prec_{\Gamma_1} \mathit{Seller}.\texttt{askPrice} \prec_{\Gamma_1} \mathit{Seller}.\texttt{priceVal} \prec_{\Gamma_1} x_1.\texttt{price} \prec_{\Gamma_1} \\
\qquad \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec_{\Gamma_1} x_1.\texttt{product} \prec_{\Gamma_1} x_1.\texttt{details}
\end{array}
$$

which captures the ordering of the events associated to the *Seller*-*Shipper* subsystem (recall that *Seller* dynamically invites *Shipper* to join the conversation).

We thus have $\Gamma; \emptyset \vdash_{(\mathsf{up},\mathsf{here})} \mathit{PurchaseSystem}$, which is derived from (cf., rule (Par)):

$$\begin{array}{l}\Gamma; \emptyset \vdash_{(\mathsf{up},\mathsf{here})} \mathit{Buyer} \blacktriangleleft [\ (\cdots)\ ] \\ \Gamma; \emptyset \vdash_{(\mathsf{up},\mathsf{here})} \mathit{Seller} \blacktriangleleft [\ (\cdots)\ ] \\ \Gamma; \emptyset \vdash_{(\mathsf{up},\mathsf{here})} \mathit{Shipper} \blacktriangleleft [\ (\cdots)\ ]\end{array}$$

When analyzing within the scope of the restricted name c, Γ is extended to Γ_3 such that:

$$\mathit{Seller}.\mathtt{BuyService}.(x_1)\Gamma_1 \prec_{\Gamma_3} c.\mathtt{buy} \prec_{\Gamma_3} \mathit{Seller}.\mathtt{askPrice} \prec_{\Gamma_3} \mathit{Seller}.\mathtt{priceVal} \prec_{\Gamma_3} \\ c.\mathtt{price} \prec_{\Gamma_3} \mathit{Shipper}.\mathtt{DeliveryService}.(x_2)\Gamma_2 \prec_{\Gamma_3} c.\mathtt{product} \prec_{\Gamma_3} c.\mathtt{details}$$

Thus, $\Gamma = \Gamma_3 \setminus c$ (cf., the rule for name restriction). Notice such ordering corresponds to the vertical timeline of the message sequence chart shown in Figure 1.2. We can then verify that:

$$\Gamma_3; \emptyset \vdash_{(\mathsf{here},\mathit{Buyer})} \mathit{Seller} \blacktriangleleft [\ \mathtt{BuyService}^{\downarrow}!(c)]$$

which is derived, omitting the step for rule (Piece), from rule (Output):

$$\Gamma_4; \emptyset \vdash_{(\mathit{Buyer},\mathit{Seller})} \mathtt{BuyService}^{\downarrow}!(c)$$

where $\Gamma_4 = \mathit{Seller}.\mathtt{BuyService}.(x_1)\Gamma_1 \bot \Gamma_3$, hence Γ_4 is such that:

$$c.\mathtt{buy} \prec_{\Gamma_4} \mathit{Seller}.\mathtt{askPrice} \prec_{\Gamma_4} \mathit{Seller}.\mathtt{priceVal} \prec_{\Gamma_4} c.\mathtt{price} \prec_{\Gamma_4} \\ \mathit{Shipper}.\mathtt{DeliveryService}.(x_2)\Gamma_2 \prec_{\Gamma_4} c.\mathtt{product} \prec_{\Gamma_4} c.\mathtt{details}$$

and where we have that the sent name c complies to the prescribed ordering for `BuyService`, hence $\Gamma_1\{x_1/c\} \subseteq \Gamma_4$. We can thus be sure that the prescribed ordering for the name passed in event $\mathit{Seller}.\mathtt{BuyService}.(x_1)\Gamma_1$ is respected by the ordering specified for the name actually being sent (c), relatively to events of greater rank than $\mathit{Seller}.\mathtt{BuyService}.(x_1)\Gamma_1$. Deriving:

$$\Gamma_3; \emptyset \vdash_{(\mathsf{here},\mathit{Buyer})} c \blacktriangleleft [\ \mathtt{buy}^{\downarrow}!(\mathit{prod}).\mathtt{price}^{\downarrow}?(p).\mathtt{details}^{\downarrow}?(d)\ ]$$

follows expected lines, where each prefix causes the relation to be trimmed down accordingly.

Regarding the seller code (abstracting away from process $\mathit{PriceDB}$), we have that:

$$\begin{array}{rl}\Gamma; \emptyset \vdash_{(\mathsf{here},\mathit{Seller})} \mathtt{BuyService}^{\downarrow}?(x).x \blacktriangleleft & [\ \mathtt{buy}^{\downarrow}?(\mathit{prod}).\mathtt{askPrice}^{\uparrow}!(\mathit{prod}). \\ & \mathtt{priceVal}^{\uparrow}?(p).\mathtt{price}^{\downarrow}!(p). \\ & \mathbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \mathtt{DeliveryService}^{\downarrow}!(y)\ ] \\ & \qquad\qquad \mid \mathtt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\end{array}$$

which is derived from (considering the rule for the input prefix):

$$\begin{array}{rl}\Gamma_5; \emptyset \vdash_{(\mathsf{here},\mathit{Seller})} x \blacktriangleleft & [\ \mathtt{buy}^{\downarrow}?(\mathit{prod}).\mathtt{askPrice}^{\uparrow}!(\mathit{prod}). \\ & \mathtt{priceVal}^{\uparrow}?(p).\mathtt{price}^{\downarrow}!(p). \\ & \mathbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \mathtt{DeliveryService}^{\downarrow}!(y)\ ] \\ & \qquad\qquad \mid \mathtt{product}^{\downarrow}!(\mathit{prod})\ )\ ]\end{array}$$

where $\Gamma_5 = (\mathit{Seller}.\texttt{BuyService}.(x_1)\Gamma_1 \bot \Gamma) \cup \Gamma_1\{x_1/x\}$, hence:

$$\begin{aligned}
\Gamma_5 \;=\;& \mathit{Seller}.\texttt{askPrice} \prec_\Gamma \mathit{Seller}.\texttt{priceVal} \prec_\Gamma \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \\
& \cup\; x.\texttt{buy} \prec_{\Gamma_1} \mathit{Seller}.\texttt{askPrice} \prec_{\Gamma_1} \mathit{Seller}.\texttt{priceVal} \prec_{\Gamma_1} x.\texttt{price} \prec_{\Gamma_1} \\
& \qquad \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec_{\Gamma_1} x.\texttt{product} \prec_{\Gamma_1} x.\texttt{details} \\
=\;& x.\texttt{buy} \prec_{\Gamma_5} \mathit{Seller}.\texttt{askPrice} \prec_{\Gamma_5} \mathit{Seller}.\texttt{priceVal} \prec_{\Gamma_5} x.\texttt{price} \prec_{\Gamma_5} \\
& \qquad \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec_{\Gamma_5} x.\texttt{product} \prec_{\Gamma_5} x.\texttt{details}
\end{aligned}$$

The derivation for messages buy, askPrice, priceVal and price follows expected lines, leading to the call to the shipper service, for which we then have that:

$$\begin{aligned}
&\mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec x.\texttt{product} \prec x.\texttt{details}; \emptyset \vdash_{(\mathit{Seller},x)} \\
&\qquad \textbf{this}(y).(\ \mathit{Shipper} \blacktriangleleft [\ \texttt{DeliveryService}^{\downarrow}!(y)\ ]\ \mid \texttt{product}^{\downarrow}!(\mathit{prod})\ )
\end{aligned}$$

which then leads to the following extension of the ordering (the ordering of y is obtained from the ordering specified for the current conversation x — cf., rule (This)):

$$\begin{aligned}
\Gamma_6 \;=\;& \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec x.\texttt{product} \prec x.\texttt{details} \\
& \cup \\
& \mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \prec y.\texttt{product} \prec y.\texttt{details}
\end{aligned}$$

which will then allow us to verify that:

$$\Gamma_2\{x_2/y\} \subseteq (\mathit{Shipper}.\texttt{DeliveryService}.(x_2)\Gamma_2 \bot \Gamma_6)$$

thus guaranteeing that the name that will eventually instantiate y copes with the ordering prescribed for the name received by the shipper upon instantiation of the DeliveryService. The rest of the derivation follows similar lines. According to the derivation shown in Section 3.3.2, we have that our purchase system is characterized by the following type:

$$\begin{aligned}
T \;\triangleq\;& \mathit{Seller} : [\ \tau\,\texttt{BuyService}(?\,\texttt{buy}(\mathit{Tp}).!\,\texttt{price}(\mathit{Tm}).\tau\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td})) \\
& \qquad \mid \tau\,\texttt{askPrice}(\mathit{Tp}).\tau\,\texttt{priceVal}(\mathit{Tm})\ ] \\
& \mid \mathit{Shipper} : [\ \tau\,\texttt{DeliveryService}(?\,\texttt{product}(\mathit{Tp}).!\,\texttt{details}(\mathit{Td}))\ ]
\end{aligned}$$

We thus have that $\Gamma; \emptyset \vdash_{(\mathsf{up},\mathsf{here})} \mathit{PurchaseSystem}$ and that $\mathit{PurchaseSystem} :: T$, where T is a closed type ($\mathit{closed}(T)$), which guarantees, in the light of Corollary 6.3.11, that the process *PurchaseSystem* will never reach a stuck configuration.

Notice our progress results apply to systems where conversations can be interleaved, including delegated conversations. For instance, consider the *Seller* code which interleaves the received service conversation with the *Seller* conversation (to access the price database) and with conversation *Shipper* (to ask *Shipper* to join in the ongoing conversation). This sort of configurations is out of reach for reference works for progress in session type literature, namely [9, 40].

6.4 Proving Progress in the π_{lab}-Calculus

To demonstrate our progress technique is not specific to systems specified Conversation Calculus, we reproduce the approach considering systems modeled in the π_{lab}-Calculus. A progress

$$\frac{\Gamma;\Delta\vdash P\quad \Gamma;\Delta\vdash Q}{\Gamma;\Delta\vdash P\mid Q}(Par)\qquad \frac{wf(\Gamma,\Delta)}{\Gamma;\Delta\vdash \mathbf{0}}(Stop)\qquad \frac{\Gamma;\Delta\vdash P}{\Gamma\setminus a;\Delta\setminus a\vdash(\boldsymbol{\nu}a)P}(Res)$$

$$\frac{\mathcal{X}\notin dom(\Delta)\quad \Gamma;(\Delta,\mathcal{X}\to\Gamma)\vdash_\ell P}{\Gamma;\Delta\vdash \textsf{rec}\,\mathcal{X}.P}(Rec)\qquad \frac{\mathcal{X}\in dom(\Delta)\quad wf(\Gamma,\Delta)}{\Gamma;\Delta\vdash\mathcal{X}}(RecVar)$$

$$\frac{\Delta(\mathcal{X});\Delta\vdash_\ell P\quad fv(\Delta(\mathcal{X}))=\emptyset\quad wf(\Gamma,\Delta)}{\Gamma;\Delta\vdash \textsf{rec}_t\,\mathcal{X}.P}(RecTerm)$$

$$\frac{(n.l.(y)\Gamma'\perp\Gamma)\cup\Gamma'\{y/x\};\Delta\vdash P\quad \Gamma'\setminus y\subseteq(n.l.(y)\Gamma'\perp\Gamma)\quad wf(\Gamma,\Delta)}{\Gamma;\Delta\vdash n\cdot l?(x).P}(Input)$$

$$\frac{(n.l.(x)\Gamma'\perp\Gamma);\Delta\vdash P\quad \Gamma'\{x/m\}\subseteq(n.l.(x)\Gamma'\perp\Gamma)\quad wf(\Gamma,\Delta)}{\Gamma;\Delta\vdash n\cdot l!(m).P}(Output)$$

Figure 6.3: Progress Proof Rules (π_{lab}-Calculus).

judgment for a π_{lab}-Calculus process P is of the form $\Gamma;\Delta\vdash P$, stating that P is well-ordered by $\Gamma;\Delta$. Notice the only difference is that now we no longer need the ℓ to keep track of the surrounding conversations of a process, since π_{lab}-Calculus action prefixes explicitly refer the location in which the communication is to take place.

We define the notion of well-ordered π_{lab}-Calculus process.

Definition 6.4.1 (Well-Ordered π_{lab}-Calculus Process)
We say π_{lab}-Calculus process P is well-ordered by Γ and Δ, noted $\Gamma;\Delta\vdash P$, if $\Gamma;\Delta\vdash P$ can be derived using the rules given in Figure 6.3.

We may then reproduce the progress results stated for CC systems. We show the main results.

Definition 6.4.2 (Well-Formed π_{lab}-Calculus Process)
We say π_{lab}-Calculus process P is well-formed if for any occurrence of a recursive process $\textsf{rec}\,\mathcal{X}.Q$ in P then no recursive process occurs in Q, the occurrences of $\mathcal{X}$ in Q are guarded by a prefix, and such variables $\mathcal{X}$ do not occur in parallel processes.

Theorem 6.4.3 (π_{lab}-Calculus Preservation of Event Ordering)
Let process P be well-formed and $\Gamma;\Delta\vdash P$. If $P\to Q$ then there is Γ' and Δ' such that $\Gamma';\Delta,\Delta'\vdash Q$ and $\Gamma'\cup\bigcup(\Delta,\Delta')\subseteq\Gamma\cup\bigcup\Delta$.

Proof. By induction on the derivation of $P\to Q$ (analogously to the proof of Theorem 6.3.7). ■

As for CC preservation of ordering, an immediate consequence of Theorem 6.4.3 is that processes ordered by some Γ and an empty recursion environment, preserve Γ as overall ordering throughout their evolution.

Corollary 6.4.4 (π_{lab}-Calculus Initial Ordering)
Let process P be well-formed and $\Gamma;\emptyset\vdash P$. If $P\xrightarrow{*}Q$ then there is Γ' and Δ' such that $\Gamma';\Delta'\vdash Q$ and $\Gamma'\cup\bigcup\Delta'\subseteq\Gamma$.

Proof. Immediate from Theorem 6.4.3. ■

Theorem 6.4.3 ensures the well-ordered condition is invariant under process reduction. Before presenting our main progress result we must first introduce π_{lab}-Calculus finished processes.

Definition 6.4.5 (π_{lab}-Calculus Finished Process)
We say P is a finished process if for any static context $\mathcal{C}$ and process Q such that $P = \mathcal{C}[Q]$ and $Q \xrightarrow{\lambda} Q'$ then $\lambda = c \cdot l?(a)$ and $l \in \mathcal{L}_\star$.

As for CC processes the only immediate observations that can be made over π_{lab}-Calculus finished processes are inputs defined on shared labels. We may now present the main progress result that ensures well-ordered and well-typed (with a *closed* type so as to ensure all communications are matched) π_{lab}-Calculus systems are not stuck.

Theorem 6.4.6 (π_{lab}-Calculus Lock Freeness)
If P is a well-formed π_{lab}-Calculus process such that $P :: L$, where $closed(L)$, and $\Gamma; \Delta \vdash P$, then either P is a finished process or there is $P \to Q$.

Proof. Follows from auxiliary results (analogously to the proof of Theorem 6.3.10). ∎

We then have as an immediate corollary that stuck processes are unreachable from well-ordered and (closed) well-typed processes.

Corollary 6.4.7 (Progress)
Let P is a well-formed π_{lab}-Calculus process such that $P :: L$, where $closed(L)$, and $\Gamma; \emptyset \vdash P$. If $P \xrightarrow{*} Q$ then either Q is a finished process or there is $Q \to Q'$.

Proof. Immediate from Theorem 6.4.3, Theorem 3.3.5 and Theorem 6.3.10. ∎

We have thus shown that our progress technique is not specific to systems modeled in the Conversation Calculus. In fact, our approach can be directly applied to a more canonical model.

6.5 Remarks

We close the chapter with some notes on related work. There are a number of progress studies for binary sessions (e.g., [3, 15, 40]), and for multiparty sessions (e.g., [9, 57]). The techniques of Dezani-Ciancaglini et al. [40] and of Bettini et al. [9] are nearer to ours as orderings on channels are imposed to guarantee the absence of cyclic dependencies. However they disallow processes that get back to interact in a session after interacting in another, and exclude interleaving on received sessions, while we allow processes that re-interact in a conversation and interleave received conversations. On the one hand, the fact that we allow processes to reenter a conversation is a consequence of our particular underlying model, that allows us to make use not only of the name of the conversation, but also of the message label to support the ordering of the same conversation interleaved with others, so we may have:

$$c.\mathtt{price} \prec \mathit{Shipper}.\mathtt{DeliveryService}.(x_2)\Gamma_2 \prec c.\mathtt{product}$$

which is not possible if one is to consider only the conversation name to support the ordering:

$$c \prec \mathit{Shipper}.(x_2)\Gamma_2 \prec c$$

since we immediately obtain a non well-founded ordering. On the other hand, we believe our technique based on prescribed event orderings is general enough to be used in other settings, so as to support the analysis of systems where processes interleave received conversations.

Regarding other works that address progress properties of systems specified on the π-Calculus we distinguish the works of Kobayashi and coauthors, namely [62, 63, 65]. Since our progress proof system relies on the conversation types analysis to capture the communication structure within each conversation, ensuring conversation protocols are followed, our approach ends up being simpler with respect to the works of Kobayashi. Namely, we do need to annotate processes with obligation/capability levels (cf., [62]), neither do we require a "simulation" over the type structure ensuring such levels of obligation/capabilities are respected throughout type evolution (cf., [63]). We do acknowledge that the approach described in [63] already provides with inference techniques, while we do not address this problem yet.

Chapter 7

Exception Handling

In this chapter we present an extension of the CC with primitives that support exception handling, originally introduced in [95]. We start by motivating the introduction of exception handling primitives through some typical examples of error recovery protocols. Then we present the language extension, define its operational semantics and briefly present some results regarding the behavioral semantics. We then focus on a possible application of the Exception Handling Conversation Calculus, which is to model compensations and thus give support for long running transactions. We close the chapter with some remarks on related work.

7.1 Recovering from Error

In any realistic model for service-oriented computation we expect to find some form of language support to handle exceptional behavior. Examples to motivate this claim include, for example, the need to express services that may be interrupted on request at any point in their execution, or services that may be interrupted after some timeout occurs — a party waiting on some communication may be willing to wait only some given amount of time, giving up after that. Of particular importance is then the coordination of the several parties involved in the service conversation, since an exception local to a party must be somehow propagated to all other parties involved in the service task. Other session-based approaches to service-oriented computing model such error propagation internally to the semantics of the exception handling primitives, namely in the SCC model (Boreale et al. [12]), in the CaSPiS model (Boreale et al. [13]), and in the approach of Carbone et al. [31]. We take a different approach, by providing our exception handling primitives with a standard "local" semantics, leaving the task of coordinating the exceptions in charge of the basic communication primitives.

We introduce two primitives to support exception handling: a primitive to signal exceptions **throw**.P, and a construct **try** P **catch** Q that allows process P to actively run, up to the point an exception is thrown, in which case P is terminated and the exception handler code Q is activated. We illustrate a few usage idioms for our exception handling primitives, starting by a programming idiom that captures remote exception throwing.

$$\begin{array}{ll} \mathit{Server} \blacktriangleleft [& \\ \quad \mathbf{def}\ \mathtt{Interruptible} \Rightarrow & \mathbf{new}\ \mathit{Server} \cdot \mathtt{Interruptible} \Leftarrow \\ \quad\quad \mathtt{stopRequest}^{\downarrow}?().\mathtt{urgentStop}^{\downarrow}!().\mathbf{throw}.\mathbf{0} & \quad \mathtt{urgentStop}^{\downarrow}?().\mathbf{throw}.\mathbf{0} \\ \quad\quad \mid \mathit{ServiceProtocol}\ ] & \quad \mid \mathit{ClientProtocol} \end{array}$$

In this example, any instance of the `Interruptible` service may be interrupted by process *ServiceProtocol* by dropping a message `stopRequest` in the service conversation. Such message causes the `Interruptible` service code to send an `urgentStop` message to the service client side, after sending which the service code proceeds by throwing an exception which causes abortion of the server side of the conversation. On the other hand, the client code throws an exception at the client side upon reception of `urgentStop`. Notice that this behavior will happen concurrently with ongoing interactions between *ServiceProtocol* and *ClientProtocol*. In this case, the exception issued at the server and client sites has to be managed by appropriate handlers (by enclosing try-catch blocks). In the next example, no exception will be propagated to the server site, but only to the client site, and as the result of any exception thrown by the *ServiceProtocol* process.

```
Server ◀ [
  def Interruptible ⇒
    try
      ServiceProtocol
    catch urgentStop↓!()
```

```
new Server · Interruptible ⇐
  urgentStop↓?().throw.0
  | ClientProtocol
```

In the above examples, the decision to terminate the ongoing remote interactions is triggered by the service code. In the next example, we show a simple variation of the idioms above, where the decision to terminate the ongoing service instance is responsibility of the service context. In the following, any instance of the `Interruptible` service may be terminated by the service provider by means of dropping a `killRequest` message in conversation *Server*.

```
Server ◀ [
  def Interruptible ⇒
    try
      killRequest↑?().throw.0
      | ServiceProtocol
    catch urgentStop↓!() ]
```

```
new Server · Interruptible ⇐
  urgentStop↓?().throw.0
  | ClientProtocol
```

If we were to extend the language with a **wait** primitive with the expected semantics, then we would also be able to specify services that have a limited time span to run, illustrated next.

```
Server ◀ [
  def TimeBounded ⇒
    try
      timeAllowed↓?(delay).wait(delay).throw.0 |
      ServiceProtocol
    catch urgentStop↓!() ]
```

```
new Server · TimeBounded ⇐
  urgentStop↓?().throw.0
  | ClientProtocol
```

Thus, any instance of the `TimeBounded` service will be allocated no more than *delay* time units before being interrupted, where *delay* is a dynamic parameter value read from the server side context. As a final example of our exception handling idioms, consider service `Repeatable` is instantiated on site *Server*, and repeatedly relaunched on each failure — failure will be signaled

by an exception thrown within the local protocol *ClientProtocol*, possibly as a result of a remote `urgentStop` message.

$$
\begin{array}{ll}
\textit{Server} \blacktriangleleft [& \textbf{rec}\ \textit{Restart}. \\
\quad \textbf{def}\ \texttt{Repeatable} \Rightarrow & \quad \textbf{try} \\
\quad\quad \textbf{try} & \quad\quad \textbf{new}\ \textit{Server} \cdot \texttt{Repeatable} \Leftarrow \\
\quad\quad\quad \texttt{killRequest}^{\uparrow}?().\textbf{throw}.\mathbf{0} & \quad\quad\quad \texttt{urgentStop}^{\downarrow}?().\textbf{throw}.\mathbf{0} \\
\quad\quad\quad \mid \textit{ServiceProtocol} & \quad\quad\quad \mid \textit{ClientProtocol} \\
\quad\quad \textbf{catch}\ \texttt{urgentStop}^{\downarrow}!()\ ] & \quad \textbf{catch}\ \textit{Restart}
\end{array}
$$

We may thus combine the core CC primitives with the exception handling primitives so as to model several interesting error recovery scenarios. Furthermore, exception handling primitives also give support to the definition of more complex control flow structures. Of particular interest to the setting of service-oriented computing is to support some form of compensation so as to provide support for long running transactions. After formally presenting the CC_{exc} we show how a model of transactions with compensations can be encoded in our language.

7.2 The Exception Handling Conversation Calculus (CC_{exc})

In this section we define the syntax of the extension of the Conversation Calculus with exception handling primitives (CC_{exc}). The CC_{exc} extends the core CC, presented in Chapter 4, with a try-catch block and the **throw** primitive to signal exceptions.

Definition 7.2.1 (CC_{exc} Syntax)
The syntax of the Exception Handling Conversation Calculus is given in Figure 7.1.

The sets of free and bound names, free labels, free variables and free recursion variables of CC_{exc} processes are defined as expected, extending core CC definitions with the cases of the try-catch and the **throw** constructs. We informally describe these constructs next.

Exception Handling Primitives

To model exceptional behavior, in particular fault signaling, fault detection, and resource disposal, we borrow the classical **try** – **catch**– and **throw**– from imperative languages, and adapt them to the concurrent setting. The primitive to signal exceptional behavior is denoted by:

$$\textbf{throw}.\textit{Exception}$$

This construct throws an exception with continuation the process *Exception*, and has the effect of forcing the termination of all other processes running in all enclosing contexts, up to the point where a **try** – **catch** block is found (if any). The continuation *Exception* will be activated when (and if) the exception is caught by such an exception handler. The exception handler construct:

$$\textbf{try}\ P\ \textbf{catch}\ \textit{Handler}$$

actively allows process P to run until some exception is thrown inside P. At that moment, all of P is terminated, and the *Handler* process, which is guarded by **try** – **catch**, is activated,

$$
\begin{array}{lcll}
a, b, c, \ldots & \in & \Lambda & \text{(Names)} \\
x, y, z, \ldots & \in & \mathcal{V} & \text{(Variables)} \\
n, m, o, u, v \ldots & \in & \Lambda \cup \mathcal{V} & \text{(Identifiers)} \\
l, s \ldots & \in & \mathcal{L} & \text{(Labels)} \\
\mathcal{X}, \mathcal{Y}, \ldots & \in & \chi & \text{(Recursion Variables)}
\end{array}
$$

$$
\begin{array}{lcll}
P, Q, R & ::= & \mathbf{0} & \text{(Inaction)} \\
 & | & P \mid Q & \text{(Parallel Composition)} \\
 & | & (\boldsymbol{\nu} a)P & \text{(Name Restriction)} \\
 & | & \textbf{rec}\, \mathcal{X}.P & \text{(Recursion)} \\
 & | & \mathcal{X} & \text{(Variable)} \\
 & | & n \blacktriangleleft [P] & \text{(Conversation Access)} \\
 & | & \Sigma_{i \in I}\ \alpha_i.P_i & \text{(Prefix Guarded Choice)} \\
 & | & \textbf{try}\ P\ \textbf{catch}\ Q & \text{(Try-Catch Block)} \\
 & | & \textbf{throw}.P & \text{(Throw)}
\end{array}
$$

$$
\begin{array}{lcll}
d & ::= & \downarrow \mid \uparrow & \text{(Directions)} \\
\alpha & ::= & l^d!(n_1, \ldots, n_k) & \text{(Output)} \\
 & | & l^d?(x_1, \ldots, x_k) & \text{(Input)} \\
 & | & \textbf{this}(x) & \text{(Conversation Awareness)}
\end{array}
$$

Figure 7.1: The Exception Handling Conversation Calculus ($\mathrm{CC}_{\mathrm{exc}}$) Syntax.

concurrently with the continuation *Exception* of the **throw**.*Exception* that originated the exception, in the context of the given **try** − **catch**− block. By exploiting the interaction potential of the *Handler* and *Exception* processes, one may represent many adequate recovery and resource disposal protocols.

Given this informal understanding of the exception handling primitives we may now formally define the operational semantics of the $\mathrm{CC}_{\mathrm{exc}}$.

7.3 Operational Semantics

We define the operational semantics of the $\mathrm{CC}_{\mathrm{exc}}$ by means of a labeled transition system. The $\mathrm{CC}_{\mathrm{exc}}$ labeled transition system conservatively extends that of the core CC, in the sense it incorporates all the rules defined for the CC. We nevertheless present the full definitions, repeating what was introduced for core CC, but focus our presentation on the new aspects.

As for the core CC, by $P \xrightarrow{\lambda} Q$ we denote that process P may evolve to process Q by performing the action represented by the transition label λ. We define $\mathrm{CC}_{\mathrm{exc}}$ transition labels.

Definition 7.3.1 ($\mathrm{CC}_{\mathrm{exc}}$ Transition Labels)
Extended directions, actions and transition labels are defined as follows:

$$
\begin{array}{lcll}
d_e & ::= & d \mid \downarrow\uparrow & \text{(Extended Directions)} \\
\sigma & ::= & \tau \mid l^d!(\tilde{a}) \mid l^d?(\tilde{a}) \mid \texttt{this}^{d_e} \mid \texttt{throw} & \text{(Actions)} \\
\lambda & ::= & c\ \sigma \mid \sigma \mid (\nu a)\lambda & \text{(Transition Labels)}
\end{array}
$$

We denote by $\mathcal{T}$ the set of all transition labels.

CC_{exc} transition labels extend CC transition labels with an action `throw` which represents an exception signal. We show an example to motivate the definition of the semantics of the CC_{exc}.

Example 7.3.2 *Consider process:*

$$\textbf{throw}.\texttt{notify}^{\downarrow}!(\mathit{errorInfo})$$

which is willing to signal an exception and afterwards notifies another process that a particular error has occurred. The process exhibits the following `throw` *transition:*

$$\textbf{throw}.\texttt{notify}^{\downarrow}!(\mathit{errorInfo}) \xrightarrow{\texttt{throw}} \texttt{notify}^{\downarrow}!(\mathit{errorInfo})$$

Such a `throw` *observation causes the termination of all processes that are running in parallel to the process that signaled the exception, so, e.g., if in parallel lies process* $\texttt{chat}^{\downarrow}!(\mathit{text})$ *we have:*

$$\textbf{throw}.\texttt{notify}^{\downarrow}!(\mathit{errorInfo}) \mid \texttt{chat}^{\downarrow}!(\mathit{text}) \xrightarrow{\texttt{throw}} \texttt{notify}^{\downarrow}!(\mathit{errorInfo})$$

If the process is placed in a try-catch block, the `throw` *observation is caught by the try-catch, resulting in the activation of the handler in parallel with the continuation of the exception:*

$$\begin{array}{l} \textbf{try} \\ \quad \textbf{throw}.\texttt{notify}^{\downarrow}!(\mathit{errorInfo}) \mid \texttt{chat}^{\downarrow}!(\mathit{text}) \\ \textbf{catch}\ \texttt{notify}^{\downarrow}?(x).P \\ \xrightarrow{\tau} \\ \texttt{notify}^{\downarrow}!(\mathit{errorInfo}) \mid \texttt{notify}^{\downarrow}?(x).P \end{array}$$

Notice that the continuation of the process that originally signals the exception gets to interact with the try-catch handler, allowing for fault handler to exploit specific error information.

Given this basic intuition we may now present the CC_{exc} labeled transition system. We describe the rules specific to the exception handling behavior, starting by the rule that captures the signaling of an exception:

$$\textbf{throw}.P \xrightarrow{\texttt{throw}} P$$

The rule specifies an exception signal may be observed over a **throw** prefix, resulting in the activation of the continuation process. The exception signal causes the termination of parallel contexts and conversation pieces, as expressed by the following rules:

$$\frac{P \xrightarrow{\texttt{throw}} R}{P \mid Q \xrightarrow{\texttt{throw}} R} \qquad \frac{P \xrightarrow{\texttt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\texttt{throw}} R}$$

On the other hand, the exception signal transparently crosses a name restriction boundary:

$$\frac{P \xrightarrow{\texttt{throw}} R}{(\boldsymbol{\nu} a)P \xrightarrow{\texttt{throw}} (\boldsymbol{\nu} a)R}$$

$$l^d!(\tilde{a}).P \xrightarrow{l^d!(\tilde{a})} P \; (Out) \qquad l^d?(\tilde{x}).P \xrightarrow{l^d?(\tilde{a})} P\{\tilde{x}/\tilde{a}\} \; (In) \qquad \frac{\alpha_j.P_j \xrightarrow{\lambda} Q \quad j \in I}{\Sigma_{i \in I}\ \alpha_i.P_i \xrightarrow{\lambda} Q} (Sum)$$

$$\frac{P \xrightarrow{\lambda} Q \quad a \in out(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{(\nu a)\lambda} Q} (Open) \qquad \frac{P \xrightarrow{\lambda} Q \quad a \notin na(\lambda)}{(\boldsymbol{\nu} a)P \xrightarrow{\lambda} (\boldsymbol{\nu} a)Q} (Res)$$

$$\frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R) \quad \lambda \neq \mathtt{throw}}{P \mid R \xrightarrow{\lambda} Q \mid R} (Par\text{-}l) \qquad \frac{P \xrightarrow{\lambda} Q \quad bn(\lambda) \# fn(R) \quad \lambda \neq \mathtt{throw}}{R \mid P \xrightarrow{\lambda} R \mid P} (Par\text{-}r)$$

$$\frac{P \xrightarrow{(\nu\tilde{a})\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \tilde{a} \# fn(Q) \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')} (Close\text{-}l)$$

$$\frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{(\nu\tilde{a})\lambda_2} Q' \quad \tilde{a} \# fn(P) \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')} (Close\text{-}r)$$

$$\frac{P \xrightarrow{\lambda_1} P' \quad Q \xrightarrow{\lambda_2} Q' \quad \lambda_1 \bullet \lambda_2 \neq \circ}{P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} P' \mid Q'} (Comm) \qquad \frac{P\{\mathcal{X}/\mathbf{rec}\,\mathcal{X}.P\} \xrightarrow{\lambda} Q}{\mathbf{rec}\,\mathcal{X}.P \xrightarrow{\lambda} Q} (Rec)$$

Figure 7.2: Transition Rules for Basic Operators (π-Calculus).

The following rule captures the exception signal handling:

$$\frac{P \xrightarrow{\mathtt{throw}} R}{\mathbf{try}\ P\ \mathbf{catch}\ Q \xrightarrow{\tau} R \mid Q}$$

The rule specifies that if the process running in the **try** part signals an exception, then the try-catch block evolves, through a τ transition, to a state where the exception handler Q is activated in parallel with the continuation of the exception R. On the other hand, to capture the case when the **try** process exhibits behavior different from `throw` we have the rule:

$$\frac{P \xrightarrow{\lambda} Q \quad \lambda \neq \mathtt{throw}}{\mathbf{try}\ P\ \mathbf{catch}\ R \xrightarrow{\lambda} \mathbf{try}\ Q\ \mathbf{catch}\ R}$$

which states that the try-catch block behaves like the process running in the **try**, except if it signals an exception.

We may now present the formal definition of our transition relations. We assume a direct extension of the synchronization algebra of the CC (Definition 4.3.8), adding the cases for the `throw` label which does not synchronize with any label, (i.e., $\mathtt{throw} \bullet \lambda = \circ$ and $\lambda \bullet \mathtt{throw} = \circ$). For clarity, we split the presentation into three sets of rules: the first in Figure 7.2 contains the rules for the basic operators; the second in Figure 7.3 groups the rules specific to the core Conversation Calculus; the third groups the rules specific to exception handling behavior.

Definition 7.3.3 (Transition Relations)
The transition relations $\{\xrightarrow{\lambda} \mid \lambda \in \mathcal{T}\}$ *are defined by the rules in Figures 7.2, 7.3 and 7.4.*

$$\frac{P \xrightarrow{\lambda^{\uparrow}} Q \quad (c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda^{\downarrow}} c \blacktriangleleft [Q]} (\mathit{Here}) \qquad \frac{P \xrightarrow{\lambda} Q \quad (\mathit{unloc}(\lambda))}{c \blacktriangleleft [P] \xrightarrow{c \cdot \lambda} c \blacktriangleleft [Q]} (\mathit{Loc})$$

$$\frac{P \xrightarrow{\lambda} Q \quad (\mathit{loc}(\lambda), c \notin bn(\lambda))}{c \blacktriangleleft [P] \xrightarrow{\lambda} c \blacktriangleleft [Q]} (\mathit{Through}) \qquad \frac{P \xrightarrow{\tau} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (\mathit{Tau})$$

$$\frac{P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q}{c \blacktriangleleft [P] \overset{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} c \blacktriangleleft [Q]} (\mathit{ThisHere}) \qquad \frac{P \overset{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} Q}{c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q]} (\mathit{ThisLoc})$$

$$\textbf{this}(x).P \overset{c}{} \xrightarrow{\mathtt{this}^{\downarrow}} P\{x/c\} \; (\mathit{This})$$

Figure 7.3: Transition Rules for Conversation Operators.

$$\textbf{throw}.P \xrightarrow{\mathtt{throw}} P \;\; (\mathit{Throw}) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{P \mid Q \xrightarrow{\mathtt{throw}} R} (\mathit{ThrowPar})$$

$$\frac{P \xrightarrow{\mathtt{throw}} R}{n \blacktriangleleft [P] \xrightarrow{\mathtt{throw}} R} (\mathit{ThrowConv}) \qquad \frac{P \xrightarrow{\mathtt{throw}} R}{\textbf{try } P \textbf{ catch } Q \xrightarrow{\tau} R \mid Q} (\mathit{Catch})$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{\textbf{try } P \textbf{ catch } R \xrightarrow{\lambda} \textbf{try } Q \textbf{ catch } R} (\mathit{Try})$$

Figure 7.4: Transition Rules for Exception Handling Operators.

The transition relation of the CC_{exc} is a conservative extension of the transition relation of the core CC (Definition 4.3.14) in the sense that any behavior that can be derived by the CC transition system can also be derived by the CC_{exc} transition system. Notice that action `throw` is excluded in rules (*Par-l*) and (*Par-r*) since a different behavior is expected when a `throw` is observed in one of the branches of a parallel composition. Notice also rule (*Res*) includes the case of exception signal which transparently crosses the restriction boundary.

As for the CC we define a notion of autonomous evolution for CC_{exc} processes, corresponding to τ observations.

Definition 7.3.4 (CC_{exc} Reduction)
The relation of reduction on processes, noted $P \rightarrow Q$, is defined as $P \xrightarrow{\tau} Q$. Also, we denote by $P \xrightarrow{*} Q$ the reflexive transitive closure of the reduction relation.

Using the operational semantics provided by the labeled transition system, we may now turn to characterizing the behavioral semantics of the CC_{exc}.

7.4 Behavioral Semantics

In this section we define the behavioral semantics of the CC_{exc} and report results, as established for the core CC, that corroborate our choices in the development of the mathematical interpretation for the syntactical constructs of our language. The behavioral semantics of the CC_{exc} is defined by means of a standard notion of bisimilarity.

Definition 7.4.1 ($\mathbf{CC_{exc}}$ Strong Bisimulation)
A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:

If $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \# fn(Q)$
then there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$.

Two processes are related by a bisimulation if they can simulate one another by performing the same observations and arriving at observationally-equivalent states. We prove strong bisimulations are closed under set union (see Proposition 4.4.2) and define strong bisimilarity.

Definition 7.4.2 ($\mathbf{CC_{exc}}$ Strong Bisimilarity)
Strong bisimilarity, noted $\sim$, is the union of all strong bisimulations.

By considering such observational equivalence we are able to characterize our intended notion of semantic object: if two processes are bisimilar then they represent (possibly distinct) specifications of the same abstract behavior. We also establish strong bisimilarity is an equivalence relation and that it is preserved under the usual structural equivalence laws (see Proposition 4.4.4 and Theorem 4.4.5). As for the core CC we establish strong bisimilarity is a congruence.

Theorem 7.4.3 (Congruence)
Strong bisimilarity is a congruence.

1. *If $P \sim Q$ then $l^d!(\tilde{n}).P \sim l^d!(\tilde{n}).Q$.*
2. *If $P\{\tilde{x}/\tilde{n}\} \sim Q\{\tilde{x}/\tilde{n}\}$ for all $\tilde{n}$ then $l^d?(\tilde{x}).P \sim l^d?(\tilde{x}).Q$.*
3. *If $P\{x/n\} \sim Q\{x/n\}$ for all n then $\mathbf{this}(x).P \sim \mathbf{this}(x).Q$.*
4. *If $\alpha_i.P_i \sim \alpha'_i.Q_i$, for all $i \in I$, then $\Sigma_{i\in I}\, \alpha_i.P_i \sim \Sigma_{i\in I}\, \alpha'_i.Q_i$.*
5. *If $P \sim Q$ then $n \blacktriangleleft [P] \sim n \blacktriangleleft [Q]$.*
6. *If $P \sim Q$ then $(\boldsymbol{\nu} a)P \sim (\boldsymbol{\nu} a)Q$.*
7. *If $P \sim Q$ then $P \mid R \sim Q \mid R$.*
8. *If $P\{\mathcal{X}/R\} \sim Q\{\mathcal{X}/R\}$, for all R, then $\mathbf{rec}\,\mathcal{X}.P \sim \mathbf{rec}\,\mathcal{X}.Q$.*
9. *If $P \sim Q$ then $\mathbf{throw}.P \sim \mathbf{throw}.Q$.*
10. *If $P \sim Q$ then $\mathbf{try}\ P\ \mathbf{catch}\ R \sim \mathbf{try}\ Q\ \mathbf{catch}\ R$.*
11. *If $P \sim Q$ then $\mathbf{try}\ R\ \mathbf{catch}\ P \sim \mathbf{try}\ R\ \mathbf{catch}\ Q$.*

Proof. By coinduction on the definition of strong bisimulation (see Appendix A.5). ∎

The congruence result serves as a sanity check for the mathematical interpretation we choose for our syntactical constructs, showing they are proper functions at the level of the abstract behavior: if we take two specifications of the same abstract behavior and place them in a $\mathrm{CC_{exc}}$ language context, then we again have two specifications of the same abstract behavior (which, in general, is distinct from the original one).

7.4.1 Weak Bisimilarity

In this section we introduce the weak variant of strong bisimilarity for $\mathrm{CC}_{\mathrm{exc}}$ processes, which equates processes by observing only their external actions, abstracting away from internal behavior the processes may have. Weak transitions are defined as usual, by collapsing together sequences of τ transitions, possibly along with some external action.

Definition 7.4.4 (Weak Transition)
We denote by $\xrightarrow{\tau}^*$ *the reflexive transitive closure of* $\xrightarrow{\tau}$*. Then we say* P *has a weak* λ *transition to* Q*, noted* $P \stackrel{\lambda}{\Longrightarrow} Q$*, if it can be derived from:*

$$\begin{array}{lll} & P \xrightarrow{\tau}^* P' \xrightarrow{\lambda} Q' \xrightarrow{\tau}^* Q & (\textit{if } \lambda \neq \tau) \\ \textit{or} & P \xrightarrow{\tau}^* Q & (\textit{if } \lambda = \tau) \end{array}$$

Notice that in a $P \stackrel{\lambda}{\Longrightarrow} Q$ transition, process P may not evolve at all ($P \stackrel{\tau}{\Longrightarrow} P$), or may evolve in a countless number of internal reduction steps. On the other hand when $\lambda \neq \tau$ we are sure that the process has evolved, at least through a λ transition. We define weak bisimulation.

Definition 7.4.5 (Weak Bisimulation)
A weak bisimulation is a symmetric binary relation $\mathcal{R}$ *on processes such that, for all processes* P *and* Q*, if* $P\mathcal{R}Q$*, we have:*

If $P \xrightarrow{\lambda} P'$ *and* $bn(\lambda) \# fn(Q)$
then there is Q' *such that* $Q \stackrel{\lambda}{\Longrightarrow} Q'$ *and* $P'\mathcal{R}Q'$*.*

Two processes are related by a weak bisimulation if given an observation in one of the processes then the other can match it, up to some internal actions, and the arrival states of the transitions are related. We prove some properties of weak bisimulations, namely that the union of weak bisimulations is a weak bisimulation (cf., Proposition 4.4.24), and define weak bisimilarity.

Definition 7.4.6 (Weak Bisimilarity)
Weak bisimilarity, noted $\approx$*, is the union of all weak bisimulations.*

Weak bisimilarity thus provides with another useful notion of semantic object, equating processes that correspond to the same abstract behavior, regardless of some "private" internal implementations. We establish that weak bisimilarity is an equivalence relation and that it is preserved under a standard set of structural laws (cf., Proposition 4.4.26 and Theorem 4.4.27). Analogously to strong bisimilarity we prove weak bisimilarity is a congruence.

Theorem 7.4.7 (Congruence)
Weak bisimilarity is a congruence.

1. *If* $P \approx Q$ *then* $l^d!(\tilde{n}).P \approx l^d!(\tilde{n}).Q$*.*
2. *If* $P\{\tilde{x}/\tilde{n}\} \approx Q\{\tilde{x}/\tilde{n}\}$ *for all* $\tilde{n}$ *then* $l^d?(\tilde{x}).P \approx l^d?(\tilde{x}).Q$*.*
3. *If* $P\{x/n\} \approx Q\{x/n\}$ *for all* n *then* $\textbf{this}(x).P \approx \textbf{this}(x).Q$*.*
4. *If* $\alpha_i.P_i \approx \alpha'_i.Q_i$*, for all* $i \in I$*, then* $\Sigma_{i\in I}\, \alpha_i.P_i \approx \Sigma_{i\in I}\, \alpha'_i.Q_i$*.*

5. *If* $P \approx Q$ *then* $n \blacktriangleleft [P] \approx n \blacktriangleleft [Q]$.

6. *If* $P \approx Q$ *then* $(\boldsymbol{\nu} a)P \approx (\boldsymbol{\nu} a)Q$.

7. *If* $P \approx Q$ *then* $P \mid R \approx Q \mid R$.

8. *If* $P\{\mathcal{X}/R\} \approx Q\{\mathcal{X}/R\}$*, for all* R*, then* $\textsf{rec}\, \mathcal{X}.P \approx \textsf{rec}\, \mathcal{X}.Q$.

9. *If* $P \approx Q$ *then* $\textsf{throw}.P \approx \textsf{throw}.Q$.

10. *If* $P \approx Q$ *then* $\textsf{try}\ P\ \textsf{catch}\ R \approx \textsf{try}\ Q\ \textsf{catch}\ R$.

11. *If* $P \approx Q$ *then* $\textsf{try}\ R\ \textsf{catch}\ P \approx \textsf{try}\ R\ \textsf{catch}\ Q$.

Proof. By coinduction on the definition of weak bisimulation, along the lines of the proof of Theorem 7.4.3. ■

Given such results that report on the theoretical soundness of our approach, we now turn to the more practical perspective so as to illustrate what sort of service-oriented communication patterns can be captured using the $\mathrm{CC}_{\mathrm{exc}}$.

7.5 Implementing Compensations in the $\mathrm{CC}_{\mathrm{exc}}$

In this section we show how the $\mathrm{CC}_{\mathrm{exc}}$ may be used to model interesting error recovery protocols, namely we show how compensations can be implemented in $\mathrm{CC}_{\mathrm{exc}}$. In the service-oriented computing setting, traditional ACID transactional properties are hard to enforce, since service transactions are most likely to be long running, and it is typically not possible to "block" the system resources for so long so as to guarantee such properties are uphold. The notion of compensation was developed in order to introduce some flexibility so as to support long running transactions while keeping some of the ideal transactional properties: the intuitive idea is that we may optimistically run the operations involved in the transaction, e.g., allowing such operations to access uncommitted resources, and if something goes wrong further along, then compensations are executed so as to recover the initial state of the system.

Although well known in the context of transaction processing systems for quite a long time (see e.g., [49, 67]), the use of compensation as a mechanism to undo the effect of long running transactions, and thus recover some properties of confined ACID transactions (at least consistency and durability), is frequently assumed to be the recovery mechanism of choice for aborted transactions in distributed services. In this context, some confusion sometimes arises between the notions of exception and compensation: obviously these are quite different and even independent concepts. Exceptions are a mechanism to signal abnormal conditions during program execution, while compensations are commands intended to undo the effects of previously successfully completed tasks during a transaction. Sometimes, an exception mechanism may be a useful tool to describe a compensation mechanism, but this does not need to be always the case. Clearly, compensations are most useful as a structuring device to describe undoable actions, and make particular sense when the underlying process language is based on a concept of basic action, which effect can be reversed in some way. Given this understanding, it is not difficult to express a structured compensation mechanism using our primitives.

We illustrate a possible approach by encoding the Compensating CSP calculus (cCSP) presented in [18]. We start by introducing the cCSP language, then we show how we can encode it

Atomic actions:

$$
\begin{array}{llll}
A, B & ::= & a & \text{(Primitive Action)} \\
& | & \langle R \rangle & \text{(Transaction)}
\end{array}
$$

Basic Programs:

$$
\begin{array}{llll}
P, Q & ::= & A & \text{(Action)} \\
& | & P; Q & \text{(Sequential Composition)} \\
& | & P \oplus Q & \text{(Choice)} \\
& | & P \mid Q & \text{(Parallel Composition)} \\
& | & P \rhd Q & \text{(Exception Handler)} \\
& | & \mathit{skip} & \text{(Normal Termination)} \\
& | & \mathit{throw} & \text{(Throw an Interrupt)}
\end{array}
$$

Compensable Programs:

$$
\begin{array}{llll}
R, T & ::= & A \div B & \text{(Compensation Pair)} \\
& | & R; T & \text{(Sequential Composition)} \\
& | & R \oplus T & \text{(Choice)} \\
& | & R \mid T & \text{(Parallel Composition)} \\
& | & R \rhd T & \text{(Exception Handler)} \\
& | & \mathit{skipp} & \text{(Normal Termination)} \\
& | & \mathit{throww} & \text{(Thrown an Interrupt)}
\end{array}
$$

Figure 7.5: Compensating CSP (cCSP) Syntax.

in the CC_{exc}. Although we do not present here a formal proof of correctness of the encoding, as it falls out of scope of this text, such proof has been reported elsewhere [24].

7.5.1 The cCSP Language

We briefly present the cCSP language, starting by the language syntax (cf., [18, 24]).

Definition 7.5.1 (cCSP Syntax)
The syntax of the cCSP language is given in Figure 7.5.

We informally describe the semantics of the cCSP primitives. Atomic actions are either primitive actions a or transactions $\langle R \rangle$. Primitive actions are some indivisible basic actions which either execute successfully or fail, in which case they have no effect in the system state. Transactions are composed by a structured set of actions, but have the same behavior as a primitive action: they either successfully terminate or they fail, in which case they have no visible effect in the system state — the final state is, in some sense, equivalent to the state of the system at the point right before the transaction started executing.

Basic programs (P, Q) are built from such atomic actions, by composing them in parallel $(P \mid Q)$, and by composing them in sequence $(P; Q)$ — notice that the prefix P in $P; Q$ may be a structured process which must terminate before Q can start. Also, we consider a choice construct $P \oplus Q$, which represents processes that can internally choose to proceed as P or as Q, and an exception handler construct $P \rhd Q$ which behaves like a try-catch block: $P \rhd Q$ behaves as P up to the point an exception is signaled by P, in which case all of P is terminated and Q is activated. The *throw* primitive signals an exception has occurred, while *skip* signals normal termination.

Basic programs allow us to structure atomic actions. On the other hand, compensable programs support the specification of the operations in the scope of a transaction, ensuring that for each atomic action there is an associated compensation. The basic compensable program is a pair $A \div B$ where A and B are atomic actions: the intuition is that the atomic action B is able to undo the effect of the A atomic action, leading to a state that should be in some sense equivalent to the state right before A was executed. Compensable programs may also be specified by sequential composition, choice, parallel composition, and the exception handler construct, analogously to basic programs, and also include primitives to signal normal and abnormal termination: *skipp* and *throww*, respectively.

Given this basic informal understanding of the Compensating CSP language we may now present how we encode it in the Exception Handling Conversation Calculus.

7.5.2 Encoding cCSP in $\mathrm{CC}_{\mathrm{exc}}$

In this section we present an encoding of the Compensating CSP language in the $\mathrm{CC}_{\mathrm{exc}}$. The starting point to develop such an encoding is the notion of primitive action: we consider a basic action to be any $\mathrm{CC}_{\mathrm{exc}}$ process P that, after successful completion, sends (only once) the message `ok` to its environment. A basic action is always supposed to enjoy the following property: it either executes successfully to completion, or it aborts. It the case of abortion, a basic action is required not to perform any interesting relevant actions, except signaling abortion by throwing an exception. Thus, to model primitive actions we assume a mapping from cCSP primitive actions to $\mathrm{CC}_{\mathrm{exc}}$ processes that behave as described above: they either signal completion by sending a single `ok` message, or signal failure by throwing an exception, leaving the system in an equivalent state.

Before presenting the encoding we introduce anonymous protected conversation contexts, which are useful to isolate the several parts of the code that realize the encoding, and an auxiliary notation useful to lighten presentation.

Notation 7.5.2 *We abbreviate $l^d!()$ and $l^d?()$ with $l^d!$ and $l^d!$. Also, we omit $\downarrow$ directions.*

Definition 7.5.3 (Anonymous Protected Conversations)
By $[P]$ we denote that CC_{exc} process P is running in an anonymous protected conversation, defined as:

$$[P] \triangleq (\boldsymbol{\nu} a)(a \blacktriangleleft [P]) \quad (a \notin fn(P))$$

By placing a process P in such a protected conversation, we are sure that interactions local to P (here $\downarrow$ messages) are not subject to interference from any other part in the system.

We denote by $[\![P]\!]_{\mathtt{ok}}$ the $\mathrm{CC}_{\mathrm{exc}}$ process which encodes the cCSP basic program P. The `ok` subscript specifies the label of the message that signals normal completion. Basic programs include atomic actions, therefore include primitive actions and closed transactions. We show the inductively defined encoding of basic programs in Figure 7.6. Notice that most encodings make use of anonymous protected conversations to protect the code from external interference. We thus have that a *throw* action is implemented by the **throw** primitive, and that *skip* is implemented by the process that sends message `ok` (notice $[\ \mathtt{ok}^{\uparrow}!\ ] \sim \mathtt{ok}^{\downarrow}!$). To encode the structuring operators we use the `ok` messages to define the control flow (cf., Milner's encoding of

$$
\begin{array}{lcl}
[\![a]\!]_{\mathtt{ok}} & \triangleq & P_a \\
[\![\langle T\rangle]\!]_{\mathtt{ok}} & \triangleq & [\ [\![T]\!]_{\mathtt{ok,ab,cm,cb}} \mid \mathtt{ab}?.\textbf{throw}.\mathbf{0} \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!\ ] \\
[\![P;Q]\!]_{\mathtt{ok}} & \triangleq & [\ [\![P]\!]_{\mathtt{ok}_1} \mid \mathtt{ok}_1?.[\![Q]\!]_{\mathtt{ok}} \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!\ ] \\
[\![P \oplus Q]\!]_{\mathtt{ok}} & \triangleq & [\ \mathtt{t}! + \mathtt{f}! \mid \mathtt{t}?.[\![P]\!]_{\mathtt{ok}} \mid \mathtt{f}?.[\![Q]\!]_{\mathtt{ok}} \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!\ ] \\
[\![P \mid Q]\!]_{\mathtt{ok}} & \triangleq & [\ [\![P]\!]_{\mathtt{ok}_1} \mid [\![Q]\!]_{\mathtt{ok}_2} \mid \mathtt{ok}_1?.\mathtt{ok}_2?.\mathtt{ok}^{\uparrow}!\ ] \\
[\![P \rhd Q]\!]_{\mathtt{ok}} & \triangleq & [\ (\textbf{try}\ [\![P]\!]_{\mathtt{ok}}\ \textbf{catch}\ [\![Q]\!]_{\mathtt{ok}}) \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!\ ] \\
[\![\mathit{throw}]\!]_{\mathtt{ok}} & \triangleq & [\ \textbf{throw}.\mathbf{0}\ ] \\
[\![\mathit{skip}]\!]_{\mathtt{ok}} & \triangleq & [\ \mathtt{ok}^{\uparrow}!\ ]
\end{array}
$$

Figure 7.6: Encoding of cCSP Basic Programs in the CC_{exc}.

concurrent programs [75]). To encode a primitive action a we consider some process P_a, specified by some external mapping that takes into account the intended functionality of the primitive action. Instead, the encoding of closed transactions is given by the encoding of the compensable action, described next.

We denote by $[\![T]\!]_{\mathtt{ok,ab,cm,cb}}$ the CC_{exc} process which encodes the cCSP compensable program T. The `ok`, `ab`, `cm` and `cb` are the message labels that capture the signals over which the encoding is defined. The `ok` message is used to signal normal completion of the process, while `ab` signals that the process has **ab**orted execution before completing its task. In the case the process successfully completes its task, it will from then on be ready to receive a `cm` signal (**c**ompensate **me**) which results in the activation of all compensations of the actions in T, in the appropriate order — the reverse order with respect to the order in which the actions to be compensated were originally executed. To signal such compensating task has completed, signal `cb` (**c**ompensate **b**ack) is sent so as to activate compensations of tasks that were executed before T. Given this understanding on the signals involved in the encoding of a compensable program, we may now give a more informed description of the encoding of the closed transaction:

$$[\![\langle T\rangle]\!]_{\mathtt{ok}} \triangleq [\ [\![T]\!]_{\mathtt{ok,ab,cm,cb}} \mid \mathtt{ab}?.\textbf{throw}.\mathbf{0} \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!\ ]$$

which specifies that if an `ab` signal is picked up then an exception is thrown, and that if an `ok` signal is picked up then it is forwarded to the appropriate environment. The compensation signals `cm` and `cb` are discarded, since the closed transaction is not subject to internal compensation due to external errors — only compensable processes are able to activate their compensations upon external failures.

We informally describe the several encodings for compensable programs. The encoding for the compensation pair is as follows:

$$[\![P \div Q]\!]_{\mathtt{ok,ab,cm,cb}} \triangleq [\ \textbf{try}\ [\![P]\!]_{\mathtt{ok}}\ \textbf{catch}\ \mathtt{ab}^{\uparrow}! \mid \mathtt{ok}?.\mathtt{ok}^{\uparrow}!.(\mathtt{cm}^{\uparrow}?.[\![Q]\!]_{\mathtt{cb}} \mid \mathtt{cb}?.\mathtt{cb}^{\uparrow}!)\ ]$$

which specifies that the positive action (P) is to be tentatively run, in a try-catch block, up to the point some exception is thrown (if any). If such an exception is signaled, it is picked up by the try-catch block, resulting in the emission of message `ab` to signal abnormal termination. Otherwise, if no exception is signaled and P successfully terminates then it sends message `ok`, to signal normal completion, directed to the external environment (notice the usage of the

anonymous protected context). At that point, the compensation handler is installed by means of a process ready to receive a cm signal, after which activating the encoding of Q with completion signal cb (notice the subscript). When Q completes it will send message cb, which is forwarded to the external environment.

The encoding of the sequential composition $T_1; T_2$ is as follows:

$$\begin{aligned}[\![T_1; T_2]\!]_{\mathtt{ok,ab,cm,cb}} \triangleq\ & [\ [\![T_1]\!]_{\mathtt{ok_1,ab_1,cm_1,cb}} \mid \mathtt{ab_1?.ab^{\uparrow}!} \mid \\ & \mathtt{ok_1?.}[\![T_2]\!]_{\mathtt{ok,ab,cm,cm_1}} \mid \mathtt{ab?.cm_1!.cb?.ab^{\uparrow}!} \mid \\ & \mathtt{ok?.ok^{\uparrow}!.cm^{\uparrow}?.cm!.cb?.cb^{\uparrow}!}\ ]\end{aligned}$$

The encoding essentially plugs the signals of the encodings of T_1 and T_2 so as to guarantee that: (1) T_2 is only activated after successful completion of T_1 (message $\mathtt{ok_1}$); (2) If T_2 aborts, then the compensation of T_1 is activated (message $\mathtt{cm_1}$); (3) if both processes succeed or one of them fails then the enclosing environment is notified accordingly.

The encoding for the choice construct relies on a private message exchange on messages t and f so as to simulate the internal choice:

$$\begin{aligned}[\![T_1 \oplus T_2]\!]_{\mathtt{ok,ab,cm,cb}} \triangleq\ & [\ \mathtt{t!} + \mathtt{f!} \mid \mathtt{t?.}[\![T_1]\!]_{\mathtt{ok,ab,cm,cb}} \mid \mathtt{f?.}[\![T_2]\!]_{\mathtt{ok,ab,cm,cb}} \mid \\ & \mathtt{ab?.ab^{\uparrow}!} \mid \mathtt{ok?.ok^{\uparrow}!.cm^{\uparrow}?.cm!.cb?.cb^{\uparrow}!}\ ]\end{aligned}$$

Instead, the encoding for parallel composition wires the signals so as wait for completion of both processes before signaling completion of the parallel composition process:

$$\begin{aligned}[\![T_1 \mid T_2]\!]_{\mathtt{ok,ab,cm,cb}} \triangleq\ & [\ [\![T_1]\!]_{\mathtt{ok_1,ab,cm_1,cb_1}} \mid [\![T_2]\!]_{\mathtt{ok_2,ab,cm_2,cb_2}} \mid \\ & \mathtt{ok_1?.ok_2?.ok^{\uparrow}!.cm^{\uparrow}?.(cm_1!} \mid \mathtt{cm_2!} \mid \mathtt{cb_1?.cb_2?.cb^{\uparrow}!)} \mid \\ & \mathtt{ab?.(ok_1?.cm_1!.cb_1?.ab^{\uparrow}!} \mid \mathtt{ok_2?.cm_2!.cb_1?.ab^{\uparrow}!} \mid \mathtt{ab?.ab^{\uparrow}!)}\ \ ]\end{aligned}$$

Notice that if one of the process aborts then, before signaling abortion to the enclosing environment, the other process must either terminate normally (in which case its compensation is activated) or abort.

The encoding of the exception handler construct implements the try-catch behavior:

$$\begin{aligned}[\![T_1 \rhd T_2]\!]_{\mathtt{ok,ab,cm,cb}} \triangleq\ & [\ [\![T_1]\!]_{\mathtt{ok_1,ab_1,cm_1,cb_1}} \mid \mathtt{ok_1?.ok^{\uparrow}!.cm^{\uparrow}?.(cm_1!} \mid \mathtt{cb_1?.cb^{\uparrow}!)} \mid \\ & \mathtt{ab_1?.(}[\![T_2]\!]_{\mathtt{ok_2,ab_2,cm_2,cb_2}} \mid \mathtt{ok_2?.ok^{\uparrow}!.cm^{\uparrow}?.(cm_2!} \mid \mathtt{cb_2?.cb^{\uparrow}!)} \mid \mathtt{ab_2?.ab^{\uparrow}!)}\ ]\end{aligned}$$

ensuring that the exception handler behaves as T_1 if no exception is thrown, and otherwise, behaves as T_2.

We show the inductively defined encoding of compensable programs in Figure 7.7. We leave open the encoding of basic actions, assuming some mapping from basic actions a to processes P_a. However, it is not difficult to conceive how to represent such basic tasks by remote task invocations, namely:

$$[\![a]\!]_{ok} \triangleq \textbf{new}\ \mathit{Server} \cdot \mathsf{a} \Leftarrow (ok^{\downarrow}?().ok^{\uparrow}!() + ko^{\downarrow}?().\textbf{throw}.\mathbf{0})$$

which relies on a service a published at conversation *Server*, which may be defined as follows:

$$\textbf{def}\ \mathsf{a} \Rightarrow (\textbf{try}\ P\ \textbf{catch}\ ko^{\downarrow}!().\textbf{throw}.\mathbf{0})$$

$$
\begin{array}{lcl}
\llbracket P \div Q \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \textbf{try}\ \llbracket P \rrbracket_{\texttt{ok}}\ \textbf{catch}\ \texttt{ab}^{\uparrow}! \mid \\
& & \texttt{ok}?.\texttt{ok}^{\uparrow}!.(\texttt{cm}^{\uparrow}?.\llbracket Q \rrbracket_{\texttt{cb}} \mid \texttt{cb}?.\texttt{cb}^{\uparrow}!)\ ] \\
\llbracket T_1 ; T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \llbracket T_1 \rrbracket_{\texttt{ok}_1,\texttt{ab}_1,\texttt{cm}_1,\texttt{cb}} \mid \\
& & \texttt{ab}_1?.\texttt{ab}^{\uparrow}! \mid \\
& & \texttt{ok}_1?.\llbracket T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cm}_1} \mid \\
& & \texttt{ab}?.\texttt{cm}_1!.\texttt{cb}?.\texttt{ab}^{\uparrow}! \mid \\
& & \texttt{ok}?.\texttt{ok}^{\uparrow}!.\texttt{cm}^{\uparrow}?.\texttt{cm}!.\texttt{cb}?.\texttt{cb}^{\uparrow}!\ ] \\
\llbracket T_1 \oplus T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \texttt{t}! + \texttt{f}! \mid \texttt{t}?.\llbracket T_1 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} \mid \\
& & \texttt{f}?.\llbracket T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} \mid \texttt{ab}?.\texttt{ab}^{\uparrow}! \mid \\
& & \texttt{ok}?.\texttt{ok}^{\uparrow}!.\texttt{cm}^{\uparrow}?.\texttt{cm}!.\texttt{cb}?.\texttt{cb}^{\uparrow}!\ ] \\
\llbracket T_1 \mid T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \llbracket T_1 \rrbracket_{\texttt{ok}_1,\texttt{ab},\texttt{cm}_1,\texttt{cb}_1} \mid \llbracket T_2 \rrbracket_{\texttt{ok}_2,\texttt{ab},\texttt{cm}_2,\texttt{cb}_2} \mid \\
& & \texttt{ok}_1?.\texttt{ok}_2?.\texttt{ok}^{\uparrow}!. \\
& & \quad \texttt{cm}^{\uparrow}?.(\texttt{cm}_1! \mid \texttt{cm}_2! \mid \\
& & \qquad\qquad \texttt{cb}_1?.\texttt{cb}_2?.\texttt{cb}^{\uparrow}!) \mid \\
& & \texttt{ab}?.(\texttt{ok}_1?.\texttt{cm}_1!.\texttt{cb}_1?.\texttt{ab}^{\uparrow}! \mid \\
& & \qquad \texttt{ok}_2?.\texttt{cm}_2!.\texttt{cb}_1?.\texttt{ab}^{\uparrow}! \mid \\
& & \qquad \texttt{ab}?.\texttt{ab}^{\uparrow}!)\ ] \\
\llbracket T_1 \rhd T_2 \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \llbracket T_1 \rrbracket_{\texttt{ok}_1,\texttt{ab}_1,\texttt{cm}_1,\texttt{cb}_1} \mid \\
& & \texttt{ok}_1?.\texttt{ok}^{\uparrow}!. \\
& & \quad \texttt{cm}^{\uparrow}?.(\texttt{cm}_1! \mid \texttt{cb}_1?.\texttt{cb}^{\uparrow}!) \mid \\
& & \texttt{ab}_1?.(\llbracket T_2 \rrbracket_{\texttt{ok}_2,\texttt{ab}_2,\texttt{cm}_2,\texttt{cb}_2} \mid \\
& & \qquad \texttt{ok}_2?.\texttt{ok}^{\uparrow}!. \\
& & \qquad\quad \texttt{cm}^{\uparrow}?.(\texttt{cm}_2! \mid \texttt{cb}_2?.\texttt{cb}^{\uparrow}!) \mid \\
& & \qquad \texttt{ab}_2?.\texttt{ab}^{\uparrow}!)\ ] \\
\llbracket \mathit{throww} \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \texttt{ab}^{\uparrow}!\ ] \\
\llbracket \mathit{skipp} \rrbracket_{\texttt{ok},\texttt{ab},\texttt{cm},\texttt{cb}} & \triangleq & [\ \texttt{ok}^{\uparrow}!.\texttt{cm}^{\uparrow}?.\texttt{cb}^{\uparrow}!\ ]
\end{array}
$$

Figure 7.7: Encoding of cCSP Compensable Programs in the CC_{exc}.

where the service code P itself may be the result of encoding a basic program. In such way we expect to be able to model general distributed nested compensating transactions. In the next section we show an application for such compensating transactions in a service-oriented scenario, where the primitive actions are implemented by means of remote services.

7.5.3 Compensations in a Service-Oriented Application

In this section we show an example of a service-oriented application where we use compensations to model the control structure of the application. We consider a car break scenario (adapted from the SENSORIA [59] automotive case study) which involves an onboard service planner, a car repair service, a tow truck service and a rental car service. We specify the control flow of the example using a cCSP like specification, which allows us to structure the control flow of the scenario using compensations. Then we implement the basic activities in CC which, since the code that implements the cCSP composition operators is directly provided by the encoding of cCSP in CC_{exc}, then gives us a complete CC_{exc} model of the car break scenario.

We informally describe the intended scenario. A car equipped with a service planner on an

onboard computer breaks down on the road. The service planner immediately tries to book a garage repair service, by calling some service available on the network. After that the service planner tries to call a tow truck service so as to arrange for the pick up of the car at its current location, and also tries to rent a car, indicating the repair shop location, so as to allow for the driver to have the rental car waiting for him at the repair shop. On the one hand, if the car rental booking fails, the rest of the operations should nevertheless proceed. On the other hand, if there is no available tow truck, then the garage appointment must be canceled and the car rental gets redirected to the broken down car location.

We implement the service planner in cCSP as follows:

$$
\begin{aligned}
&\langle \mathit{BookRepairShop} \div \mathit{CancelRepairShop};\\
&((\mathit{BookCarRental} \div \mathit{RedirectCarRental}) \rhd \textsf{\textbf{skipp}}\\
&\ |\\
&\ \mathit{OrderTowTruck} \div \mathit{CancelTowTruck})\rangle
\end{aligned}
$$

The code specifies that first the repair shop is to be booked, then the car rental service and the tow truck service are contacted simultaneously (in parallel). Each of such actions is accompanied by the action that compensates them: in the case of the repair shop booking, the compensating action is the canceling of the repair shop appointment; in the case of the tow truck request, the compensating action is the canceling of the request; and in the case of the car rental booking, the action that compensates it is the operation that redirects the rental car to the broken down car location. Notice that if the car rental operation is to fail in any way, then such failure does not influence the remaining operations, as specified by the exception handler construct $\rhd$ which picks up any failure resulting from the operation that rents the car. Notice also the orthogonal usage of the exception handling primitive and the compensation operator.

Informally, if all operations succeed then the code behaves like:

$$\mathit{BookRepairShop}; (\mathit{BookCarRental} \mid \mathit{OrderTowTruck})$$

which corresponds to the execution of the “positive” parts of the code. Instead, if the car rental booking fails and the other operations succeed, then the process behaves like:

$$\mathit{BookRepairShop}; \mathit{OrderTowTruck}$$

which captures the fact that a failure in the car rental does not influence the other operations. Another possibility is that the tow truck reservation fails and other operations succeed, in which case the behavior of the process may be described as:

$$\mathit{BookRepairShop}; \mathit{BookCarRental}; \mathit{RedirectCarRental}; \mathit{CancelRepairShop}$$

which captures the idea that a failure of the tow truck reservation causes the compensation of the other activities to be executed, in the reverse order. Notice that if the repair shop booking fails then the process should have no visible behavior at all.

Building on our encoding for cCSP processes we may now implement the basic activities in $\mathrm{CC}_{\mathrm{exc}}$, abstracting away from the “glue” code which is directly provided by the encoding. Our implementation for the basic activities in $\mathrm{CC}_{\mathrm{exc}}$ must respect the external interface mentioned

previously: they either succeed in which case they send an `ok` message, or they fail in which case they throw an exception. In such way we obtain a $\mathrm{CC}_{\mathrm{exc}}$ model of the car break scenario.

We implement the repair shop basic operations as follows:

$$
\begin{aligned}
&\mathit{BookRepairShop} \triangleq \\
&\quad \textbf{new}\ \mathit{RepairShop} \cdot \mathtt{MakeAppointment} \Leftarrow \\
&\qquad \mathtt{bookRepairOperation}^{\downarrow}!(\mathit{myId}). \\
&\qquad (\mathtt{repairBookingSucceeded}^{\downarrow}?().\mathtt{localDiagnosis}^{\downarrow}!(\mathit{data}).\mathtt{ok}^{\uparrow}!() \\
&\qquad + \\
&\qquad \mathtt{repairBookingFailed}^{\downarrow}?().\textbf{throw}.\mathbf{0})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CancelRepairShop} \triangleq \\
&\quad \textbf{new}\ \mathit{RepairShop} \cdot \mathtt{CancelAppointment} \Leftarrow \\
&\qquad \mathtt{cancelRepairOperation}^{\downarrow}!(\mathit{myId}).\mathtt{repairOperationCanceled}^{\downarrow}?().\mathtt{ok}^{\uparrow}!()
\end{aligned}
$$

The repair shop booking specifies the instantiation of service `MakeAppointment`, available at *RepairShop*. The service instance code sends message `bookRepairOperation`, after which waits to receive either message `repairBookingSucceeded` or message `repairBookingFailed`, informing on the success of the booking. If the booking succeeds a message containing a `localDiagnosis` is sent, after which the enclosing environment is notified of the success via $\uparrow$ directed message `ok` — that will get $\downarrow$ directed when crossing the service conversation access piece. If the booking failed, the enclosing environment is notified of the failure by an exception signal raised by **throw**.

The repair shop canceling operation specifies the instantiation of service `CancelAppointment`, in which messages `cancelRepairOperation` and `repairOperationCanceled` are exchanged, after which the enclosing environment is notified of the success via message `ok` — here we assume compensations do not fail. Notice the external interfaces of processes *BookRepairShop* and *CancelRepairShop* respect the intended invariant: they either send message $\mathtt{ok}^{\downarrow}$ or signal an exception.

We assume some variables are provided by the environment of the process, namely *myId* which identifies the car or driver, and *data* which represents some diagnosis data the service planner locally collects. The implementation of the car rental operations follows similar lines:

$$
\begin{aligned}
&\mathit{BookCarRental} \triangleq \\
&\quad \textbf{new}\ \mathit{CarRental} \cdot \mathtt{RentACar} \Leftarrow \\
&\qquad \mathtt{requestCar}^{\downarrow}!(\mathit{myId}). \\
&\qquad \mathtt{pickUpLocation}^{\downarrow}!(\mathit{shopLoc}). \\
&\qquad (\mathtt{carAvailable}^{\downarrow}?().\mathtt{ok}^{\uparrow}!() \\
&\qquad + \\
&\qquad \mathtt{noCarAvailable}^{\downarrow}?().\textbf{throw}.\mathbf{0})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{RedirectCarRental} \triangleq \\
&\quad \textbf{new}\ \mathit{CarRental} \cdot \mathtt{RedirectCar} \Leftarrow \\
&\qquad \mathtt{changePickUpLocation}^{\downarrow}!(\mathit{myId}, \mathit{myLoc}).\mathtt{pickUpLocationChanged}^{\downarrow}?().\mathtt{ok}^{\uparrow}!()
\end{aligned}
$$

The booking and redirecting operations instantiate services `RentACar` and `RedirectCar`, respec-

tively, and after fulfilling the service protocol, signal success or failure by sending message `ok` or by throwing an exception, respectively. We assume some extra variables are instantiated by the environment, namely *shopLoc* and *myLoc*, representing the locations of the repair shop and of the broken down car. Finally we implement the basic activities for the tow truck operations:

$$
\begin{aligned}
&\mathit{orderTowTruck} \triangleq \\
&\quad \textbf{new}\ \mathit{TowTruck} \cdot \texttt{Order} \Leftarrow \\
&\qquad \texttt{requestTowTruck}^{\downarrow}!(\mathit{myId}). \\
&\qquad (\texttt{towTruckAvailable}^{\downarrow}?().\texttt{fromTo}^{\downarrow}!(\mathit{myLoc}, \mathit{shopLoc}).\texttt{ok}^{\uparrow}!() \\
&\qquad + \\
&\qquad \texttt{noTowTruckAvailable}^{\downarrow}?().\textbf{throw}.\mathbf{0})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{CancelTowTruck} \triangleq \\
&\quad \textbf{new}\ \mathit{TowTruck} \cdot \texttt{CancelOrder} \Leftarrow \\
&\qquad \texttt{cancelTowTruck}^{\downarrow}!(\mathit{myId}).\texttt{towTruckCanceled}^{\downarrow}?().\texttt{ok}^{\uparrow}!()
\end{aligned}
$$

The implementation for the *OrderTowTruck* and *CancelTowTruck* follows similar lines to the other basic activities. We have thus provided a compete model for the car break scenario in $\mathrm{CC}_{\mathrm{exc}}$, illustrating how we may combine the cCSP encoding with the appropriate processes to represent the basic activities.

7.6 Remarks

Primitives to deal with exceptional behavior (for example, closing sessions) are present in several service calculi [12, 13, 31, 71]. Perhaps surprisingly, our exception mechanism, although clearly based on the classical construct for imperative languages, does not seem to have been much explored in process calculi. We believe that it allows us to express many interesting exceptional behavior scenarios, and, combined with the basic communication primitives, support the modeling of protocols that allow for the coordination of exceptional behavior between the several distributed parties involved in a service-oriented application.

We have shown that the language extension with the exception handling primitives still enjoys some essential fundamental properties, namely that our reference observational equivalences (strong and weak bisimilarity) are congruences. We are not aware of the existence of such results in related models of service-oriented computing.

Several proposals of process calculi, both flow and interaction based, natively support long running transactions and forms of compensations [10, 14, 17, 18, 70]. We have shown an encoding of one of such languages that natively supports compensations, namely the cCSP [18], which thus demonstrates that the expressiveness of the CC is enough to cope with such higher level programming constructs. We do not formally state here a proof of correctness of our encoding, as it falls out of scope of this text, but we have reported it in [24].

We acknowledge that some related approaches already aim at typed exception handling models, namely [31], while we are yet to develop an extension of the conversation type system that addresses the Exception Handling Conversation Calculus.

Chapter 8

Conclusions

We close the dissertation with a summary of our contributions and some notes on possible evolutions and future work.

8.1 Summary of Contributions

In this dissertation we have presented a core typed model for expressing and analyzing service and communication based systems, based on the notions of conversation, conversation context, and context-dependent communication. Building on the identification of some general aspects of service-based systems, we propose the Conversation Calculus as a model for service-oriented computation, a process calculus which incorporates abstractions of the several aspects involved by means of carefully chosen programming language primitives. We also propose analysis techniques, namely the conversation type system and the progress proof system, that improve on previous works on aspects that are important from a practical point of view, as such improvements allow us to address highly dynamic configurations that can be found in real service-oriented applications, and that are out of reach of other approaches.

A Model for Service-Oriented Computation

We have focused our presentation of the model on a detailed justification of the concepts involved, on examples that illustrate the expressiveness of our model, and on the semantic theory for our calculus, based on a standard strong bisimilarity. We show how higher-level service-oriented idioms can be programmed using the basic communication primitives of the language, thus supporting the claim that our model is more fundamental than other session based approaches, which include such primitives as native to the languages.

We prove that our chosen syntactical constructs enjoy essential theoretical properties, namely that they are proper functions of behavior. The behavioral semantics allowed us to prove several interesting behavioral identities. Some of these identities suggested a normal form result that clarifies the spatial communication topology of Conversation Calculus systems. Obtaining the reported results took clarifying subtle points of the model, and even some fine tuning. For instance, the refinement of the duality in the synchronization algebra turned out to be essential to prove the split rule, which states that two pieces of the same conversation behave the same as one. The behavioral identities also suggested the establishment of a structural congruence based operational semantics.

We believe that, operationally, the core CC can be seen as a specialized idiom of the π-Calculus [85], if one considers π extended with labeled channels or pattern matching. However, for the purpose of studying communication disciplines for service-oriented computing and their typings, it is much more convenient to adopt a primitive conversation context construct, for it allows to encapsulate closely related communication and computation in a simple way, as well as it allows for conversation identities to be kept implicit until needed.

Conversation Types

Conversation types elucidate the intended dynamic structure of conversations, in particular how freshly instantiated conversations may dynamically engage and dismiss participants, modeling, in a fairly abstract way, the much lower level correlation mechanisms available in Web-Services technology. Conversation types also describe the information and control flow of general service-based collaborations, in particular they may describe the behavior of orchestrations and choreographies. We have established subject reduction and type safety theorems, which entail that well-typed systems follow the defined protocols.

Conversation types extend the notion of binary session types to multiple participants, but discipline their communication by exploiting distinctions between labeled messages in a single shared communication medium, rather than by introducing multiple or indexed more traditional session typed communication channels as, e.g., in the approach of Honda et al. [57]. Our approach allows us to unify the notions of global type and local type, and type highly dynamic scenarios of multiparty concurrent conversations not covered by other approaches. On the other hand, being more abstract and uniform, our type system does not explicitly keep track of participant identities. However, we conceive it is possible to specialize our approach so as to consider extra constraints on projections on types and merges, restricting particular message exchanges to some roles, so as to ensure the protocols are carried out by determined parties.

Progress Proof System

Our progress proof system allows us to single out well-ordered systems, for which we prove a progress property — well-ordered systems never get stuck — even when parties are engaged in multiple interleaved conversations. This implies conversation protocols proceed until they have finished, and that parties never get stuck on a call to a service. The ability to address systems that interleave their participation between dynamically acquired mediums of communication and others was a previously open problem in session types literature, and is crucial, for instance, to allow for the analysis to address systems where parties access local resources and call external services to collaborate on a task. Thus, this ability is essential to address real service-oriented systems, where such requirements are often found, and is out of reach of previous works.

Our examples show how we may specify and analyze challenging configurations, where parties are dynamically engaged to participate in conversations, where the number of parties that are simultaneously interacting in a conversation may actually depend on some runtime condition, and where parties interleave their participation in several of such conversations, including in conversations to which they dynamically gained access to.

To support the claim that our analysis techniques are not specific to our proposed model, we show that they can be applied in a more canonical setting, while essentially retaining their

expressiveness. Also we have presented a language extension that includes exception handling primitives, and we show how it allow us to accommodate compensations, and thus give support for long-running transactions which are often required to model the business protocols behind service-oriented applications.

8.2 Evolutions and Future Work

The work presented in this dissertation may be seen as a first step in the direction of supporting the specification and analysis of general dynamic multiparty interaction in communication based systems. Naturally, there is still plenty of room for improvement and further development, starting from the fact that we have left out of our focus important features such as failure handling, security, and interoperation mechanisms.

There are several further developments of our model and techniques that may motivate future work. For instance, it would be interesting to develop inference techniques for conversation types and for event orderings. Also, extending the conversation type discipline so as to cope with the exception handling fragment may prove to be a worthwhile challenge. At a more fine-grained technical level we may consider generalizing the recursion constructs so as to allow for names to be passed from one iteration to another (parameterized recursion), and lift the analysis techniques so as to consider such setting. Also, exploring variants of the behavioral merge relation, namely at the level of how alternative behaviors are handled, may lead to further generalization of how protocols are distributed throughout the parties. Namely, it would be interesting to consider shuffling of branches/choices with other behaviors so as to allow for composing behaviors subject to possible futures.

One interesting possibility for future work at a more conceptual level would be to have a means to allow for fragments of a distributed protocol to be local to a part of a system, hence visible only to the parties that intervene in such fragments, while invisible for other parties. In such way, we would still verify that a global public protocol is followed by the distributed parties, and allow for some local hidden interactions to take place in between the public ones. It may be argued that we can already cope with such scenarios by interleaving a conversation shared by all with a conversation shared by a few, since conversation names can be restricted to a part of a system. However, given the premise that such public and hidden interactions are related, we may ask why should they take place in distinct conversations?

One idea that may be worthwhile to pursue so as to support this, would be to extend the language with restriction over message labels (cf., conversation name restriction). Ideally, this would allow for parts of a protocol that are realized on a determined label to be local to the parties that intervene in the message exchanges defined for that label. As for protocols on restricted conversations, when closing the scope of a restricted label we would elide all messages referring such label from the behavioral types, while making sure that the interactions are closed, hence that all such message message types have polarity τ. In such way, we would support, for instance, service definitions that export a public interface to the clients, which may specify message exchanges which will be carried out by several parties, but that may internally resort to message exchanges that are not visible to the client. Notice that labels would still not be passed in messages, so, in some sense, we would be considering CCS [74] like restriction in message labels (no label mobility), while we would keep π-Calculus like restriction in conversation names

(conversation name mobility).

Such a feature could prove to be important from a security point of view. For example, consider a group of two trusted parties that are interacting in a conversation and that intend to ask a third untrusted party to join the conversation. The three parties may share a global public conversation protocol, while the two trusted parties may extend such protocol with some local message exchanges, so as to allow for some sensitive information to nevertheless be exchanged.

On more general lines of research it would be interesting to see if there is a notion of typability (perhaps extending and combining conversation types and event orderings) that precisely characterizes the abstract behavior of CC processes, in the sense that if two processes have the same types then they have the same behavior (they are bisimilar). This would allow us to use type-checking techniques to test behavioral equivalences, and have a means to precisely characterize safe-substitution of subsystems up to behavior: we may replace a subsystem with another, just as long they are typed (behave) in the same way, and obtain a system that behaves as the original one. Notice that, in our current setting, if we are to replace subsystems by others that are typed and ordered in the same way as the initial one, then our properties (conversation fidelity and progress) still hold.

Another line of work which would be interesting to pursue at the level of the characterization of the abstract semantics of the model, would be the use of logics to behaviorally characterize CC systems — as introduced by Hennessy et al. [52] — and also spatially characterize CC systems — as introduced by Caires et al. [25] and by Cardelli et al. [33] and further pursued in [21, 22] — given that behavioral and spatial configurations are in some sense interlinked in the CC model. We would then be able — following the approach of Caires [19] for which we have developed a prototype model-checker [88] — to model-check spatial behavioral properties of CC systems. We have already began using the prototype model-checker to verify conformance of CC systems with respect to WS-CDL like choreographies, using translations of CC systems into π-calculus specifications and of WS-CDL choreographies into spatial logic formulae, where we prove that the check in the original models is equivalent to the check in the translations (described in [5]). Also, it would be interesting to investigate the relation of such spatial behavioral logical characterizations of CC processes and the behavioral semantics, a line of work we have pursued previously in other models [26, 89], for instance by developing standard observational equivalences using the notion of external observer that may interact with the system (and observe some barbs) and see if such an external observer of CC systems can, in an extensional way, acquire knowledge of how such systems are spatially configured.

Ultimately, our developments would gain their full relevance if they were transported to practice. To that end it would be interesting to see if our notion of conversations can be mapped into a mainstream service-oriented computing programming language, for instance, by using correlation tokens to encode conversations (e.g., encoding conversations in WS-BPEL [4]) or using an intermediate language based on correlations to establish the comparison (e.g., COWS [71]). We establish decidability of our techniques (if bound names are type/ordering annotated) as our rules are syntax directed and auxiliary operations are finitary, namely the merge relation, and typability/well-ordering is witnessed by a proof tree (as usual). As such, we intend to pursue the development of a prototype implementation of a programming language inspired by the Conversation Calculus, and also of a conversation type checker and progress proof system checker, so as to serve as a proof-of-concept of the applicability of our ideas. This work has been started

by Lourenço et al., already leading to other interesting contributions, namely a type inference algorithm for conversation types [72].

Bibliography

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL 2002, 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 104–115. ACM Press, 2001.

[2] L. Acciai, C. Bodei, M. Boreale, R. Bruni, and H. Vieira. *The* SENSORIA *book*, chapter Static Analysis Techniques for Session-Oriented Calculi. Springer-Verlag, 2010. To appear.

[3] L. Acciai and M. Boreale. A Type System for Client Progress in a Service-Oriented Calculus. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 642–658. Springer-Verlag, 2008.

[4] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. Liu, R. Khalaf, D. K. M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Standard, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[5] M. Bartoletti, L. Caires, I. Lanese, F. Mazzanti, D. Sangiorgi, H. Vieira, and R. Zunino. *The* SENSORIA *book*, chapter Tools and Verification. Springer-Verlag, 2010. To appear.

[6] J. Beatty, G. Kakivaya, D. Kemp, T. Kuehnel, B. Lovering, B. Roe, C. John, J. Schlimmer, G. Simonnet, D. Walter, J. Weast, Y. Yarmosh, and P. Yendluri. Web Services Dynamic Discovery. Technical report, Microsoft Corporation and Co-Developers, 2005. http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf.

[7] T. Bellwood, S. Capell, L. Clement, J. Colgrave, M. Dovey, D. Feygin, A. Hately, R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen, T. Rogers, K. Sycara, P. Wenzel, and Z. Wu. Universal Description, Discovery, and Integration Version 3.0.2. Technical report, OASIS Committee Draft, 2005. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm.

[8] Benjamin C. Pierce and D. Turner. Pict: a Programming Language Based on the π-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[9] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In F. van Breugel and

M. Chechik, editors, *CONCUR 2008, 19th International Conference on Concurrency Theory, Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2008.

[10] L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long-Running Transactions. In E. Najm, U. Nestmann, and P. Stevens, editors, *FMOODS 2003, 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Proceedings*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer-Verlag, 2003.

[11] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In G. Barthe and C. Fournet, editors, *TGC 2007, Third International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 240–256. Springer-Verlag, 2008.

[12] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. SCC: a Service Centered Calculus. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM 2006, Third International Workshop on Web Services and Formal Methods, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer-Verlag, 2006.

[13] M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and Pipelines for Structured Service Programming. In G. Barthe and F. de Boer, editors, *FMOODS 2008, 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Proceedings*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer-Verlag, 2008.

[14] R. Bruni, H. Melgratti, and U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In J. Palsberg and M. Abadi, editors, *POPL 2005, 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 209–220. ACM Press, 2005.

[15] R. Bruni and L. Mezzina. Types and Deadlock Freedom in a Calculus of Services, Sessions and Pipelines. In J. Meseguer and G. Rosu, editors, *AMAST 2008, 12th International Conference on Algebraic Methodology and Software Technology, Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2008.

[16] M. Bugliesi, G. Castagna, and S. Crafa. Access Control for Mobile Agents: The Calculus of Boxed Ambients. *ACM Transactions on Programming Languages and Systems*, 26(1):57–124, 2004.

[17] M. Butler and C. Ferreira. A Process Compensation Language. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *IFM 2000, Second International Conference on Integrated Formal Methods, Proceedings*, volume 1945 of *Lecture Notes in Computer Science*, pages 61–76. Springer-Verlag, 2000.

[18] M. Butler, C. Hoare, and C. Ferreira. A Trace Semantics for Long-Running Transactions. In A. Abdallah, C. Jones, and J. Sanders, editors, *Communicating Sequential Processes: The*

First 25 Years, Symposium on the Occasion of 25 Years of CSP, Revised Invited Papers, volume 3525 of *Lecture Notes in Computer Science*, pages 133–150. Springer-Verlag, 2005.

[19] L. Caires. Behavioral and Spatial Observations in a Logic for the π-Calculus. In I. Walukiewicz, editor, *FOSSACS 2004, 7th International Conference on Foundations of Software Science and Computation Structures, Proceedings*, volume 2987 of *Lecture Notes in Computer Science*, pages 72–89. Springer-Verlag, 2004.

[20] L. Caires. Spatial-Behavioral Types for Concurrency and Resource Control in Distributed Systems. *Theoretical Computer Science*, 402(2-3):120–141, 2008.

[21] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part I). *Information and Computation*, 186(2):194–235, 2003.

[22] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). *Theoretical Computer Science*, 322(3):517–565, 2004.

[23] L. Caires, R. De Nicola, R. Pugliese, V. Vasconcelos, and G. Zavattaro. *The* SENSORIA *book*, chapter Core Calculi for Service-Oriented Computing. Springer-Verlag, 2010. To appear.

[24] L. Caires, C. Ferreira, and H. Vieira. A Process Calculus Analysis of Compensations. In C. Kaklamanis and F. Nielson, editors, *TGC 2008, Fourth International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 87–103. Springer-Verlag, 2009.

[25] L. Caires and L. Monteiro. Verifiable and Executable Logic Specifications of Concurrent Objects in $\mathcal{L}_\pi$. In C. Hankin, editor, *ESOP 1998, 7th European Symposium on Programming, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag, 1998.

[26] L. Caires and H. Vieira. Extensionality of Spatial Observations in Distributed Systems. *Electronic Notes in Theoretical Computer Science*, 175(3):131–149, 2007.

[27] L. Caires and H. Vieira. Conversation Types. In G. Castagna, editor, *ESOP 2009, 18th European Symposium on Programming, Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 285–300. Springer-Verlag, 2009.

[28] L. Caires and H. Vieira. Conversation Types. *Theoretical Computer Science*, 2010. To appear.

[29] S. Capecchi, M. Coppo, M. Dezani-Ciancaglini, S. Drossopoulou, and E. Giachino. Amalgamating Sessions and Methods in Object-Oriented Languages with Generics. *Theoretical Computer Science*, 410(2-3):142–167, 2009.

[30] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In R. De Nicola, editor, *ESOP 2007, 16th European Symposium on Programming, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2007.

[31] M. Carbone, K. Honda, and N. Yoshida. Structured Interactional Exceptions in Session Types. In F. van Breugel and M. Chechik, editors, *CONCUR 2008, 19th International Conference on Concurrency Theory, Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer-Verlag, 2008.

[32] M. Carbone and S. Maffeis. On the Expressive Power of Polyadic Synchronisation in π-Calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[33] L. Cardelli and A. Gordon. Anytime, Anywhere: Modal Logics for Mobile Ambients. In *POPL 2000, 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 365–377. ACM Press, 2000.

[34] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of Session Types. In A. Porto and F. López-Fraguas, editors, *PPDP'09, 11th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, Proceedings*, pages 219–230. ACM Press, 2009.

[35] G. Castagna, N. Gesbert, and L. Padovani. A Theory of Contracts for Web Services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

[36] G. Castagna, J. Vitek, and F. Nardelli. The Seal Calculus. *Information and Computation*, 201(1):1–54, 2005.

[37] S. Chaki, S. Rajamani, and J. Rehof. Types as Models: Model Checking Message-Passing Programs. In *POPL 2002, 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 45–57. ACM Press, 2002.

[38] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language Version 2.0. Technical report, W3C Recommendation, 2007. http://www.w3.org/TR/wsdl20/.

[39] R. Corin, P.-M. Deniélou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CSF 2007, 20th IEEE Computer Security Foundations Symposium, Proceedings*, pages 170–186. IEEE Computer Society, 2007.

[40] M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On Progress for Structured Communications. In G. Barthe and C. Fournet, editors, *TGC 2007, Third International Symposium on Trustworthy Global Computing, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 257–275. Springer-Verlag, 2008.

[41] M. Dezani-Ciancaglini, S. Drossopoulou, D. Mostrous, and N. Yoshida. Objects and Session Types. *Information and Computation*, 207(5):595–641, 2009.

[42] E. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[43] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. In Y. Berbers and W. Zwaenepoel, editors, *Proceedings of the EuroSys 2006 Conference*, pages 177–190. ACM Press, 2006.

[44] C. Ferreira, I. Lanese, A. Ravara, H. Vieira, and G. Zavattaro. *The* SENSORIA *book*, chapter Advanced Mechanisms for Service Combination and Transactions. Springer-Verlag, 2010. To appear.

[45] J. Fiadeiro, A. Lopes, and L. Bocchi. A Formal Approach to Service Component Architecture. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM 2006, Third International Workshop on Web Services and Formal Methods, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer-Verlag, 2006.

[46] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In J. Jeuring and S. Peyton Jones, editors, *AFP 2002, 4th International School on Advanced Functional Programming, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer-Verlag, 2003.

[47] P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In A. Bossi and M. Maher, editors, *PPDP'06, 8th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, Proceedings*, pages 61–72. ACM Press, 2006.

[48] S. Gay and M. Hole. Subtyping for Session Types in the π-Calculus. *Acta Informatica*, 42(2-3):191–225, 2005.

[49] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[50] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. Simple Object Access Protocol Version 1.2. Technical report, W3C Recommendation, 2007. http://www.w3.org/TR/soap/.

[51] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro. SOCK: A Calculus for Service Oriented Computing. In A. Dan and W. Lamersdorf, editors, *ICSOC 2006, 4th International Conference on Service-Oriented Computing, Proceedings*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 2006.

[52] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[53] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173(1):82–120, 2002.

[54] K. Honda. Types for Dyadic Interaction. In E. Best, editor, *CONCUR 1993, 4th International Conference on Concurrency Theory, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer-Verlag, 1993.

[55] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *ESOP 1998, 7th European Symposium on Programming, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer-Verlag, 1998.

[56] K. Honda and N. Yoshida. A Uniform Type Structure for Secure Information Flow. *ACM Transactions on Programming Languages and Systems*, 29(6), 2007.

[57] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In G. Necula and P. Wadler, editors, *POPL 2008, 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 273–284. ACM Press, 2008.

[58] A. Igarashi and N. Kobayashi. A Generic Type System for the π-Calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004.

[59] IP Sensoria Project. http://www.sensoria-ist.eu/.

[60] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto. Web Services Choreography Description Language Version 1.0. Technical report, W3C Candidate Recommendation, 2005. http://www.w3.org/TR/ws-cdl-10/.

[61] D. Kitchin, W. Cook, and J. Misra. A Language for Task Orchestration and Its Semantic Properties. In C. Baier and H. Hermanns, editors, *CONCUR 2006, 17th International Conference on Concurrency Theory, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 477–491. Springer-Verlag, 2006.

[62] N. Kobayashi. A Partially Deadlock-Free Typed Process Calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998.

[63] N. Kobayashi. A New Type System for Deadlock-Free Processes. In C. Baier and H. Hermanns, editors, *CONCUR 2006, 17th International Conference on Concurrency Theory, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer-Verlag, 2006.

[64] N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-Calculus. In *POPL 1996, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Proceedings*, pages 358–371. ACM Press, 1996.

[65] N. Kobayashi, S. Saito, and E. Sumii. An Implicitly-Typed Deadlock-Free Process Calculus. In C. Palamidessi, editor, *CONCUR 2000, 11th International Conference on Concurrency Theory, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, 2000.

[66] N. Kobayashi, K. Suenaga, and L. Wischik. Resource Usage Analysis for the π-Calculus. *Logical Methods in Computer Science*, 2(3):1–42, 2006.

[67] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *VLDB 1990, 16th International Conference on Very Large Data Bases, Proceedings*, pages 95–106. Morgan Kaufmann, 1990.

[68] I. Lanese, F. Martins, V. Vasconcelos, and A. Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. In *SEFM 2007, Fifth IEEE International Conference on Software Engineering and Formal Methods, Proceedings*, pages 305–314. IEEE Computer Society Press, 2007.

[69] I. Lanese, A. Ravara, and H. Vieira. *The* SENSORIA *book*, chapter Behavioral Theory for Session-Oriented Calculi. Springer-Verlag, 2010. To appear.

[70] C. Laneve and G. Zavattaro. Foundations of Web Transactions. In V. Sassone, editor, *FOSSACS 2005, 8th International Conference on Foundations of Software Science and Computational Structures, Proceedings*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer-Verlag, 2005.

[71] A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In R. De Nicola, editor, *ESOP 2007, 16th European Symposium on Programming, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer-Verlag, 2007.

[72] L. Lourenço and L. Caires. Inference of Conversation Types for Distributed Multiparty Systems. In *PLACES'10, 3rd International Workshop in Programming Language Approaches to Concurrency and Communication-cEntric Software, Proceedings*, 2010. To appear.

[73] N. Lynch. Fast Allocation of Nearby Resources in a Distributed System. In *STOC 1980, Twelfth Annual ACM Symposium on Theory of Computing, Proceedings*, pages 70–81. ACM Press, 1980.

[74] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[75] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[76] R. Milner. Elements of Interaction: Turing Award Lecture. *Communications of the ACM*, 36(1):78–89, 1993.

[77] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.

[78] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[79] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I + II. *Information and Computation*, 100(1):1–77, 1992.

[80] L. Padovani. Session Types at the Mirror. In *ICE 2009, 2nd International Workshop on Interaction and Concurrency Experience, Proceedings*, pages 71–86. EPTCS 12, 2009.

[81] D. Park. Concurrency and Automata on Infinite Sequences. In P. Deussen, editor, *TCS 1981, 5th GI-Conference on Theoretical Computer Science, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1981.

[82] C. Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[83] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[84] B. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996.

[85] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[86] P. Sewell, J. Leifer, K. Wansbrough, F. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4-5):547–612, 2007.

[87] P. Sewell and J. Vitek. Secure Composition of Untrusted Code: Box π, Wrappers, and Causality. *Journal of Computer Security*, 11(2):135–188, 2003.

[88] Spatial Logic Model Checker. http://www-ctp.di.fct.unl.pt/slmc/.

[89] E. Tuosto and H. Vieira. An Observational Model for Spatial Logics. *Electronic Notes in Theoretical Computer Science*, 142:229–254, 2006.

[90] A. Vallecillo, V. Vasconcelos, and A. Ravara. Typing the Behavior of Software Components using Session Types. *Fundamenta Informaticae*, 73(4):583–598, 2006.

[91] W. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, and E. Verbeek. Conformance Checking of Service Behavior. *ACM Transactions on Internet Technology*, 8(3), 2008.

[92] V. Vasconcelos. Fundamentals of Session Types. In M. Bernardo, L. Padovani, and G. Zavattaro, editors, *SFM 2009, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Advanced Lectures*, volume 5569 of *Lecture Notes in Computer Science*, pages 158–186. Springer-Verlag, 2009.

[93] V. Vasconcelos and R. Bastos. Core-TyCO, The Language Definition, Version 0.1. DI/FCUL TR 98-3, Department of Computer Science, University of Lisbon, 1998.

[94] V. Vasconcelos, S. Gay, and A. Ravara. Type Checking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science*, 368(1-2):64–87, 2006.

[95] H. Vieira, L. Caires, and J. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In S. Drossopoulou, editor, *ESOP 2008, 17th European Symposium on Programming, Proceedings*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2008.

[96] G. Winskel. Event Structure Semantics for CCS and Related Languages. In M. Nielsen and E. Schmidt, editors, *ICALP 1982, 9th Colloquium on Automata, Languages and Programming, Proceedings*, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer-Verlag, 1982.

[97] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *ASA/MA '99, 1st International Symposium on Agent Systems and Applications / 3rd International Symposium on Mobile Agents*, pages 2–12. IEEE Computer Society, 1999.

[98] N. Yoshida. Graph Types for Monadic Mobile Processes. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science, 16th Conference, Proceedings*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer-Verlag, 1996.

Appendix A

Proofs

A.1 Chapter 3

Proposition 3.2.14 (Behavioral Type Merge Relation Properties)
(repetition of the statement in page 51)

The behavioral types merge relation is commutative and associative:

(1). If $B = B_1 \bowtie B_2$ then $B = B_2 \bowtie B_1$.

(2). If $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ then there is B'' such that $B'' = B_2 \bowtie B_3$ and $B = B_1 \bowtie B''$.

Proof.

(*1*). Follows immediately from the definition.

(*2*). By induction on the derivation of $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$. Follows lines similar to the proof of Proposition 5.2.19(*2*) (see Appendix A.3). ∎

Proposition 3.3.3 (Weakening)
(repetition of the statement in page 58)

Let P be a well-typed process such that $P :: L_1$. If exponential output located type $\star L_2^!$ is such that L_1 and $\star L_2^!$ are apart, hence $L_1 \# \star L_2^!$, then $P :: L_1 \mid \star L_2^!$.

Proof. By induction on the length of the derivation of $P :: L_1$. Follows lines similar to the proof of Proposition 5.3.3 (see Appendix A.3). ∎

Lemma 3.3.4 (Substitution Lemma)
(repetition of the statement in page 58)

Let P be a well-typed process such that $P :: L \mid x : C$ and $x \notin dom(L)$. If there is type L' such that $L' = L \bowtie a : C$ then $P\{x/a\} :: L'$.

Proof. By induction on the length of the derivation of $P :: L \mid x : C$. We show the case of rule (*Output*), when P is of the form $b \cdot l!(x).P$.

(*Rule* (*Output*))

We have:

$$b \cdot l!(x).Q :: L \mid x : C \tag{A.1.3.1}$$

where $x \notin dom(L)$. We have (A.1.3.1) is derived from:

$$Q :: L_1 \bowtie b : [B] \tag{A.1.3.2}$$

and:

$$L \mid x : C \equiv (L_1 \bowtie b : [!\, l(C').B]) \bowtie x : C' \tag{A.1.3.3}$$

In order to apply the induction hypothesis we first characterize types L_1, C and L. We separate L_1 into two parts L'' and $x : C''$ such that L'' does not mention x (i.e., $x \notin dom(L'')$):

$$L_1 \equiv L'' \mid x : C'' \tag{A.1.3.4}$$

From (A.1.3.3) we have that $x : C$ is the result of the merge of $x : C'$ with the type that L_1 specifies for x (which we identify in (A.1.3.4) as C''), hence:

$$x : C = x : C'' \bowtie x : C' \tag{A.1.3.5}$$

Also, since L'' does not mention x and $x \neq b$ we have that $((L'' \mid x : C'') \bowtie b : [!\, l(C').B]) \bowtie x : C'$ yields the same type as $(L'' \bowtie b : [!\, l(C').B]) \mid (x : C'' \bowtie x : C')$, hence from (A.1.3.3) we have:

$$L \equiv L'' \bowtie b : [!\, l(C').B] \tag{A.1.3.6}$$

We now assume the hypothesis in the statement of the Lemma: there is type L' such that:

$$L' = L \bowtie a : C \tag{A.1.3.7}$$

From (A.1.3.7) and (A.1.3.6) and since C'' is a partial view of C (A.1.3.5), we conclude there is type L'_1 such that:

$$L'_1 = (L'' \bowtie a : [B]) \bowtie a : C'' \tag{A.1.3.8}$$

We rewrite (A.1.3.2) considering (A.1.3.4), using the subsumption rule, and obtain:

$$Q :: (L'' \bowtie b : [B]) \mid x : C'' \tag{A.1.3.9}$$

By induction hypothesis on (A.1.3.9) and (A.1.3.8) we conclude:

$$Q\{x/a\} :: (L'' \bowtie b : [B]) \bowtie a : C'' \tag{A.1.3.10}$$

from which we have:

$$Q\{x/a\} :: (L'' \bowtie a : C'') \bowtie b : [B] \tag{A.1.3.11}$$

From (A.1.3.11) using rule (*Output*) we derive:

$$b \cdot l!(a).Q\{x/a\} :: ((L'' \bowtie a : C'') \bowtie b : [!\, l(C).B]) \bowtie a : C' \tag{A.1.3.12}$$

from which we have:

$$b \cdot l!(a).Q\{x/a\} :: (L'' \bowtie b : [!\, l(C).B]) \bowtie (a : C' \bowtie a : C'') \tag{A.1.3.13}$$

From (A.1.3.13) and (A.1.3.6) and (A.1.3.5) we conclude:

$$b \cdot l!(a).Q\{x/a\} :: L \bowtie a : C \tag{A.1.3.14}$$

which completes the proof for this case. ■

We start by stating some auxiliary results to the proof of Theorem 3.3.5. Such auxiliary results allows us to characterize the types associated to processes that exhibit a determined behavior. Then, using the type information, we may explain the several kinds of internal interactions in the proof of Theorem 3.3.5.

Lemma A.1.4
If $p\,l(C).B <: B'$ then there are B_1, B_2 such that $B' \equiv B_1 \mid p\,l(C).B_2$ and $B <: B_1 \mid B_2$.

Proof. By induction on the length of the derivation of $p\,l(C).B <: B'$. Follows expected lines. ■

Lemma A.1.5
Let L be a type such that $L = L_1 \bowtie L_2$. If there is L' such that $L <: L'$ then there are L'_1 and L'_2 such that $L_1 <: L'_1$ and $L_2 <: L'_2$ and $L' = L'_1 \bowtie L'_2$.

Proof. Follows by induction on the length of the derivation of $L <: L'$ in expected lines. ■

Lemma A.1.6
Let P be a process such that $P :: L$. If $P \xrightarrow{c \cdot l?(a)} Q$ then there are L', C, B such that $L \equiv L' \bowtie c : [?\,l(C).B]$. Furthermore if there is $L'' = (L' \bowtie c : [B]) \bowtie a : C$ then $Q :: L''$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c \cdot l?(a)} Q$. We show the case when the transition originates in an input prefix and in one component of a parallel composition.

(*Case* $c \cdot l?(x).P \xrightarrow{c \cdot l?(a)} P\{x/a\}$)

We have that:

$$c \cdot l?(x).P :: L \qquad \text{and} \qquad c \cdot l?(x).P \xrightarrow{c \cdot l?(a)} P\{x/a\} \tag{A.1.6.1}$$

From (A.1.6.1) and considering rule (*Input*) we have there is L', C, B such that:

$$L' \bowtie c : [?\,l(C).B] <: L \tag{A.1.6.2}$$

and:

$$c \cdot l?(x).P :: L' \bowtie c : [?\,l(C).B] \qquad \text{and} \qquad P :: (L' \bowtie c : [B]) \mid x : C \tag{A.1.6.3}$$

From (A.1.6.2) and considering Lemma A.1.5 and Lemma A.1.4 we conclude there are L'', B', B'' such that:

$$L \equiv L'' \bowtie c : [B' \mid ?\,l(C).B''] \tag{A.1.6.4}$$

and:

$$L' <: L'' \qquad \text{and} \qquad B <: B' \mid B'' \tag{A.1.6.5}$$

We rewrite (A.1.6.4) to:

$$L \equiv (L'' \bowtie c : [B']) \bowtie c : [?\, l(C).B''] \tag{A.1.6.6}$$

Let us now consider there is L'' such that:

$$L''' \equiv ((L'' \bowtie c : [B']) \bowtie c : [B'']) \bowtie a : C \tag{A.1.6.7}$$

Since $L' <: L''$ and $B <: B' \mid B''$ and from (A.1.6.7) we directly have that there is L_1 such that

$$L_1 \equiv (L' \bowtie c : [B]) \bowtie a : C \tag{A.1.6.8}$$

Considering Lemma 3.3.4 we then have:

$$P\{x/a\} :: (L' \bowtie c : [B]) \bowtie a : C \tag{A.1.6.9}$$

From (A.1.6.9) we conclude:

$$P\{x/a\} :: (L'' \bowtie c : [B' \mid B'']) \bowtie a : C \tag{A.1.6.10}$$

hence:

$$P\{x/a\} :: ((L'' \bowtie c : [B']) \bowtie c : [B'']) \bowtie a : C \tag{A.1.6.11}$$

which completes the proof for this case.

($Case$ $P' \mid R \xrightarrow{c \cdot l?(a)} Q' \mid R$)

We have that:

$$P' \mid R :: L \qquad \text{and} \qquad P' \mid R \xrightarrow{c \cdot l?(a)} Q' \mid R \tag{A.1.6.12}$$

which is derived from:

$$P' \xrightarrow{c \cdot l?(a)} Q' \tag{A.1.6.13}$$

Considering (A.1.6.12) and rule (Par) we have that there is L_1, L_2 such that $L_1 \bowtie L_2 <: L$ and:

$$P' \mid R :: L_1 \bowtie L_2 \qquad \text{and} \qquad P' :: L_1 \qquad \text{and} \qquad R :: L_2 \tag{A.1.6.14}$$

By induction hypothesis on (A.1.6.13) and (A.1.6.14) we have that there is L_1', C, B such that:

$$L_1 \equiv L_1' \bowtie c : [?\, l(C).B] \tag{A.1.6.15}$$

From $L_1 \bowtie L_2 <: L$ we conclude:

$$(L_1' \bowtie c : [?\, l(C).B]) \bowtie L_2 <: L \tag{A.1.6.16}$$

which leads to:

$$(L_1' \bowtie L_2) \bowtie c : [?\, l(C).B] <: L \tag{A.1.6.17}$$

which, considering Lemma A.1.5 and Lemma A.1.4 gives us there are L', B', B'' such that:

$$L \equiv L' \bowtie c : [B' \mid ?\, l(C).B''] \tag{A.1.6.18}$$

where:

$$(L_1' \bowtie L_2) <: L' \tag{A.1.6.19}$$

and:

$$B <: B' \mid B'' \tag{A.1.6.20}$$

From (A.1.6.18), (A.1.6.15) and $L_1 \bowtie L_2 <: L$ we have that there is $(L_1' \bowtie c : [B]) \bowtie a : C$, and thus by induction hypothesis we conclude:

$$Q' :: (L_1' \bowtie c : [B]) \bowtie a : C \tag{A.1.6.21}$$

From (A.1.6.21) and (A.1.6.14) we derive:

$$Q' \mid R :: ((L_1' \bowtie c : [B]) \bowtie a : C) \bowtie L_2 \tag{A.1.6.22}$$

which leads to:

$$Q' \mid R :: ((L_1' \bowtie c : [B]) \bowtie a : C) \bowtie L_2 \tag{A.1.6.23}$$

and hence:

$$Q' \mid R :: ((L_1' \bowtie L_2) \bowtie c : [B]) \bowtie a : C \tag{A.1.6.24}$$

From (A.1.6.24), (A.1.6.19) and (A.1.6.20) we conclude:

$$Q' \mid R :: (L' \bowtie c : [B' \mid B'']) \bowtie a : C \tag{A.1.6.25}$$

which completes the proof for this case. ■

Lemma A.1.7
Let P be a process such that $P :: L$. If $P \xrightarrow{c \cdot l!(a)} Q$ then there are L', C, B such that $L \equiv L' \bowtie c : [!\, l(C).B]$ and there is L'' such that $L' \equiv L'' \bowtie a : C$ and $Q :: L'' \bowtie c : [B]$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c \cdot l!(a)} Q$. We show the cases when the transition results from a output prefix.

(*Case* $c \cdot l!(a).P' \xrightarrow{c \cdot l!(a)} P'$)

We have that:

$$c \cdot l!(a).P' :: L \qquad \text{and} \qquad c \cdot l!(a).P' \xrightarrow{c \cdot l!(a)} P' \tag{A.1.7.1}$$

From (A.1.7.1) and considering rule (*Output*) we have there is L', C, B such that:

$$(L' \bowtie c : [!\, l(C).B]) \bowtie a : C <: L \tag{A.1.7.2}$$

and:

$$c \cdot l!(a).P' :: (L' \bowtie c : [!\, l(C).B]) \bowtie a : C \qquad \text{and} \qquad P' :: L' \bowtie c : [B] \tag{A.1.7.3}$$

We rewrite (A.1.7.2) to:

$$(L' \bowtie a : C) \bowtie c : [!\, l(C).B] <: L \tag{A.1.7.4}$$

Then considering Lemma A.1.5 and Lemma A.1.4 we conclude there are L'', B', B'' such that:

$$L \equiv L'' \bowtie c : [B' \mid !\, l(C).B''] \tag{A.1.7.5}$$

where:

$$L' \bowtie a : C <: L'' \tag{A.1.7.6}$$

and:

$$B <: B' \mid B'' \tag{A.1.7.7}$$

From (A.1.7.6) we have that there is L_1 such that $L' <: L_1$ and:

$$L'' \equiv L_1 \bowtie a : C \tag{A.1.7.8}$$

From (A.1.7.3), $L' <: L_1$ and (A.1.7.7) we conclude:

$$P' :: L_1 \bowtie c : [B' \mid B''] \tag{A.1.7.9}$$

which completes the proof for this case. ■

Lemma A.1.8
Let P be a process such that $P :: L$. If $P \xrightarrow{(\nu a)c \cdot l!(a)} Q$ then there are L', C, B such that $L \equiv L' \bowtie c : [!\, l(C).B]$ and there are B', C' such that $closed(B')$, $a : [B'] = a : C' \bowtie a : C$ and $Q :: (L' \bowtie c : [B]) \mid a : C'$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{(\nu a)c \cdot l!(a)} Q$. The proof follows similar lines to that of Lemma A.1.7. We show the case of rule (*Open*) (Figure 2.2).

(*Case* $(\boldsymbol{\nu} a)P' \xrightarrow{(\nu a)c \cdot l!(a)} Q'$)

We have that:

$$(\boldsymbol{\nu} a)P' :: L \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' \xrightarrow{c \cdot l!(a)} Q' \tag{A.1.8.1}$$

which is derived from:

$$P' \xrightarrow{c \cdot l!(a)} Q' \tag{A.1.8.2}$$

From (A.1.8.1) and considering rule (*Res*) we have that there is L', B such that:

$$L' <: L \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' :: L' \qquad \text{and} \qquad P' :: L' \mid a : [B] \tag{A.1.8.3}$$

and $closed(B)$. Considering Lemma A.1.7 and (A.1.8.2) and (A.1.8.3) we have that there are L_1, C, B' such that either:

$$L' \mid a : [B] \equiv L_1 \bowtie c : [!\, l(C).B'] \tag{A.1.8.4}$$

and there is L_1' such that:

$$L_1 \equiv L_1' \bowtie a : C \tag{A.1.8.5}$$

and:

$$Q' :: L_1' \bowtie c : [B'] \tag{A.1.8.6}$$

We rewrite (A.1.8.4) considering (A.1.8.5):

$$L' \mid a : [B] \equiv (L'_1 \bowtie a : C) \bowtie c : [!\, l(C).B'] \tag{A.1.8.7}$$

From (A.1.8.7) we conclude there is L''_1, C' such that $L'_1 \equiv L''_1 \mid a : C'$ and:

$$a : [B] \equiv a : C' \bowtie a : C \tag{A.1.8.8}$$

and:

$$L' \equiv L''_1 \bowtie c : [!\, l(C).B'] \tag{A.1.8.9}$$

From $L' <: L$ and (A.1.8.9), considering Lemma A.1.5) and Lemma A.1.4 we conclude there is L'', B_1, B'_1 such that:

$$L \equiv L'' \bowtie c : [B_1 \mid\, !\, l(C).B'_1] \tag{A.1.8.10}$$

where:

$$L''_1 <: L'' \qquad \text{and} \qquad B' <: B_1 \mid B'_1 \tag{A.1.8.11}$$

We rewrite (A.1.8.6), considering $L'_1 \equiv L''_1 \mid a : C'$:

$$Q' :: (L''_1 \mid a : C') \bowtie c : [B'] \tag{A.1.8.12}$$

Since $a \neq c$ we then have:

$$Q' :: (L''_1 \bowtie c : [B']) \mid a : C' \tag{A.1.8.13}$$

From (A.1.8.13) and (A.1.8.11) we conclude:

$$Q' :: (L'' \bowtie c : [B_1 \mid B'_1]) \mid a : C' \tag{A.1.8.14}$$

which completes the proof for this case. ■

Lemma A.1.9
Let types L_1, L'_1, L'_2 be such that $L'_1 <: L_1$ and $L'_1 \to L'_2$. Then we have that there is type L_2 such that L_1 and $P :: L'_2$ implies $P :: L_2$.

Proof. By induction on the derivation of $L'_1 <: L_1$, following expected lines. ■

Theorem 3.3.5 (Subject Reduction)
(repetition of the statement in page 58)
Let P be a well-typed process such that $P :: L$. If $P \to Q$ then there is $L \to L'$ such that $Q :: L'$.

Proof. By induction on the length of the derivation of the reduction $P \to Q$. We show the case of a reduction derived through rule (*Comm*) and from rule (*Close*), distinguishing between when the message is defined on a plain label and on a shared label.

(*Case* (*Comm*))

We have:

$$P_1 \mid P_2 \xrightarrow{\tau} Q_1 \mid Q_2 \tag{A.1.10.1}$$

derived from:

$$P_1 \xrightarrow{c \cdot l!(a)} Q_1 \;\; (i) \qquad \text{and} \qquad P_2 \xrightarrow{c \cdot l?(a)} Q_2 \;\; (ii) \tag{A.1.10.2}$$

Since $P_1 \mid P_2$ is a well-typed process, we have $P_1 \mid P_2 :: L$ for some L, L' such that $L' <: L$ and:

$$P_1 \mid P_2 :: L' \tag{A.1.10.3}$$

where (A.1.10.3) is derived from (rule (Par)):

$$P_1 :: L_1 \;\; (i) \qquad \text{and} \qquad P_2 :: L_2 \;\; (ii) \qquad \text{and} \qquad L' = L_1 \bowtie L_2 \;\; (iii) \tag{A.1.10.4}$$

From (A.1.10.2)(i) and (A.1.10.4)(i) and considering Lemma A.1.7 we conclude that there are L_1', C_1, B_1 such that:

$$L_1 \equiv L_1' \bowtie c : [!\, l(C_1).B_1] \tag{A.1.10.5}$$

From (A.1.10.2)(ii) and (A.1.10.4)(ii), considering Lemma A.1.6, we conclude that there are L_2', C_2, B_2 such that:

$$L_2 \equiv L_2' \bowtie c : [?\, l(C_2).B_2] \tag{A.1.10.6}$$

We consider the two possible cases: either the label is plain ($l \in \mathcal{L}_p$) or it is shared ($l \in \mathcal{L}_\star$).

(*Plain label*) If l is a plain label, from (A.1.10.4)(iii) we have that it must be the case that in L' there is a τ introduced by rule (*Plain*) for this synchronization which is only possible if $C_1 \equiv C_2$ and thus:

$$L' \equiv (L_1' \bowtie c : [!\, l(C).B_1]) \bowtie (L_2' \bowtie c : [?\, l(C).B_2]) \equiv (L_1' \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1 \bowtie B_2)]) \tag{A.1.10.7}$$

otherwise the merge $L_1 \bowtie L_2$ (A.1.10.4)(iii) would not be defined. We use C such that $C \equiv C_1 \equiv C_2$. We also have that, from Lemma A.1.7, there is L_1'' such that:

$$L_1' = L_1'' \bowtie a : C \;\; (i) \qquad \text{and} \qquad Q_1 :: L_1'' \bowtie c : [B_1] \;\; (ii) \tag{A.1.10.8}$$

From the merge in (A.1.10.7) and (A.1.10.8)(i) we conclude:

$$L' \equiv ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1 \bowtie B_2)]) \tag{A.1.10.9}$$

From (A.1.10.9) we have that there is L_2'' such that: $L_2'' = (L_2' \bowtie c : [B_2]) \bowtie a : C$, hence from Lemma A.1.6 we conclude:

$$Q_2 :: (L_2' \bowtie c : [B_2]) \bowtie a : C \tag{A.1.10.10}$$

From (A.1.10.9) and (A.1.10.8)(ii) and (A.1.10.10) we conclude:

$$Q_1 \mid Q_2 :: ((L_1'' \bowtie c : [B_1]) \bowtie ((L_2' \bowtie c : [B_2]) \bowtie a : C)) \tag{A.1.10.11}$$

We can derive a reduction for type L' as follows:

$$\begin{aligned} L' \;\; \equiv \;\; & ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1 \bowtie B_2)]) \\ & \rightarrow ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [B_1 \bowtie B_2]) \\ & \equiv ((L_1'' \bowtie c : [B_1]) \bowtie ((L_2' \bowtie c : [B_2]) \bowtie a : C)) \end{aligned}$$

Since $L' <: L$ and considering Lemma A.1.9 we have that there is L'' such that $L \to L''$ and $Q_1 \mid Q_2 :: T''$ which completes the proof for this case.

(*Shared label*) If l is a shared label then by conformance we have that $C_1 \equiv C_2 (\equiv C)$. From the definition of merge and (A.1.10.4)(iii) we conclude that $?\, l(C).B_2 \equiv \star ?\, l(C)$. Also, considering Lemma A.1.7 we have there is L_1'' such that:

$$L_1' = L_1'' \bowtie a : C \;\; (i) \qquad \text{and} \qquad Q_1 :: L_1'' \bowtie c : [B_1] \;\; (ii) \tag{A.1.10.12}$$

From the merge in (A.1.10.4)(iii), and (A.1.10.12)(i) and (A.1.10.6) we conclude:

$$L' = ((L_1'' \bowtie a : C) \bowtie c : [!\, l(C).B_1]) \bowtie (L_2' \bowtie c : [\star ?\, l(C)]) \tag{A.1.10.13}$$

for which we have:

$$\begin{array}{l} ((L_1'' \bowtie a : C) \bowtie c : [!\, l(C).B_1]) \bowtie (L_2' \bowtie c : [\star ?\, l(C)]) \\ \equiv \\ ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [!\, l(C).B_1 \bowtie \star ?\, l(C)]) \\ \equiv \\ ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1\{!\ l(C)/\tau\ l(C)\}) \mid \star\, ?\, l(C)]) \end{array} \tag{A.1.10.14}$$

From (A.1.10.14) we conclude there is $(L_2' \bowtie c : [\star ?\, l(C)]) \bowtie a : C$, which, from Lemma A.1.6, gives us that:

$$Q_2 :: (L_2' \bowtie c : [\star ?\, l(C)]) \bowtie a : C \tag{A.1.10.15}$$

From (A.1.10.12)(ii), (A.1.10.15) and (A.1.10.14) we derive:

$$Q_1 \mid Q_2 :: (L_1'' \bowtie c : [B_1]) \bowtie ((L_2' \bowtie c : [\star ?\, l(C)]) \bowtie a : C) \tag{A.1.10.16}$$

From (A.1.10.14) we have:

$$\begin{array}{l} ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1\{!\ l(C)/\tau\ l(C)\}) \mid \star\, ?\, l(C)]) \\ \to \\ ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [(B_1\{!\ l(C)/\tau\ l(C)\}) \mid \star\, ?\, l(C)]) \\ \equiv \\ ((L_1'' \bowtie a : C) \bowtie L_2') \bowtie (c : [B_1 \bowtie \star ?\, l(C)]) \\ \equiv \\ (L_1'' \bowtie c : [B_1]) \bowtie ((L_2' \bowtie c : [\star ?\, l(C)]) \bowtie a : C) \end{array} \tag{A.1.10.17}$$

We then have, from $L' <: L$, (A.1.10.13), (A.1.10.14) and (A.1.10.17) and considering Lemma A.1.9 that there is L'' such that $L \to L''$ and $Q_1 \mid Q_2 :: L''$ which completes the proof for this case.

(*Case* (*Close*))

We have:

$$P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu} a)(Q_1 \mid Q_2) \tag{A.1.10.18}$$

derived from:

$$P_1 \xrightarrow{(\nu a) c \cdot l!(a)} Q_1 \;\; (i) \qquad \text{and} \qquad P_2 \xrightarrow{c \cdot l?(a)} Q_2 \;\; (ii) \tag{A.1.10.19}$$

Since $P_1 \mid P_2$ is a well-typed process, we have $P_1 \mid P_2 :: L$ for some L, L' such that $L' <: L$ and:

$$P_1 \mid P_2 :: L' \tag{A.1.10.20}$$

where (A.1.10.20) is derived from (rule (*Par*)):

$$P_1 :: L_1 \ (i) \qquad \text{and} \qquad P_2 :: L_2 \ (ii) \qquad \text{and} \qquad L' = L_1 \bowtie L_2 \ (iii) \tag{A.1.10.21}$$

From (A.1.10.19)(i) and (A.1.10.21)(i) and considering Lemma A.1.8 we conclude that there are L_1', C_1, B_1 such that:

$$L_1 \equiv L_1' \bowtie c : [!\, l(C_1).B_1] \tag{A.1.10.22}$$

From (A.1.10.19)(ii) and (A.1.10.21)(ii), considering Lemma A.1.6, we conclude that there are L_2', C_2, B_2 such that:

$$L_2 \equiv L_2' \bowtie c : [?\, l(C_2).B_2] \tag{A.1.10.23}$$

We show the case when the label is plain — the proof for the case the label is shared follows similar lines. From (A.1.10.21)(iii) we have that it must be the case that in L' there is a τ introduced by rule (*Plain*) for this synchronization which is only possible if $C_1 \equiv C_2$ and thus:

$$L' \equiv (L_1' \bowtie c : [!\, l(C).B_1]) \bowtie (L_2' \bowtie c : [?\, l(C).B_2]) \equiv (L_1' \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1 \bowtie B_2)]) \tag{A.1.10.24}$$

otherwise the merge $L_1 \bowtie L_2$ (A.1.10.21)(iii) would not be defined. We use C such that $C \equiv C_1 \equiv C_2$. We also have that, from Lemma A.1.8, there is B', C' such that $\mathit{closed}(B')$ and:

$$a : [B'] = a : C' \bowtie a : C \ (i) \qquad \text{and} \qquad Q_1 :: (L_1' \bowtie c : [B_1]) \mid a : C' \ (ii) \tag{A.1.10.25}$$

Since $a \notin \mathit{fn}(Q_2)$ we have that there is L_2'' such that: $L_2'' = (L_2' \bowtie c : [B_2]) \bowtie a : C$, hence from Lemma A.1.6 we conclude:

$$Q_2 :: (L_2' \bowtie c : [B_2]) \bowtie a : C \tag{A.1.10.26}$$

From (A.1.10.24) and (A.1.10.25)(ii) and (A.1.10.26) we conclude:

$$Q_1 \mid Q_2 :: ((L_1' \bowtie c : [B_1]) \mid a : C') \bowtie ((L_2' \bowtie c : [B_2]) \bowtie a : C)) \tag{A.1.10.27}$$

From (A.1.10.27) we conclude:

$$Q_1 \mid Q_2 :: ((L_1' \bowtie c : [B_1]) \bowtie (L_2' \bowtie c : [B_2])) \mid (a : C' \bowtie a : C) \tag{A.1.10.28}$$

which, along with $\mathit{closed}(B')$ and (A.1.10.25)(i), gives us:

$$(\boldsymbol{\nu} a)(Q_1 \mid Q_2) :: (L_1' \bowtie c : [B_1]) \bowtie (L_2' \bowtie c : [B_2]) \tag{A.1.10.29}$$

We can derive a reduction for type L' as follows:

$$\begin{array}{rcl} L' & \equiv & (L_1' \bowtie L_2') \bowtie (c : [\tau\, l(C).(B_1 \bowtie B_2)]) \\ & & \rightarrow (L_1' \bowtie L_2') \bowtie (c : [B_1 \bowtie B_2]) \\ & & \equiv (L_1' \bowtie c : [B_1]) \bowtie (L_2' \bowtie c : [B_2]) \end{array}$$

Since $L' <: L$ and considering Lemma A.1.9 we conclude there is L'' such that $L \to L''$ and $(\boldsymbol{\nu} a)(Q_1 \mid Q_2) :: L''$ which completes the proof. ∎

Proposition 3.3.9 (Error Freeness)

(repetition of the statement in page 59)

If P is a well-typed process then P is not an error process.

Proof. The result follows from the definition of merge $\bowtie$. A parallel composition is well typed if the types of the parallel branches can be merged. Since it is not possible to synchronize message types defined on plain labels with the same polarity (which is the case for competing messages) and such types are not apart # (the label sets are not disjoint) it is not possible to merge them. Hence the composition of processes that exhibit competing plain messages is not typable, hence error processes are not typable.

From the definition of error process we there are $\mathcal{C}, Q, R, c, l, \mathtt{flag}$ such that $l \in \mathcal{L}_p$ and $P = \mathcal{C}[Q \mid R]$ and:

$$\mathcal{C}[Q \mid c \cdot l -.\mathtt{flag}^{\downarrow}!()] \to \mathcal{C}[Q' \mid \mathtt{flag}^{\downarrow}!()]$$

and

$$\mathcal{C}[R \mid c \cdot l -.\mathtt{flag}^{\downarrow}!()] \to \mathcal{C}[R' \mid \mathtt{flag}^{\downarrow}!()]$$

Let us consider the case when $c \cdot l - = c \cdot l?(x)$. We then have that there is a such that either:

$$Q \stackrel{(\nu a) c \cdot l!(a)}{\longrightarrow} Q' \tag{A.1.11.1}$$

or:

$$Q \stackrel{c \cdot l!(a)}{\longrightarrow} Q' \tag{A.1.11.2}$$

and b such that either:

$$R \stackrel{(\nu b) c \cdot l!(b)}{\longrightarrow} R' \tag{A.1.11.3}$$

or:

$$R \stackrel{c \cdot l!(b)}{\longrightarrow} R' \tag{A.1.11.4}$$

We consider the case (A.1.11.1) and (A.1.11.4), and elide the proof for the remaining combinations as their proofs follow similar lines. Let us assume that there are L_1, L_2 such that:

$$Q :: L_1 \tag{A.1.11.5}$$

and:

$$R :: L_2 \tag{A.1.11.6}$$

From (A.1.11.1) and (A.1.11.5) and Lemma A.1.8 we conclude there is L_1', B_1, C_1 such that:

$$L_1 \equiv L_1' \bowtie c : [!\ l(C_1).B_1] \tag{A.1.11.7}$$

and from (A.1.11.5) and (A.1.11.6) and Lemma A.1.7 we conclude there is L_2', B_2, C_2 such that:

$$L_2 \equiv L_2' \bowtie c : [!\ l(C_2).B_2] \tag{A.1.11.8}$$

Then, by definition of merge, we conclude it is not possible to synchronize such message types

as they both have polarity $!$. Furthermore they are not apart as both types specify message l in conversation c. Thus there is no type $L_1 \bowtie L_2$ and therefore $Q \mid R$ is untypable. ■

A.2 Chapter 4

Proposition 4.4.2 (Closure Under Union)
(repetition of the statement in page 81)
Let $\mathcal{S}$ be a set of strong bisimulations. Then $\bigcup \mathcal{S}$ is a strong bisimulation.

Proof. Let us consider processes P, Q such that $(P, Q) \in \bigcup \mathcal{S}$. We intend to prove that any transition of process P is matched by a transition of process Q leading to states which are equated by relation $\bigcup \mathcal{S}$. We have there must be $\mathcal{R} \in \mathcal{S}$ such that $P\mathcal{R}Q$, and thus for any λ, P' such that $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \# fn(Q)$ there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P'\mathcal{R}Q'$. Thus $(P', Q') \in \bigcup \mathcal{S}$, which completes the proof. ■

Next we show some results auxiliary to the proof of transitivity of strong bisimilarity. The first, Lemma A.2.1, shows that given two bisimilar processes we can swap around names that occur free in one of the processes and do not occur free in the other process, and still obtain bisimilar processes; The second, Lemma A.2.2, shows that given two bisimilar processes we can place one of the processes in a parallel context that has no behavior, and still obtain bisimilar processes.

Lemma A.2.1
Let P, Q be processes such that $P \sim Q$. If we take σ such that it is a name permutation on $fn(Q) \setminus fn(P)$ and the identity substitution on $\Lambda \setminus (fn(Q) \setminus fn(P))$, then $P \sim \sigma(Q)$.

Proof. Follows by coinduction on the definition of strong bisimulation in expected lines. Notice that the set of names swapped around occurs free only in one of the processes and hence will never be observed in the transitions. The set of names is kept invariant by the substitution so as to prevent clashes with names introduced by bound transitions. The proof proceeds by witnessing relation:

$$\mathcal{R} \triangleq \{(P, \sigma(Q)) \mid P \sim Q \ \wedge \ fn(\sigma(Q)) = fn(Q) \ \wedge \ \forall R. fn(R) \# (fn(Q) \setminus fn(P)) \implies \sigma(R) = R\} \tag{A.2.1.1}$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation.

Let us take P, Q such that $(P, \sigma(Q)) \in \mathcal{R}$. We first match a transition of P by a transition of $\sigma(P)$. We take λ, P' such that $P \xrightarrow{\lambda} P'$ and $bn(\lambda) \# fn(\sigma(Q))$. From $fn(\sigma(Q)) = fn(Q)$ (A.2.1.1) we then have $bn(\lambda) \# fn(Q)$. Then from $P \sim Q$ (A.2.1.1) we have there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $P' \sim Q'$. We then have $\sigma(Q) \xrightarrow{\sigma(\lambda)} \sigma(Q')$. From $bn(\lambda) \# fn(Q)$ and $fn(\lambda) \subseteq fn(P)$ and (A.2.1.1) we have $\sigma(P) = P$ and $\sigma(\lambda) = \lambda$. Thus $\sigma(Q) \xrightarrow{\lambda} \sigma(Q')$, which completes this part of the proof.

Now let us take λ, Q' such that $\sigma(Q) \xrightarrow{\lambda} Q'$ and $bn(\lambda) \# fn(P)$. We have $Q \xrightarrow{\sigma^{-1}(\lambda)} \sigma^{-1}(Q')$. From $bn(\lambda) \# fn(P)$ we conclude $bn(\sigma^{-1}(\lambda)) \# fn(P)$ (from $\sigma(P) = P$). Then from $P \sim Q$ we have there is P' such that $P \xrightarrow{\sigma^{-1}(\lambda)} P'$ and $P' \sim \sigma^{-1}(Q')$. We have $\sigma(\sigma^{-1}(\lambda)) = \lambda$. Thus from $P \xrightarrow{\sigma^{-1}(\lambda)} P'$ we have $\sigma(P) \xrightarrow{\sigma(\sigma^{-1}(\lambda))} \sigma(P')$ and hence $P \xrightarrow{\lambda} \sigma(P')$. From $\sigma(Q) \xrightarrow{\lambda} Q'$ we have that $bn(\lambda) \# fn(\sigma(Q))$, hence $bn(\lambda) \# fn(Q)$. We have that $fn(P') \subseteq fn(P) \cup bn(\lambda)$, from which

we then conclude $\mathit{fn}(P') \# (\mathit{fn}(Q) \setminus \mathit{fn}(P))$ and hence $\sigma(P') = P'$. So we have $P \xrightarrow{\lambda} P'$ and $P' \sim \sigma^{-1}(Q')$ and therefore $(P', Q') \in \mathcal{R}$ which completes the proof. ∎

Lemma A.2.2

Let P and Q be processes such that $P \sim Q$. Then $P \sim Q \mid (\boldsymbol{\nu} c)(c \blacktriangleleft [l^{\downarrow}!(\tilde{a})])$.

Proof. Follows by coinduction on the definition of strong bisimulation in expected lines. Notice process $(\boldsymbol{\nu} c)(c \blacktriangleleft [l^{\downarrow}!(\tilde{a})])$ has no observable behavior due to the name restriction on c. ∎

Proposition 4.4.4 (Equivalence Relation)

(repetition of the statement in page 81)

Strong bisimilarity is an equivalence relation.

Proof. Proof of reflexivity is immediate and of symmetry follows directly from the definition. Proof of transitivity follows by coinduction on the definition of strong bisimulation, by witnessing relation:

$$\mathcal{R} \triangleq \{(P, Q) \mid \exists R. P \sim R \wedge R \sim Q\}$$

is a bisimulation consistent relation, i.e., $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$ where $\mathcal{F}$ is the bisimulation function, and therefore contained in $\sim$, since $\sim$ is the union of all $\mathcal{F}$-consistent relations.

Let us consider processes P, Q such that $(P, Q) \in \mathcal{R}$. We must show that any transition of P is matched by a transition of Q leading to states related in $\mathcal{R}$. Let us consider λ, P' such that:

$$P \xrightarrow{\lambda} P' \;\; \mathit{and} \;\; \mathit{bn}(\lambda) \# \mathit{fn}(Q) \tag{A.2.3.1}$$

From $(P, Q) \in \mathcal{R}$ we have by definition of $\mathcal{R}$ that there is R such that $P \sim R$ and $R \sim Q$. We consider two distinct cases: either $\mathit{bn}(\lambda) \cap \mathit{fn}(R) = \emptyset$ or $\mathit{bn}(\lambda) \cap \mathit{fn}(R) \neq \emptyset$. If $\mathit{bn}(\lambda) \cap \mathit{fn}(R) = \emptyset$ then there is R' such that $R \xrightarrow{\lambda} R'$ and $P' \sim R'$. Since $R \sim Q$, from $R \xrightarrow{\lambda} R'$ and $\mathit{bn}(\lambda) \cap \mathit{fn}(Q) = \emptyset$ (A.2.3.1) we have that there is Q' such that $Q \xrightarrow{\lambda} Q'$ and $R' \sim Q'$. From $P' \sim R'$ and $R' \sim Q'$ we conclude $(P', Q') \in \mathcal{R}$ and thus complete the proof for this case.

If $\mathit{bn}(\lambda) \cap \mathit{fn}(R) \neq \emptyset$ we consider names $\tilde{a}$ such that $\mathit{bn}(\lambda) \cap \tilde{a} = \emptyset$ and a name substitution σ which is a bijection map between $\mathit{bn}(\lambda) \cap \mathit{fn}(R)$ and $\tilde{a}$ and the identity substitution otherwise. Considering Lemma A.2.2 from $P \sim R$ we then have:

$$P \sim R \mid (\boldsymbol{\nu} c)(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a})\right])$$

Then from Lemma A.2.1 we obtain:

$$P \sim \sigma(R \mid (\boldsymbol{\nu} c)(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a})\right]))$$

which gives us:

$$P \sim \sigma(R) \mid \sigma((\boldsymbol{\nu} c)(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a})\right]))$$

which again from Lemma A.2.2 gives us:

$$P \sim \sigma(R)$$

Likewise we show $\sigma(R) \sim Q$. Since $\mathit{bn}(\lambda) \cap \mathit{fn}(\sigma(R)) = \emptyset$ we can proceed as in the first case. ∎

Theorem 4.4.5 (Preservation of Strong Bisimilarity Under Structural Laws)
(repetition of the statement in page 81)
Given processes P, Q and R, the following axioms hold:

1. $(\boldsymbol{\nu} a)\mathbf{0} \sim \mathbf{0}$.
2. $(\boldsymbol{\nu} a)(\boldsymbol{\nu} b)P \sim (\boldsymbol{\nu} b)(\boldsymbol{\nu} a)P$.
3. *If* $a \notin fn(P)$ *then* $P \mid (\boldsymbol{\nu} a)Q \sim (\boldsymbol{\nu} a)(P \mid Q)$.
4. $P \mid \mathbf{0} \sim P$.
5. $P \mid (Q \mid R) \sim (P \mid Q) \mid R$.
6. $P \mid Q \sim Q \mid P$.
7. *If* $i \in I$ *if and only if* $i \in J$ *then* $\Sigma_{i \in I}\, \alpha_i.P_i \sim \Sigma_{i \in J}\, \alpha_i.P_i$.
8. $\textsf{rec}\ \mathcal{X}.P \sim P\{\mathcal{X}/\textsf{rec}\ \mathcal{X}.P\}$.

Proof. By coinduction on the definition of strong bisimulation.

1. Immediate as no transition can be observed on either process.
2. Follows expected lines.
3. We witness $\mathcal{R}$ defined as:

$$\mathcal{R} \triangleq \{((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q), (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q)) \mid \forall \tilde{a}, \tilde{c}, b.\ b \notin fn(P) \cup bn(P)\} \quad \text{(A.2.4.1)}$$

is such that $\mathcal{R} \cup \sim\ \subseteq\ \sim$ by coinduction on the definition of bisimulation. N.B. We impose $b \notin bn(P)$ to prevent bound name clashes, and then generalize the result through α-equivalence. Let us consider $((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q), (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q)) \in \mathcal{R}$ (for $(P, Q) \in\ \sim$ the proof is direct). We show that for every R_1, λ such that:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q) \xrightarrow{\lambda} R_1 \quad \text{(A.2.4.2)}$$

and $bn(\lambda) \,\#\, fn((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q))$ then there exists R_2 such that:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q) \xrightarrow{\lambda} R_2 \quad \text{(A.2.4.3)}$$

and $(R_1, R_2) \in \mathcal{R} \cup \sim$. We omit the symmetric case which follows similar lines.

We consider the three possible distinct cases for deriving (A.2.4.2): either the derivation results from an observation on P or on Q or from a synchronization between P and Q.

(*Observation on* P)

We have:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q) \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P' \mid (\boldsymbol{\nu} b)Q) \quad \text{(A.2.4.4)}$$

derived from:

$$P \xrightarrow{\lambda'} P' \quad \text{(A.2.4.5)}$$

where $\lambda = (\nu\tilde{d})\lambda'$ and $\tilde{d} = (\tilde{a} \cup \tilde{c}) \cap out(\lambda')$. We have that $bn(\lambda') \# fn((\boldsymbol{\nu} b)Q)$. Then since $b \notin fn(P) \cup bn(P)$ (A.2.4.1) we conclude $b \notin na(\lambda')$ and also $bn(\lambda') \# fn(Q)$. From (A.2.4.5) we then derive:

$$P \mid Q \xrightarrow{\lambda'} P' \mid Q \tag{A.2.4.6}$$

and thus:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q) \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P' \mid Q) \tag{A.2.4.7}$$

which completes the proof for this case since:

$$((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P' \mid (\boldsymbol{\nu} b)Q), (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P' \mid Q)) \in \mathcal{R} \tag{A.2.4.8}$$

(*Observation on* Q)

Follows similar lines.

(*Synchronization between* P *and* Q)

We show only the case of rule (*Close-l*) and (*Close-r*) (Figure 4.2), as the proofs of the remaining cases follow similar lines.

(*Case* (*Close-l*))

We have:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q) \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(\boldsymbol{\nu}\tilde{d})(P' \mid (\boldsymbol{\nu} b)Q') \tag{A.2.4.9}$$

which, eliding the cases for the restriction, is derived from:

$$P \xrightarrow{(\nu\tilde{d})\lambda_1} P' \tag{A.2.4.10}$$

and:

$$Q \xrightarrow{\lambda_2} Q' \tag{A.2.4.11}$$

and $\lambda_1 \bullet \lambda_2 \neq \circ$. From (A.2.4.10) and (A.2.4.11) we derive:

$$P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{d})(P' \mid Q') \tag{A.2.4.12}$$

which then gives us:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(P \mid Q) \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(\boldsymbol{\nu}\tilde{d})(P' \mid Q') \tag{A.2.4.13}$$

which completes the proof for this case since:

$$((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(\boldsymbol{\nu}\tilde{d})(P' \mid (\boldsymbol{\nu} b)Q'), (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu}\tilde{c})(\boldsymbol{\nu}\tilde{d})(P' \mid Q')) \in \mathcal{R} \tag{A.2.4.14}$$

(*Case* (*Close-r*))

We focus on the interesting case when Q extrudes name b to P. We have:

$$(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(P \mid (\boldsymbol{\nu} b)Q) \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{c})(\boldsymbol{\nu}\tilde{d}, b)(P' \mid Q') \tag{A.2.4.15}$$

derived from:

$$P \xrightarrow{\lambda_1} P' \tag{A.2.4.16}$$

and:

$$Q \xrightarrow{(\nu \tilde{d})\lambda_2} Q' \tag{A.2.4.17}$$

and $b \in out(\lambda)$, $\lambda_1 \bullet \lambda_2 \neq \circ$. From (A.2.4.16) and (A.2.4.17) we derive:

$$P \mid Q \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu} \tilde{d})(P' \mid Q') \tag{A.2.4.18}$$

which then gives us:

$$(\boldsymbol{\nu} \tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu} \tilde{c})(P \mid Q) \xrightarrow{\lambda_1 \bullet \lambda_2} (\boldsymbol{\nu} \tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu} \tilde{c})(\boldsymbol{\nu} \tilde{d})(P' \mid Q') \tag{A.2.4.19}$$

From Theorem 4.4.5(2) we conclude:

$$((\boldsymbol{\nu} \tilde{a})(\boldsymbol{\nu} \tilde{c})(\boldsymbol{\nu} \tilde{d}, b)(P' \mid Q'), (\boldsymbol{\nu} \tilde{a})(\boldsymbol{\nu} b)(\boldsymbol{\nu} \tilde{c})(\boldsymbol{\nu} \tilde{d})(P' \mid Q')) \in \sim \tag{A.2.4.20}$$

which completes the proof for this case.

4. Immediate.

5. Follows lines similar to the proof of 3.

6. Follows expected lines.

7. Immediate from rule (*Sum*) (Figure 4.2).

8. Immediate from rule (*Rec*) (Figure 4.2). ■

Theorem 4.4.6 (Congruence)
(repetition of the statement in page 82)
Strong bisimilarity is a congruence.

1. *If* $P \sim Q$ *then* $l^d!(\tilde{n}).P \sim l^d!(\tilde{n}).Q$.

2. *If* $P\{\tilde{x}/\tilde{n}\} \sim Q\{\tilde{x}/\tilde{n}\}$ *for all* $\tilde{n}$ *then* $l^d?(\tilde{x}).P \sim l^d?(\tilde{x}).Q$.

3. *If* $P\{x/n\} \sim Q\{x/n\}$ *for all* n *then* $\textsf{this}(x).P \sim \textsf{this}(x).Q$.

4. *If* $\alpha_i.P_i \sim \alpha'_i.Q_i$, *for all* $i \in I$, *then* $\Sigma_{i \in I}\, \alpha_i.P_i \sim \Sigma_{i \in I}\, \alpha'_i.Q_i$.

5. *If* $P \sim Q$ *then* $n \blacktriangleleft [P] \sim n \blacktriangleleft [Q]$.

6. *If* $P \sim Q$ *then* $(\boldsymbol{\nu} a)P \sim (\boldsymbol{\nu} a)Q$.

7. *If* $P \sim Q$ *then* $P \mid R \sim Q \mid R$.

8. *If* $P\{\mathcal{X}/R\} \sim Q\{\mathcal{X}/R\}$, *for all* R, *then* $\textsf{rec}\, \mathcal{X}.P \sim \textsf{rec}\, \mathcal{X}.Q$.

Proof. By coinduction on the definition of strong bisimulation.

1. The proof proceeds by witnessing relation

$$\mathcal{R} \triangleq \{(l^d!(\tilde{n}).P, l^d!(\tilde{n}).Q) \mid P \sim Q\} \cup \sim$$

is contained in $\sim$ (i.e., $\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation. Notice that pairs of processes in $\mathcal{R}$ are either in the strong bisimilarity relation in which case the proof is direct, or are of the form $(l^d!(\tilde{n}).P, l^d!(\tilde{n}).Q)$, where $P \sim Q$, in which case there is only one possible observation over any of the two processes (the same one), leading to processes which are strong bisimilar and thus are in $\mathcal{R}$, by definition.

2. Follows lines similar to 1, by witness relation

$$\mathcal{R} \triangleq \{(l^d?(\tilde{x}).P, l^d?(\tilde{x}).Q) \mid \forall \tilde{n}.\ P\{\tilde{x}/\tilde{n}\} \sim Q\{\tilde{x}/\tilde{n}\}\} \cup \sim$$

is contained in $\sim$ (i.e., $\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation. Now observing the input on both processes will lead to processes $P\{\tilde{x}/\tilde{n}\}$ and $Q\{\tilde{x}/\tilde{n}\}$, for some $\tilde{n}$, which are strong bisimilar since we have for any $\tilde{n}$ in the definition of $\mathcal{R}$.

3. Follows lines similar to 2.

4. Follows lines similar to 1.

5. The proof proceeds by witnessing relation

$$\mathcal{R} \triangleq \{(n \blacktriangleleft [P], n \blacktriangleleft [Q]) \mid P \sim Q\}$$

is contained in $\sim$ (i.e., $\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation. Let us consider P, Q such that $(n \blacktriangleleft [P], n \blacktriangleleft [Q]) \in \mathcal{R}$. We show that if there is P', λ such that:

$$n \blacktriangleleft [P] \xrightarrow{\lambda} P' \qquad \text{(A.2.5.1)}$$

and $bn(\lambda) \# fn(n \blacktriangleleft [Q])$ then there is a matching transition by $n \blacktriangleleft [Q]$ leading to a process Q' such that $(P', Q') \in \mathcal{R}$. We have that (A.2.5.1) is derived using either (*Here*), (*Loc*), (*Through*), (*Tau*), (*ThisLoc*) or (*ThisHere*) (Figure 4.3) from:

$$P \xrightarrow{\lambda'} P'' \qquad \text{(A.2.5.2)}$$

for some λ', P'' such that $P' = n \blacktriangleleft [P'']$. We have $bn(\lambda') \subseteq bn(\lambda)$ and $fn(Q) \subseteq fn(n \blacktriangleleft [Q])$. Thus we have $bn(\lambda') \# fn(Q)$. We then conclude, from $P \sim Q$ (by definition of $\mathcal{R}$) and $(A.2.5.2)$, that there is Q' such that:

$$Q \xrightarrow{\lambda'} Q' \qquad \text{(A.2.5.3)}$$

and $P'' \sim Q'$. From (A.2.5.3) we derive (using the same rule already used to derive (A.2.5.1) from (A.2.5.2)) that:

$$n \blacktriangleleft [Q] \xrightarrow{\lambda} n \blacktriangleleft [Q']$$

which completes the proof as $(n \blacktriangleleft [P''], n \blacktriangleleft [Q']) \in \mathcal{R}$.

6. Follows lines similar to 5.

7. Follows lines similar to Theorem 4.4.5(3). The proof proceeds by showing relation

$$\mathcal{R} \triangleq \{((\boldsymbol{\nu}\tilde{a})(P \mid R), (\boldsymbol{\nu}\tilde{a})(Q \mid R)) \mid P \sim Q\}$$

   is contained in $\sim$ (i.e., $\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation.

8. Follows lines similar to 2. ∎

Theorem 4.4.7 (Behavioral Identities)
(repetition of the statement in page 83)
Given processes P and Q, the following axioms hold:

1. $n \blacktriangleleft [\mathbf{0}] \sim \mathbf{0}$.
2. *If* $a \neq c$ *then* $(\boldsymbol{\nu}a)(c \blacktriangleleft [P]) \sim c \blacktriangleleft [(\boldsymbol{\nu}a)P]$.
3. $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \sim n \blacktriangleleft [P \mid Q]$.
4. $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]] \sim n \blacktriangleleft [o \blacktriangleleft [P]]$.
5. $n \blacktriangleleft [l^{\uparrow}!(\tilde{n}).P] \sim l^{\downarrow}!(\tilde{n}).n \blacktriangleleft [P]$.
6. *If* $n \notin \tilde{x}$ *then* $n \blacktriangleleft [l^{\uparrow}?(\tilde{x}).P] \sim l^{\downarrow}?(\tilde{x}).n \blacktriangleleft [P]$.
7. $m \blacktriangleleft [n \blacktriangleleft [l^{\downarrow}!(\tilde{n}).P]] \sim n \blacktriangleleft [l^{\downarrow}!(\tilde{n}).m \blacktriangleleft [n \blacktriangleleft [P]]]$.
8. *If* $\{m, n\} \# \tilde{x}$ *then* $m \blacktriangleleft [n \blacktriangleleft [l^{\downarrow}?(\tilde{x}).P]] \sim n \blacktriangleleft [l^{\downarrow}?(\tilde{x}).m \blacktriangleleft [n \blacktriangleleft [P]]]$.

Proof. By coinduction on the definition of strong bisimulation.

1. Immediate.
2. Follows expected lines.
3. The proof proceeds by showing relation:

$$\mathcal{R} \triangleq \{((\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P] \mid n \blacktriangleleft [Q]), n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)]) \mid \forall n, \tilde{a}.\ n \notin \tilde{a}\} \tag{A.2.6.1}$$

   is contained in $\sim$ ($\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation. Let us consider $((\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P] \mid n \blacktriangleleft [Q]), n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)]) \in \mathcal{R}$. We show that if there is R_1, λ such that :

$$(\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P] \mid n \blacktriangleleft [Q]) \xrightarrow{\lambda} R_1 \tag{A.2.6.2}$$

   then there is a matching transition by $n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)]$ leading to a state R_2 such that $(R_1, R_2) \in \mathcal{R}$ (we omit the symmetric case as it follows similar lines). We consider the three possible cases for the derivation of (A.2.6.2): it either results from an observation over $n \blacktriangleleft [P]$, over $n \blacktriangleleft [Q]$, or from a synchronization between $n \blacktriangleleft [P]$ and $n \blacktriangleleft [Q]$. If the transition results from an observation over $n \blacktriangleleft [P]$ we have that $R_1 = (\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P'] \mid n \blacktriangleleft [Q])$. We can then show, in non-surprising lines, that:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{\lambda} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(P' \mid Q)\right]$$

as intended. The case when (A.2.6.2) is derived from an observation over $n \blacktriangleleft [Q]$ is analogous. If (A.2.6.2) is derived from a synchronization between $n \blacktriangleleft [P]$ and $n \blacktriangleleft [Q]$ then it is either derived through rule (*Comm*) or (*Close*) of Figure 4.2. We show the proof for (*Close*) — actually (*Close-l*) — as the proof for (*Comm*) follows similar lines. We have that (A.2.6.2) is derived from:

$$n \blacktriangleleft [P] \xrightarrow{(\nu\tilde{b})\lambda_1} n \blacktriangleleft \left[P'\right] \tag{A.2.6.3}$$

and:

$$n \blacktriangleleft [Q] \xrightarrow{\lambda_2} n \blacktriangleleft \left[Q'\right] \tag{A.2.6.4}$$

and where $\lambda = \lambda_1 \bullet \lambda_2$, $\lambda \neq \circ$ and $R_1 = (\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(n \blacktriangleleft [P'] \mid n \blacktriangleleft [Q'])$. We consider the possible cases for the derivation of (A.2.6.3) (and (A.2.6.4)): either λ_1 (or λ_2) is a located label in which case (A.2.6.3) (or (A.2.6.4)) is derived either through (*Loc*) or (*Through*); or λ_1 (or λ_2) is not located in which case (A.2.6.3) (or (A.2.6.4)) is derived from (*Here*). We show the proof for these cases, omitting symmetric cases.

(*(A.2.6.3) derived through* (*Loc*) *and (A.2.6.4) derived through* (*Loc*)) Since both λ_1 and λ_2 are located labels and $\lambda_1 \bullet \lambda_2 \neq \circ$ we conclude $\lambda_1 \bullet \lambda_2 = \tau$. We have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_3^{\downarrow}} P' \tag{A.2.6.5}$$

and $\lambda_1 = n \cdot \lambda_3^{\downarrow}$. We also have that:

$$Q \xrightarrow{\lambda_4^{\downarrow}} Q' \tag{A.2.6.6}$$

and $\lambda_2 = n \cdot \lambda_4^{\downarrow}$. Since $n \cdot \lambda_3^{\downarrow} \bullet n \cdot \lambda_4^{\downarrow} = \tau$ we conclude $\lambda_3^{\downarrow} \bullet \lambda_4^{\downarrow} = \tau$. Thus, from (A.2.6.5) and (A.2.6.6) we conclude:

$$P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{\tau} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof as:

$$((\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(n \blacktriangleleft \left[P'\right] \mid n \blacktriangleleft \left[Q'\right]), n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]) \in \mathcal{R}$$

(*(A.2.6.3) derived through* (*Loc*) *and (A.2.6.4) derived through* (*Through*)) Since both λ_1 and λ_2 are located labels and $\lambda_1 \bullet \lambda_2 \neq \circ$ we conclude $\lambda_1 \bullet \lambda_2 = \tau$. We have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_3^{\downarrow}} P' \tag{A.2.6.7}$$

and $\lambda_1 = n \cdot \lambda_3^{\downarrow}$. We also have that:

$$Q \xrightarrow{\lambda_2} Q' \tag{A.2.6.8}$$

and $loc(\lambda_2)$. Since $n \cdot \lambda_3^{\downarrow} \bullet \lambda_2 = \tau$ we conclude: $\lambda_3^{\downarrow} \bullet \lambda_2 = n\ \mathtt{this}^{\downarrow}$. Thus, from (A.2.6.7) and (A.2.6.8) we conclude:

$$P \mid Q \xrightarrow{n\ \mathtt{this}^{\downarrow}} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{\tau} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof.

((A.2.6.3) derived through (*Loc*) *and (A.2.6.4) derived through* (*Here*)) We have that λ_1 is a located label and λ_2 is an unlocated label and $\lambda_1 \bullet \lambda_2 \neq \circ$. We also have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_3^{\downarrow}} P' \tag{A.2.6.9}$$

where $\lambda_1 = n \cdot \lambda_3^{\downarrow}$, and:

$$Q \xrightarrow{\lambda_4^{\uparrow}} Q' \tag{A.2.6.10}$$

and $\lambda_2 = \lambda_4^{\downarrow}$. We then have that $\lambda_1 \bullet \lambda_2 = n\ \texttt{this}^{\downarrow}$ and hence $\lambda_3^{\downarrow} \bullet \lambda_4^{\uparrow} = \texttt{this}^{\downarrow\uparrow}$. Thus, from (A.2.6.9) and (A.2.6.10) we conclude:

$$P \mid Q \xrightarrow{\texttt{this}^{\downarrow\uparrow}} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{c\ \texttt{this}^{\downarrow}} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof for this case.

((A.2.6.3) derived through (*Through*) *and (A.2.6.4) derived through* (*Through*)) We have that λ_1 and λ_2 are located labels and $\lambda_1 \bullet \lambda_2 \neq \circ$, thus $\lambda_1 \bullet \lambda_2 = \tau$. We also have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_1} P' \tag{A.2.6.11}$$

where $loc((\boldsymbol{\nu}\tilde{b})\lambda_1)$, and:

$$Q \xrightarrow{\lambda_2} Q' \tag{A.2.6.12}$$

and $loc(\lambda_2)$. Thus, from (A.2.6.11) and (A.2.6.12) we conclude:

$$P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{\tau} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof for this case.

((A.2.6.3) derived through (*Through*) *and (A.2.6.4) derived through* (*Here*)) We have that λ_1 is a located label and λ_2 is an unlocated label and $\lambda_1 \bullet \lambda_2 \neq \circ$. We also have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_1} P' \tag{A.2.6.13}$$

where $loc((\boldsymbol{\nu}\tilde{b})\lambda_1)$, and:

$$Q \xrightarrow{\lambda_4^{\uparrow}} Q' \tag{A.2.6.14}$$

where $\lambda_2 = \lambda_4^{\downarrow}$. We then have that $\lambda_1 \bullet \lambda_2 = m\ \mathtt{this}^{\downarrow}$ and hence $\lambda_1 \bullet \lambda_4^{\updownarrow} = m\ \mathtt{this}^{\uparrow}$. Thus, from (A.2.6.13) and (A.2.6.14) we conclude:

$$P \mid Q \xrightarrow{m\ \mathtt{this}^{\uparrow}} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{m\ \mathtt{this}^{\downarrow}} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof for this case.

(*(A.2.6.3) derived through* (*Here*) *and (A.2.6.4) derived through* (*Here*)) We have that λ_1 and λ_2 are unlocated labels and $\lambda_1 \bullet \lambda_2 \neq \circ$ and thus $\lambda_1 \bullet \lambda_2 = \tau$. We also have that:

$$P \xrightarrow{(\nu\tilde{b})\lambda_3^{\updownarrow}} P' \tag{A.2.6.15}$$

where $\lambda_1 = \lambda_3^{\downarrow}$, and:

$$Q \xrightarrow{\lambda_4^{\updownarrow}} Q' \tag{A.2.6.16}$$

where $\lambda_2 = \lambda_4^{\downarrow}$. We then have that $\lambda_3^{\updownarrow} \bullet \lambda_4^{\updownarrow} = \tau$. Thus, from (A.2.6.15) and (A.2.6.16) we conclude:

$$P \mid Q \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{b})(P' \mid Q')$$

from which we have:

$$n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P \mid Q)] \xrightarrow{\tau} n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{a})(\boldsymbol{\nu}\tilde{b})(P' \mid Q')\right]$$

which completes the proof.

4. The proof proceeds by showing relation:

$$\mathcal{R} \triangleq \{(m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]]\,, n \blacktriangleleft [o \blacktriangleleft [P]])\}$$

is contained in $\sim$ ($\mathcal{R} \subseteq \sim$) by coinduction on the definition of strong bisimulation. Notice that observations over process P will get located at most after crossing two context barriers, and located labels transparently cross conversation construct boundaries, as is illustrated in the following derivation:

$$\dfrac{\dfrac{\dfrac{P \xrightarrow{\sigma^{\uparrow}} P'}{o \blacktriangleleft [P] \xrightarrow{\sigma^{\downarrow}} o \blacktriangleleft [P']}}{n \blacktriangleleft [o \blacktriangleleft [P]] \xrightarrow{n \cdot \sigma^{\downarrow}} n \blacktriangleleft [o \blacktriangleleft [P']]}}{m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]] \xrightarrow{n \cdot \sigma^{\downarrow}} m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P']]]}$$

Thus all observations over $n \blacktriangleleft [o \blacktriangleleft [P]]$ can be derived for $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]]$ through (*Through*), and any observation over $m \blacktriangleleft [n \blacktriangleleft [o \blacktriangleleft [P]]]$ is surely the result of applying rule (*Through*) and hence is an observation of $n \blacktriangleleft [o \blacktriangleleft [P]]$.

5. Immediate from the fact that:

$$n \blacktriangleleft \left[l^{\uparrow}!(\tilde{n}).P\right] \xrightarrow{l^{\downarrow}!(\tilde{n})} n \blacktriangleleft [P]$$

and:

$$l^{\downarrow}!(\tilde{n}).n \blacktriangleleft [P] \xrightarrow{l^{\downarrow}!(\tilde{n})} n \blacktriangleleft [P]$$

are the only possible observations over such processes.

6. Follows similar lines to the proof of 5.

7. Follows from 4 and the fact that:

$$m \blacktriangleleft \left[n \blacktriangleleft \left[l^{\downarrow}!(\tilde{n}).P\right]\right] \xrightarrow{n\ l^{\downarrow}!(\tilde{n})} m \blacktriangleleft [n \blacktriangleleft [P]]$$

and:

$$n \blacktriangleleft \left[l^{\downarrow}!(\tilde{n}).m \blacktriangleleft [n \blacktriangleleft [P]]\right] \xrightarrow{n\ l^{\downarrow}!(\tilde{n})} n \blacktriangleleft [m \blacktriangleleft [n \blacktriangleleft [P]]]$$

are the only possible observations over such processes.

8. Follows similar lines to the proof of 7. ∎

Proposition 4.4.14 (Normal Form)
(repetition of the statement in page 85)

If P is a well-formed process then there exists a process Q in normal form such that $P \sim Q$.

Proof. By induction on the number of active processes. Essentially, we isolate each active process (through Theorem 4.4.7(3)), reduce the nesting level (through Theorem 4.4.7(4))) and then place the active process in the appropriate conversation piece (again through Theorem 4.4.7(3)).

We sketch the proof for the general case, for which we implicitly use the congruence result (Theorem 4.4.6), and directly unfold recursive processes (Theorem 4.4.5(8)) and lift all restrictions to top-level, since it is standard to prove using scope extrusion (Theorem 4.4.5(3) and Theorem 4.4.5(2)) and α-conversion. Let us then consider that P is of the form:

$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U \mid P_1] \mid P_2] \ldots \mid P_3] \mid P') \qquad \text{(A.2.7.1)}$$

for some active process U, processes $P_1, P_2, \ldots, P_3, P'$ and names $\tilde{n}$. Considering Theorem 4.4.7(3) we conclude:

$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U \mid P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U] \mid c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$\vdots$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \ldots] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
(A.2.7.2)

We thus have:

$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U \mid P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \ldots] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P') \quad \text{(A.2.7.3)}$$

From (A.2.7.3) and considering Theorem 4.4.7(4) we conclude:

$$(\boldsymbol{\nu}\tilde{a})\ (c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \ldots] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_2 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \ldots] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$\vdots$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P') \quad \text{(A.2.7.4)}$$

By induction hypothesis we conclude that process $c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P'$ (which has one less active process: U) can be rewritten in normal form, hence:

$$c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P'$$
$$\sim \quad \text{(A.2.7.5)}$$
$$(\boldsymbol{\nu}\tilde{b})\ (U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]])$$

We then have:

$$(\boldsymbol{\nu}\tilde{a})\ (c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid c_1 \blacktriangleleft [\ldots c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [P_1] \mid P_2] \ldots \mid P_3] \mid P')$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a})\ (c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid (\boldsymbol{\nu}\tilde{b})\ (U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]])) \quad \text{(A.2.7.6)}$$

Let $\tilde{b} \,\#\, fn(c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]])$ (otherwise $\tilde{b}$ can be replaced by α-conversion). Considering Theorem 4.4.5(3) we then conclude:

$$(\boldsymbol{\nu}\tilde{a})\ (c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid (\boldsymbol{\nu}\tilde{b})\ (U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]]))$$
$$\sim$$
$$(\boldsymbol{\nu}\tilde{a}, \tilde{b})\ (c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]]) \quad \text{(A.2.7.7)}$$

We consider the two possible cases: either there is $j \in 1, \ldots, l$ such that $c_{k-1} \cdot c_k = e_j \cdot f_j$ or there is no such j. In the latter case the proof is complete. In the former, i.e., there is j such that $c_{k-1} = e_j$ and $c_k = f_j$, from the fact that parallel composition is commutative and associative

under bisimilarity (Theorem 4.4.5(5) and Theorem 4.4.5(6)) we have:

$$
\begin{aligned}
&(\boldsymbol{\nu}\tilde{a},\tilde{b})\,(c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \\
&\quad \mid U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]]) \\
&\sim \\
&(\boldsymbol{\nu}\tilde{a},\tilde{b})\,(U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \\
&\quad \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid e_j \blacktriangleleft [f_j \blacktriangleleft [V_j]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]])
\end{aligned}
\tag{A.2.7.8}
$$

and then considering Theorem 4.4.7(3) we conclude:

$$
\begin{aligned}
&(\boldsymbol{\nu}\tilde{a},\tilde{b})\,(U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \\
&\quad \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U]] \mid e_j \blacktriangleleft [f_j \blacktriangleleft [V_j]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]]) \\
&\sim \\
&(\boldsymbol{\nu}\tilde{a},\tilde{b})\,(U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \\
&\quad \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U] \mid f_j \blacktriangleleft [V_j]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]]) \\
&\sim \\
&(\boldsymbol{\nu}\tilde{a},\tilde{b})\,(U_0 \mid d_1 \blacktriangleleft [U_1] \mid \ldots \mid d_k \blacktriangleleft [U_k] \\
&\quad \mid e_1 \blacktriangleleft [f_1 \blacktriangleleft [V_1]] \mid \ldots \mid c_{k-1} \blacktriangleleft [c_k \blacktriangleleft [U \mid V_j]] \mid \ldots \mid e_l \blacktriangleleft [f_l \blacktriangleleft [V_l]])
\end{aligned}
\tag{A.2.7.9}
$$

which completes the proof. ∎

Proposition 4.4.18 (Structural Congruence Included in Bisimilarity)
(repetition of the statement in page 87)
We have that $\equiv \subseteq \sim$.

Proof. The proof proceeds by witnessing relation

$$\mathcal{R} \triangleq \{(P, Q) \mid P \equiv Q\} \cup \sim \tag{A.2.8.1}$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation. For each transition of P we derive a matching transition of Q such that the arrival states are in $\mathcal{R}$. We do so by induction on the length of the derivation of $P \equiv Q$. We analyze all structural congruence axioms individually, and since for each one there is a corresponding axiom for bisimilarity (Theorem 4.4.5 and Theorem 4.4.7) we immediately obtain that any transition of P is matched by a transition of Q leading to bisimilar states. Since $\sim \subseteq \mathcal{R}$, by definition (A.2.8.1), the proof is complete.
Next we show some auxiliary results to the proof of Theorem 4.4.21.

Lemma A.2.9
Let P be a process. If $P \equiv Q$ and $Q \xrightarrow{\lambda} Q'$ then there is P' such that $P \xrightarrow{\lambda} P'$ and $P' \equiv Q'$.

Proof. By induction on the length of the derivation of $P \equiv Q$.

(*Case* (*StructAlpha*))
Immediate since we identify α-equivalent processes (Convention 4.2.11).
(*Case* (*StructParZero*))
We have $P \mid \mathbf{0} \equiv P$ and $P \xrightarrow{\lambda} P'$. We conclude $P \mid \mathbf{0} \xrightarrow{\lambda} P' \mid \mathbf{0}$ and $P' \mid \mathbf{0} \equiv P'$.
(*Case* (*StructParComm*))
We have $P \mid Q \equiv Q \mid P$ and $Q \mid P \xrightarrow{\lambda} R$. The transition is either derived from (*Par*),

(*Close*) or (*Comm*). In the case of (*Par-l*) we have $Q \mid P \xrightarrow{\lambda} Q' \mid P$ derived from $Q \xrightarrow{\lambda} Q'$. Hence we conclude (by (*Par-r*)) $P \mid Q \xrightarrow{\lambda} P \mid Q'$ and $P \mid Q' \equiv Q' \mid P$ which completes the proof for this case. In the case of (*Comm*) we have $Q \mid P \xrightarrow{\lambda} Q' \mid P'$ derived from $Q \xrightarrow{\lambda_1} Q'$ and $P \xrightarrow{\lambda_2} P'$ and $\lambda = \lambda_1 \bullet \lambda_2$. Since $\bullet$ is commutative we have $\lambda = \lambda_2 \bullet \lambda_1$ and hence $P \mid Q \xrightarrow{\lambda} P' \mid Q'$ and $P' \mid Q' \equiv Q' \mid P'$ which completes the proof for this case. Case (*Close*) follows similar lines.

(*Case* (*StructParAssoc*))

Follows expected lines.

(*Case* (*StructResZero*))

Not applicable since $(\boldsymbol{\nu} a)\mathbf{0}$ and $\mathbf{0}$ do not exhibit transitions.

(*Case* (*StructResComm*))

Follows expected lines.

(*Case* (*StructResPar*))

We have $P \mid (\boldsymbol{\nu} a)Q \equiv (\boldsymbol{\nu} a)(P \mid Q)$, where $a \notin fn(P)$. We consider the interesting case of $P \mid (\boldsymbol{\nu} a)Q \xrightarrow{\lambda} R$ when the transition is derived through (*Close-r*). We have that $P \mid (\boldsymbol{\nu} a)Q \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{c})(P' \mid Q')$ derived from $P \xrightarrow{\lambda_1} P'$ and $(\boldsymbol{\nu} a)Q \xrightarrow{(\nu\tilde{c})\lambda_2} Q'$ and $\lambda = \lambda_1 \bullet \lambda_2$. We have two distinct cases depending if name a is in $\tilde{c}$ or not. If $a \notin \tilde{c}$ then we have Q' is of the form $(\boldsymbol{\nu} a)Q''$ and $Q \xrightarrow{(\nu\tilde{c})\lambda_2} Q''$. We then have $P \mid Q \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{c})(P' \mid Q'')$ and thus $(\boldsymbol{\nu} a)(P \mid Q) \xrightarrow{\lambda} (\boldsymbol{\nu} a)(\boldsymbol{\nu}\tilde{c})(P' \mid Q'')$, which concludes the proof for this case as $(\boldsymbol{\nu} a)(\boldsymbol{\nu}\tilde{c})(P' \mid Q'') \equiv (\boldsymbol{\nu}\tilde{c})(P' \mid (\boldsymbol{\nu} a)Q'')$. If $a \in \tilde{c}$ we have that Q' is of the form Q'' such that $Q \xrightarrow{(\nu\tilde{c}')\lambda_2} Q''$, where $\tilde{c} = a, \tilde{c}'$. We then have $P \mid Q \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{c}')(P' \mid Q'')$ and thus $(\boldsymbol{\nu} a)(P \mid Q) \xrightarrow{\lambda} (\boldsymbol{\nu} a)(\boldsymbol{\nu}\tilde{c}')(P' \mid Q'')$ which completes the proof for this case as $(\boldsymbol{\nu} a)(\boldsymbol{\nu}\tilde{c}')(P' \mid Q'') \equiv (\boldsymbol{\nu}\tilde{c})(P' \mid Q'')$.

(*Case* (*StructRec*) *and* (*StructSum*))

Follow expected lines.

(*Case* (*StructCZero*))

Not applicable since $n \blacktriangleleft [\mathbf{0}]$ and $\mathbf{0}$ do not exhibit transitions.

(*Case* (*StructCRes*))

Follows expected lines.

(*Case* (*StructCSplit*))

We have $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \equiv n \blacktriangleleft [P \mid Q]$. We consider the interesting case when $n \blacktriangleleft [P] \mid n \blacktriangleleft [Q] \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P'] \mid n \blacktriangleleft [Q'])$ derived from $n \blacktriangleleft [P] \xrightarrow{(\nu\tilde{a})\lambda_1} n \blacktriangleleft [P']$ and $n \blacktriangleleft [Q] \xrightarrow{\lambda_2} n \blacktriangleleft [Q']$ and $\lambda = \lambda_1 \bullet \lambda_2$. Let us consider $n \blacktriangleleft [P] \xrightarrow{(\nu\tilde{a})\lambda_1} n \blacktriangleleft [P']$ is derived from $P \xrightarrow{(\nu\tilde{a})\lambda_1} P'$ and $n \blacktriangleleft [Q] \xrightarrow{\lambda_2} n \blacktriangleleft [Q']$ is derived from $Q \xrightarrow{\lambda_2} Q'$. We then conclude $P \mid Q \xrightarrow{\lambda} (\boldsymbol{\nu}\tilde{a})(P' \mid Q')$ and $n \blacktriangleleft [P \mid Q] \xrightarrow{\lambda} n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P' \mid Q')]$ which completes the proof for this case as $(\boldsymbol{\nu}\tilde{a})(n \blacktriangleleft [P'] \mid n \blacktriangleleft [Q']) \equiv n \blacktriangleleft [(\boldsymbol{\nu}\tilde{a})(P' \mid Q')]$.

(*Case* (*StructCNest*))

Follows expected lines.

(*Cases* (*StructOutUp*) *and* (*StructInUp*))

(*StructOutUp*) is direct from the fact that any transition of $n \blacktriangleleft \left[\Sigma_{i\in I}\, l_i^{\uparrow}!(\tilde{m}).P_i\right]$ is matched by a transition of $\Sigma_{i\in I}\, l_i^{\downarrow}!(\tilde{m}).n \blacktriangleleft [P_i]$ and the arrival states are the same exact process. For instance, $n \blacktriangleleft \left[\Sigma_{i\in I}\, l_i^{\uparrow}!(\tilde{m}).P_i\right] \xrightarrow{\lambda} n \blacktriangleleft [P_j]$ and $\Sigma_{i\in I}\, l_i^{\downarrow}!(\tilde{m}).n \blacktriangleleft [P_i] \xrightarrow{\lambda} n \blacktriangleleft [P_j]$, for $\lambda = l_j^{\downarrow}!(m_j)$. Similarly for case (*StructInUp*).

(*Cases* (*StructOutHere*) *and* (*StructInHere*))

Follows similar lines to the previous case. ∎

Lemma A.2.10
Let P be a process. If $P \xrightarrow{l^d!(\tilde{a})} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{c})(l^d!(\tilde{a}).Q + S \mid R)$, where $\tilde{a} \# \tilde{c}$, and $P' \equiv (\boldsymbol{\nu}\tilde{c})(Q \mid R)$.

Proof. By induction on the derivation of $P \xrightarrow{l^d!(\tilde{a})} P'$.

(*Cases* (*Out*) *and* (*Sum*))

Immediate.

(*Case* (*Res*))

We have that $(\boldsymbol{\nu}b)P \xrightarrow{l^d!(\tilde{a})} (\boldsymbol{\nu}b)P'$ derived from $P \xrightarrow{l^d!(\tilde{a})} P'$ and $b \notin na(l^d!(\tilde{a}))$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{c})(l^d!(\tilde{a}).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(Q \mid R)$. We then have $(\boldsymbol{\nu}b)P \equiv (\boldsymbol{\nu}b)(\boldsymbol{\nu}\tilde{c})(l^d!(\tilde{a}).Q + S \mid R)$ where $\tilde{a} \# \tilde{c}, b$, and $(\boldsymbol{\nu}b)P' \equiv (\boldsymbol{\nu}b)(\boldsymbol{\nu}\tilde{c})(Q \mid R)$, which completes the proof for this case.

(*Case* (*Par-l*))

We have that $P_1 \mid P_2 \xrightarrow{l^d!(\tilde{a})} P_1' \mid P_2$ derived from $P_1 \xrightarrow{l^d!(\tilde{a})} P_1'$. By induction hypothesis we obtain $P_1 \equiv (\boldsymbol{\nu}\tilde{c})(l^d!(\tilde{a}).Q + S \mid R)$ where $\tilde{a} \# \tilde{c}$, and $P_1' \equiv (\boldsymbol{\nu}\tilde{c})(Q \mid R)$. We then have $P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c}')(l^d!(\tilde{a}).Q' + S' \mid R' \mid P_2)$, where $(\boldsymbol{\nu}\tilde{c})(l^d!(\tilde{a}).Q + S \mid R) \equiv_\alpha (\boldsymbol{\nu}\tilde{c}')(l^d!(\tilde{a}).Q' + S' \mid R')$ and $\tilde{c}' \# fn(P_2)$ and $\tilde{a} \# \tilde{c}'$. We also have $P_1' \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c}')(Q' \mid R' \mid P_2)$, which completes the proof.

(*Case* (*Rec*))

Follows immediately from induction hypothesis.

(*Case* (*Here*))

We have that $n \blacktriangleleft [P] \xrightarrow{l^\downarrow!(\tilde{a})} n \blacktriangleleft [P']$ derived from $P \xrightarrow{l^\uparrow!(\tilde{a})} P'$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{c})(l^\uparrow!(\tilde{a}).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(Q \mid R)$. We conclude (from rules (*StructCRes*) and (*StructCSplit*)) that $n \blacktriangleleft [P] \equiv n \blacktriangleleft [(\boldsymbol{\nu}\tilde{c})(l^\uparrow!(\tilde{a}).Q + S \mid R)] \equiv (\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft [l^\uparrow!(\tilde{a}).Q + S] \mid n \blacktriangleleft [R])$. Then (from (*StructOutUp*)) we conclude $(\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft [l^\uparrow!(\tilde{a}).Q + S] \mid n \blacktriangleleft [R]) \equiv (\boldsymbol{\nu}\tilde{c})(l^\downarrow!(\tilde{a}).n \blacktriangleleft [Q] + S' \mid n \blacktriangleleft [R])$. We also have that $n \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft [Q] \mid n \blacktriangleleft [R])$, which completes the proof for this case. ∎

Lemma A.2.11
Let P be a process. If $P \xrightarrow{b\ l^\downarrow!(\tilde{a})} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)$, where $\tilde{a}, b \# \tilde{c}$.

Proof. By induction on the derivation of $P \xrightarrow{b\ l^\downarrow!(\tilde{a})} P'$.

(*Case* (*Res*))

We have that $(\boldsymbol{\nu}n)P \xrightarrow{b\ l^\downarrow!(\tilde{a})} (\boldsymbol{\nu}n)P'$ derived from $P \xrightarrow{b\ l^\downarrow!(\tilde{a})} P'$ and $n \notin na(b\ l^\downarrow!(\tilde{a}))$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)$. We then have $(\boldsymbol{\nu}n)P \equiv (\boldsymbol{\nu}n)(\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R)$ and $\tilde{a}, b \# \tilde{c}, n$, and $(\boldsymbol{\nu}n)P' \equiv (\boldsymbol{\nu}n)(\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)$ which completes the proof for this case.

(*Case* (*Par-l*))

We have that $P_1 \mid P_2 \xrightarrow{b\ l^\downarrow!(\tilde{a})} P_1' \mid P_2$ derived from $P_1 \xrightarrow{b\ l^\downarrow!(\tilde{a})} P_1'$. By induction hypothesis we obtain $P_1 \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R)$, where $\tilde{a}, b \# \tilde{c}$, and $P_1' \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)$. We then have $P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c}')(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q' + S'] \mid R' \mid P_2)$, where $(\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R) \equiv_\alpha$

$(\boldsymbol{\nu}\tilde{c}')(b \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q' + S'] \mid R')$, $\tilde{c}' \# fn(P_2)$ and $\tilde{a}, b \# \tilde{c}'$. We also have that $P_1' \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c}')(b \blacktriangleleft [Q'] \mid R' \mid P_2)$ which completes the proof.

(*Case* (*Rec*))

Follows immediately from induction hypothesis.

(*Case* (*Loc*))

We have that $b \blacktriangleleft [P] \stackrel{b\ l^{\downarrow}!(\tilde{a})}{\longrightarrow} b \blacktriangleleft [P']$ derived from $P \stackrel{l^{\downarrow}!(\tilde{a})}{\longrightarrow} P'$. Considering Lemma A.2.10 we obtain $P \equiv (\boldsymbol{\nu}\tilde{c})(l^{\downarrow}!(\tilde{a}).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(Q \mid R)$ (we consider $b \notin \tilde{c}$ otherwise we would α-rename $\tilde{c}$). Then, considering rules (*StructCRes*) and (*StructCSplit*) we conclude that $b \blacktriangleleft [P] \equiv b \blacktriangleleft [(\boldsymbol{\nu}\tilde{c})(l^{\downarrow}!(\tilde{a}).Q + S \mid R)] \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q + S] \mid b \blacktriangleleft [R])$. Likewise we conclude $b \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid b \blacktriangleleft [R])$ which completes the proof for this case.

(*Case* (*Through*))

We have that $n \blacktriangleleft [P] \stackrel{b\ l^{\downarrow}!(\tilde{a})}{\longrightarrow} n \blacktriangleleft [P']$ derived from $P \stackrel{b\ l^{\downarrow}!(\tilde{a})}{\longrightarrow} P'$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q + S] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)$ (we consider $n \notin \tilde{c}$ otherwise we would α-rename $\tilde{c}$). Then, considering rules (*StructCRes*) and (*StructCSplit*) we conclude that $n \blacktriangleleft [P] \equiv n \blacktriangleleft [(\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q + S] \mid R)] \equiv (\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft [b \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q + S]] \mid n \blacktriangleleft [R])$. We then have, considering (*StructOutHere*) that:

$$(\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft \left[b \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).Q + S\right]\right] \mid n \blacktriangleleft [R]) \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).n \blacktriangleleft [b \blacktriangleleft [Q]] + S'\right] \mid n \blacktriangleleft [R])$$

Likewise we have that $n \blacktriangleleft [P'] \equiv n \blacktriangleleft [(\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q] \mid R)] \equiv (\boldsymbol{\nu}\tilde{c})(n \blacktriangleleft [b \blacktriangleleft [Q]] \mid n \blacktriangleleft [R])$. Then from (*StructCNest*) we conclude $n \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [n \blacktriangleleft [b \blacktriangleleft [Q]]] \mid n \blacktriangleleft [R])$ which completes the proof for this case. ■

Lemma A.2.12
Let P be a process. If $P \stackrel{l^{d}?(\tilde{a})}{\longrightarrow} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{c})(l^{d}?(\tilde{x}).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(Q\{\tilde{x}/\tilde{a}\} \mid R)$, where $\tilde{a} \# \tilde{c}$.

Proof. By induction on the derivation of $P \stackrel{l^{d}?(\tilde{a})}{\longrightarrow} P'$. Follows the lines of Lemma A.2.10 proof. ■

Lemma A.2.13
Let P be a process. If $P \stackrel{b\ l^{\downarrow}?(\tilde{a})}{\longrightarrow} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [l^{\downarrow}?(\tilde{x}).Q + S] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{c})(b \blacktriangleleft [Q\{\tilde{x}/\tilde{a}\}] \mid R)$, where $\tilde{a}, b \# \tilde{c}$.

Proof. By induction on the derivation of $P \stackrel{b\ l^{\downarrow}?(\tilde{a})}{\longrightarrow} P'$. Proof analogous to that of Lemma A.2.11. ■

Lemma A.2.14
Let P be a process. If $P \stackrel{(\nu\tilde{c})l^{d}!(\tilde{a})}{\longrightarrow} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{b})(l^{d}!(\tilde{a}).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b}')(Q \mid R)$, where $\tilde{a} \cap \tilde{b} = \tilde{c}$ and $\tilde{b}' = \tilde{b} \setminus \tilde{c}$.

Proof. By induction on the derivation of $P \stackrel{(\nu\tilde{c})l^{d}!(\tilde{a})}{\longrightarrow} P'$.

(*Cases* (*Out*) *and* (*Sum*))

Immediate.

(*Case* (*Open*))

We have that $(\boldsymbol{\nu} n)P \xrightarrow{(\nu n,\tilde{c})l^d!(\tilde{a})} P'$ derived from $P \xrightarrow{(\nu\tilde{c})l^d!(\tilde{a})} P'$ and $n \in out((\nu\tilde{c})l^d!(\tilde{a}))$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q + S \mid R)$, and $P' \equiv (\boldsymbol{\nu}\tilde{b'})(Q \mid R)$, where $\tilde{c} = \tilde{a} \cap \tilde{b}$ and $\tilde{b'} = \tilde{b} \setminus \tilde{c}$. We then have $(\boldsymbol{\nu} n)P \equiv (\boldsymbol{\nu} n)(\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q + S \mid R)$, and $n, \tilde{c} = \tilde{a} \cap n, \tilde{b}$ and $\tilde{b'} = n, \tilde{b} \setminus n, \tilde{c}$ which completes the proof for this case.

(*Case* (*Res*))

We have that $(\boldsymbol{\nu} n)P \xrightarrow{(\nu\tilde{c})l^d!(\tilde{a})} (\boldsymbol{\nu} n)P'$ derived from $P \xrightarrow{(\nu\tilde{c})l^d!(\tilde{a})} P'$ and $n \notin na((\nu\tilde{c})l^d!(\tilde{a}))$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q+S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b'})(Q \mid R)$, where $\tilde{a}\cap\tilde{b} = \tilde{c}$ and $\tilde{b'} = \tilde{b}\setminus\tilde{c}$. We then have $(\boldsymbol{\nu} n)P \equiv (\boldsymbol{\nu} n)(\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q+S \mid R)$ and $(\boldsymbol{\nu} n)P' \equiv (\boldsymbol{\nu} n)(\boldsymbol{\nu}\tilde{b'})(Q \mid R)$ where $\tilde{a} \cap \tilde{b}, n = \tilde{c}$ and $n, \tilde{b'} = n, \tilde{b} \setminus \tilde{c}$, which completes the proof for this case.

(*Case* (*Par-l*))

We have that $P_1 \mid P_2 \xrightarrow{(\nu\tilde{c})l^d!(\tilde{a})} P_1' \mid P_2$ derived from $P_1 \xrightarrow{(\nu\tilde{c})l^d!(\tilde{a})} P_1'$ and $\tilde{c} \,\#\, fn(P_2)$. By induction hypothesis we obtain $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q + S \mid R)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b'})(Q \mid R)$, where $\tilde{a} \cap \tilde{b} = \tilde{c}$ and $\tilde{b'} = \tilde{b} \setminus \tilde{c}$. We then have $P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b''})(l^d!(\tilde{a}).Q' + S' \mid R' \mid P_2)$, and $P_1' \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b'''})(Q' \mid R' \mid P_2)$, where $(\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q+S \mid R) \equiv_\alpha (\boldsymbol{\nu}\tilde{b''})(l^d!(\tilde{a}).Q'+S' \mid R')$ and $(\boldsymbol{\nu}\tilde{b'})(Q \mid R) \equiv_\alpha (\boldsymbol{\nu}\tilde{b'''})(Q' \mid R')$ and $\tilde{b''} \,\#\, fn(P_2)$ and $\tilde{a} \cap \tilde{b''} = \tilde{c}$ and $\tilde{b'''} = \tilde{b''} \setminus \tilde{c}$, which completes the proof.

(*Case* (*Rec*))

Follows immediately from induction hypothesis.

(*Case* (*Here*))

We have that $n \blacktriangleleft [P] \xrightarrow{(\nu\tilde{c})l^\downarrow!(\tilde{a})} n \blacktriangleleft [P']$ derived from $P \xrightarrow{(\nu\tilde{c})l^\uparrow!(\tilde{a})} P'$ and $n \notin \tilde{c}$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{b})(l^\uparrow!(\tilde{a}).Q + S \mid R)$ and $\tilde{a} \cap \tilde{b} = \tilde{c}$ and $P' \equiv (\boldsymbol{\nu}\tilde{b'})(Q \mid R)$, where $\tilde{a}\cap\tilde{b} = \tilde{c}$ and $\tilde{b'} = \tilde{b}\setminus\tilde{c}$. We conclude (from rules (*StructCRes*) and (*StructCSplit*)) that $n \blacktriangleleft [P] \equiv n \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(l^\uparrow!(\tilde{a}).Q + S \mid R)\right] \equiv (\boldsymbol{\nu}\tilde{b})(n \blacktriangleleft [l^\uparrow!(\tilde{a}).Q + S] \mid n \blacktriangleleft [R])$. Then (from (*StructOutUp*)) we conclude $(\boldsymbol{\nu}\tilde{b})(n \blacktriangleleft [l^\uparrow!(\tilde{a}).Q + S] \mid n \blacktriangleleft [R]) \equiv (\boldsymbol{\nu}\tilde{b})(l^\downarrow!(\tilde{a}).n \blacktriangleleft [Q] + S' \mid n \blacktriangleleft [R])$. Likewise we conclude $n \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{b'})(n \blacktriangleleft [Q] \mid n \blacktriangleleft [R])$ which completes the proof for this case. ■

Lemma A.2.15

Let P be a process. If $P \xrightarrow{(\nu\tilde{c})n\ l^\downarrow!(\tilde{a})} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{b})(n \blacktriangleleft [l^\downarrow!(\tilde{a}).Q + S] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b'})(n \blacktriangleleft [Q] \mid R)$, where $\tilde{a} \cap \tilde{b} = \tilde{c}$ and $n \notin \tilde{b}$ and $\tilde{b'} = \tilde{b} \setminus \tilde{c}$.

Proof. By induction on the derivation of $P \xrightarrow{(\nu\tilde{c})n\ l^\downarrow!(\tilde{a})} P'$. Follows the lines of the proofs of Lemma A.2.11 and Lemma A.2.14. ■

Lemma A.2.16

Let P be a process. If $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P'$ then $P \equiv (\boldsymbol{\nu}\tilde{b})(l^{d_1}!(\tilde{a}).Q_1 + S_1 \mid l^{d_2}?(\tilde{x}).Q_2 + S_2 \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{a}\} \mid R)$ for d_1, d_2 such that $d_i = \downarrow$ and $d_j = \uparrow$ ($i, j \in \{1, 2\}$).

Proof. By induction on the derivation of $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P'$.

(*Case* (*Res*))

We have $(\boldsymbol{\nu} c)P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} (\boldsymbol{\nu} c)P'$ derived from $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P'$. By induction hypothesis we obtain $P \equiv (\boldsymbol{\nu}\tilde{b})(l^{d_1}!(\tilde{a}).Q_1 + S_1 \mid l^{d_2}?(\tilde{x}).Q_2 + S_2 \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{a}\} \mid R)$. We then have $(\boldsymbol{\nu} c)P \equiv (\boldsymbol{\nu} c, \tilde{b})(l^{d_1}!(\tilde{a}).Q_1 + S_1 \mid l^{d_2}?(\tilde{x}).Q_2 + S_2 \mid R)$ and $(\boldsymbol{\nu} c)P' \equiv (\boldsymbol{\nu} c, \tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{a}\} \mid R)$ which completes the proof for this case.

(*Case* (*Par-l*))

We have $P_1 \mid P_2 \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P_1' \mid P_2$ derived from $P_1 \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P_1'$. By induction hypothesis we obtain $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(l^{d_1}!(\tilde{a}).Q_1+S_1 \mid l^{d_2}?(\tilde{x}).Q_2+S_2 \mid R)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{a}\} \mid R)$. We then have $P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(l^{d_1}!(\tilde{a}).Q_1+S_1 \mid l^{d_2}?(\tilde{x}).Q_2+S_2 \mid R \mid P_2)$ and $P_1' \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{a}\} \mid R \mid P_2)$ (we assume $\tilde{b} \# fn(P_2)$, and otherwise α-rename $\tilde{b}$), which completes the proof.

(*Case* (*Comm*))

We have $P_1 \mid P_2 \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P_1' \mid P_2'$ derived from $P_1 \xrightarrow{\lambda_1} P_1'$ and $P_2 \xrightarrow{\lambda_2} P_2'$ and $\mathtt{this}^{\downarrow\uparrow} = \lambda_1 \bullet \lambda_2$. From the definition of the synchronization algebra we have that λ_1 and λ_2 are such that: one is an input and the other an output; one is defined on the $\uparrow$ direction and the other is defined on the $\downarrow$ direction. We show the case $\lambda_1 = l^{\downarrow}!(\tilde{a})$ and $\lambda_2 = l^{\uparrow}?(\tilde{a})$ and omit remaining cases as their proofs follow similar lines. From $P_1 \xrightarrow{l^{\downarrow}!(\tilde{a})} P_1'$ and considering Lemma A.2.10 we conclude $P_1 \equiv (\boldsymbol{\nu}\tilde{c})(l^{\downarrow}!(\tilde{a}).Q_1 + S_1 \mid R_1)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{c})(Q_1 \mid R_1)$. From $P_2 \xrightarrow{l^{\uparrow}?(\tilde{a})} P_2'$ and considering Lemma A.2.12 we conclude $P_2 \equiv (\boldsymbol{\nu}\tilde{b})(l^{\uparrow}?(\tilde{x}).Q_2 + S_2 \mid R_2)$ and $P_2 \equiv (\boldsymbol{\nu}\tilde{b})(Q_2\{\tilde{x}/\tilde{a}\} \mid R_2)$. We then have:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c})(l^{\downarrow}!(\tilde{a}).Q_1 + S_1 \mid R_1) \mid (\boldsymbol{\nu}\tilde{b})(l^{\uparrow}?(\tilde{x}).Q_2 + S_2 \mid R_2)$$

and thus, considering the necessary α-renaming:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{c}')(\boldsymbol{\nu}\tilde{b}')(l^{\downarrow}!(\tilde{a}).Q_1' + S_1' \mid l^{\uparrow}?(\tilde{x}).Q_2' + S_2' \mid R_2' \mid R_1')$$

Likewise we conclude

$$P_1' \mid P_2' \equiv (\boldsymbol{\nu}\tilde{c}')(\boldsymbol{\nu}\tilde{b}')(Q_1' \mid Q_2'\{\tilde{x}/\tilde{a}\} \mid R_2' \mid R_1')$$

which completes the proof for this case.

(*Case* (*Close*))

Follows lines similar to the case of rule (*Comm*).

(*Case* (*Rec*))

Follows directly from the induction hypothesis. ■

Lemma A.2.17

Let P be a process. If $P \xrightarrow{c\ \mathtt{this}^d} P'$ then we have that either $P \equiv (\boldsymbol{\nu}\tilde{b})(\mathbf{this}(x).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R)$; or $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$ and either: $\alpha_1 = l^d?(\tilde{x})$ and $\alpha_2 = l^{\downarrow}!(\tilde{a})$ in which case $Q_1' = Q_1\{\tilde{x}/\tilde{a}\}$ and $Q_2' = Q_2$; or $\alpha_1 = l^d!(\tilde{a})$ and $\alpha_2 = l^{\downarrow}?(\tilde{x})$ in which case $Q_1' = Q_1$ and $Q_2' = Q_2\{\tilde{x}/\tilde{a}\}$.

Proof. By induction on the length of the derivation of $P \xrightarrow{c\ \mathtt{this}^d} P'$.

(*Case* (*This*) *and* (*Sum*))

Immediate.

(*Case* (*Res*))

We have $(\boldsymbol{\nu}a)P \xrightarrow{c\ \mathtt{this}^d} (\boldsymbol{\nu}a)P'$ derived from $P \xrightarrow{c\ \mathtt{this}^d} P'$ and $c \neq a$. By induction hypothesis we obtain that either $P \equiv (\boldsymbol{\nu}\tilde{b})(\mathbf{this}(x).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R)$; or $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$and $P \equiv (\boldsymbol{\nu}\tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$, and either $\alpha_1 = l^d?(\tilde{x})$ and $\alpha_2 = l^{\downarrow}!(\tilde{m})$; or $\alpha_1 = l^d!(\tilde{m})$ and $\alpha_2 = l^{\downarrow}?(\tilde{x})$. If $P \equiv (\boldsymbol{\nu}\tilde{b})(\mathbf{this}(x).Q + S \mid R)$ then we have $(\boldsymbol{\nu}a)P \equiv (\boldsymbol{\nu}a, \tilde{b})(\mathbf{this}(x).Q + S \mid R)$ and $(\boldsymbol{\nu}a)P' \equiv (\boldsymbol{\nu}a, \tilde{b})(Q\{x/c\} \mid R)$. If $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ then we have $(\boldsymbol{\nu}a)P \equiv (\boldsymbol{\nu}a, \tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and

$(\boldsymbol{\nu} a)P' \equiv (\boldsymbol{\nu} a, \tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$ which completes the proof for this case.

(*Case* (*Par-l*))

We have $P_1 \mid P_2 \xrightarrow{c\ \mathtt{this}^d} P_1' \mid P_2$ derived from $P_1 \xrightarrow{c\ \mathtt{this}^d} P_1'$. By induction hypothesis we obtain that either $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R)$; or $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$, and either $\alpha_1 = l^d?(\tilde{x})$ and $\alpha_2 = l^{\downarrow}!(\tilde{a})$; or $\alpha_1 = l^d!(\tilde{a})$ and $\alpha_2 = l^{\downarrow}?(\tilde{x})$. If $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R)$ then we have $P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R \mid P_2)$ and $P_1' \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R \mid P_2)$. If $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ then we have $P_1 \mid P_2 \equiv (\boldsymbol{\nu} c, \tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R \mid P_2)$ and $P_1' \mid P_2 \equiv (\boldsymbol{\nu} c, \tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R \mid P_2)$ (we assume $\tilde{b} \# fn(P_2)$, and otherwise α-rename $\tilde{b}$), which completes the proof.

(*Case* (*Comm*))

We have $P_1 \mid P_2 \xrightarrow{c\ \mathtt{this}^d} P_1' \mid P_2'$ derived from $P_1 \xrightarrow{\lambda_1} P_1'$ and $P_2 \xrightarrow{\lambda_2} P_2'$ and $c\ \mathtt{this}^d = \lambda_1 \bullet \lambda_2$. From the definition of the synchronization algebra we have that λ_1 and λ_2 are such that: one is an input and the other an output; one is a $\downarrow$ located label (at c) and the other is defined on the d direction. We show the case $\lambda_1 = l^d!(\tilde{a})$ and $\lambda_2 = c\ l^{\downarrow}?(\tilde{a})$ and omit remaining cases as their proofs follow similar lines. From $P_1 \xrightarrow{l^d!(\tilde{a})} P_1'$ and considering Lemma A.2.10 we conclude $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q_1 + S_1 \mid R_1)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid R_1)$. From $P_2 \xrightarrow{c\ l^{\downarrow}?(\tilde{a})} P_2'$ and considering Lemma A.2.13 we conclude $P_2 \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [l^{\downarrow}?(\tilde{x}).Q_2 + S_2] \mid R_2)$ and $P_2' \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [Q_2\{\tilde{x}/\tilde{a}\}] \mid R_2)$. We then have:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(l^d!(\tilde{a}).Q_1 + S_1 \mid R_1) \mid (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_2 + S_2\right] \mid R_2)$$

and thus, considering the necessary α-renaming:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(l^d!(\tilde{a}).Q_1' + S_1' \mid c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_2' + S_2'\right] \mid R_2' \mid R_1')$$

Likewise we obtain:

$$P_1' \mid P_2' \equiv (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(Q_1' \mid c \blacktriangleleft \left[Q_2'\{\tilde{x}/\tilde{a}\}\right] \mid R_2' \mid R_1')$$

which completes the proof for this case.

(*Case* (*Close*))

Follows lines similar to the case of rule (*Comm*).

(*Case* (*Rec*))

Follows directly from the induction hypothesis.

(*Case* (*Here*))

We have $a \blacktriangleleft [P] \xrightarrow{c\ \mathtt{this}^{\downarrow}} a \blacktriangleleft [P']$ derived from $P \xrightarrow{c\ \mathtt{this}^{\uparrow}} P'$. By induction hypothesis we obtain that either $P \equiv (\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R)$; or $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$, and either $\alpha_1 = l^d?(\tilde{x})$ and $\alpha_2 = l^{\downarrow}!(\tilde{m})$; or $\alpha_1 = l^d!(\tilde{m})$ and $\alpha_2 = l^{\downarrow}?(\tilde{x})$. Since the $c\ \mathtt{this}^{\uparrow}$ label is $\uparrow$ directed we have that it must be the case that $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and α_1 is defined on the $\uparrow$ direction. We show the case α_1 is the input and α_2 is the output. We have that:

$$a \blacktriangleleft [P] \equiv a \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(l^{\uparrow}?(\tilde{x}).Q_1 + S_1 \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m}).Q_2 + S_2\right] \mid R)\right]$$

We then have, up to some α-renaming, that:

$$a \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(a \blacktriangleleft \left[l^{\uparrow}?(\tilde{x}).Q'_1 + S'_1 \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).Q'_2 + S'_2\right] \mid R'\right])$$

Then considering (*StructCSplit*) we conclude that:

$$a \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(a \blacktriangleleft \left[l^{\uparrow}?(\tilde{x}).Q'_1 + S'_1\right] \mid a \blacktriangleleft \left[c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).Q'_2 + S'_2\right]\right] \mid a \blacktriangleleft \left[R'\right])$$

Then considering (*StructInUp*) and (*StructOutHere*) we have:

$$a \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(l^{\downarrow}?(\tilde{x}).a \blacktriangleleft \left[Q'_1\right] + S''_1 \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).a \blacktriangleleft \left[c \blacktriangleleft \left[Q'_2\right]\right] + S''_2\right] \mid a \blacktriangleleft \left[R'\right])$$

for some S''_1 and S''_2. Similarly we conclude:

$$a \blacktriangleleft \left[P'\right] \equiv (\boldsymbol{\nu}\tilde{b'})(a \blacktriangleleft \left[Q'_1\{\tilde{x}/\tilde{m}\}\right] \mid c \blacktriangleleft \left[a \blacktriangleleft \left[c \blacktriangleleft \left[Q'_2\right]\right]\right] \mid a \blacktriangleleft \left[R'\right])$$

which completes the proof for this case.

(*Case* (*ThisHere*))

We have $c \blacktriangleleft [P] \xrightarrow{c\ \mathtt{this}^{\downarrow}} c \blacktriangleleft [P']$ derived from $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} P'$. Considering Lemma A.2.16 we have there is d_1, d_2 such that $d_i = \downarrow$ and $d_j = \uparrow$ for $i, j \in \{1, 2\}$, and $P \equiv (\boldsymbol{\nu}\tilde{b})(l^{d_1}!(\tilde{m}).Q_1 + S_1 \mid l^{d_2}?(\tilde{x}).Q_2 + S_2 \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1 \mid Q_2\{\tilde{x}/\tilde{m}\} \mid R)$. We show the case d_1 is $\downarrow$ and d_2 is $\uparrow$. We have that:

$$c \blacktriangleleft [P] \equiv c \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(l^{\downarrow}!(\tilde{m}).Q_1 + S_1 \mid l^{\uparrow}?(\tilde{x}).Q_2 + S_2 \mid R)\right]$$

We then have, up to some α-renaming, that:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).Q'_1 + S'_1 \mid l^{\uparrow}?(\tilde{x}).Q'_2 + S'_2 \mid R'\right]$$

Then considering (*StructCSplit*) we conclude that:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).Q'_1 + S'_1\right] \mid c \blacktriangleleft \left[l^{\uparrow}?(\tilde{x}).Q'_2 + S'_2\right] \mid c \blacktriangleleft \left[R'\right])$$

Then considering (*StructInUp*) we have:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{m'}).Q'_1 + S'_1\right] \mid l^{\downarrow}?(\tilde{x}).c \blacktriangleleft \left[Q'_2\right] + S''_2 \mid c \blacktriangleleft \left[R'\right])$$

for some S''_2. Similarly we conclude:

$$c \blacktriangleleft \left[P'\right] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q'_1\right] \mid c \blacktriangleleft \left[Q'_2\{\tilde{x}/\tilde{m'}\}\right] \mid c \blacktriangleleft \left[R'\right])$$

which completes the proof for this last case. ■

Theorem 4.4.21 (Reductions Match τ-Transitions)
(repetition of the statement in page 88)
Let P, Q be processes. We have that $P \to Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.

Proof. We prove the two directions: $P \to Q$ implies $P \xrightarrow{\tau} \equiv Q$ and $P \xrightarrow{\tau} Q$ implies $P \to Q$.

($P \to Q$ *implies* $P \xrightarrow{\tau} \equiv Q$)

By induction on the length of the derivation of $P \to Q$. For the base cases of (*RedComm*) and (*RedThis*) we immediately derive a τ-transition. (*RedRes*), (*RedPar*) and (*RedPiece*) follow directly from induction hypothesis. Case (*RedStruct*) follows from Lemma A.2.9: We have $P \to Q$ derived from $P \equiv P' \to Q' \equiv Q$; by induction hypothesis on $P' \to Q'$ we obtain $P' \xrightarrow{\tau} Q''$ and $Q'' \equiv Q'$. Then from $P \equiv P'$ and considering Lemma A.2.9 we conclude $P \xrightarrow{\tau} Q'''$ and $Q''' \equiv Q''$, which along with $Q'' \equiv Q' \equiv Q$ gives us $P \xrightarrow{\tau}\equiv Q$ as intended.

($P \xrightarrow{\tau} Q$ *implies* $P \to Q$)

By induction on the length of the derivation of $P \xrightarrow{\tau} Q$.

(*Cases* (*Res*), (*Par*), (*Rec*) *and* (*Tau*))

Follow from induction hypothesis in expected lines.

(*Case* (*Comm*))

We have $P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'$ derived from $P_1 \xrightarrow{\lambda_1} P_1'$ and $P_2 \xrightarrow{\lambda_2} P_2'$ and $\tau = \lambda_1 \bullet \lambda_2$. From the definition of the synchronization algebra we have that λ_1 and λ_2 are such that: one is an input and the other an output; either both λ_1 and λ_2 are $\downarrow$ located labels or they are both unlocated labels with matching direction. We show the case $\lambda_1 = c\ l^{\downarrow}!(\tilde{a})$ and $\lambda_2 = c\ l^{\downarrow}?(\tilde{a})$ and omit remaining cases as their proofs follow similar lines. From $P_1 \xrightarrow{c\ l^{\downarrow}!(\tilde{a})} P_1'$ and considering Lemma A.2.11 we conclude $P_1 \equiv (\boldsymbol{\nu}\tilde{b})(c \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q_1 + S_1] \mid R_1)$ and $P_1' \equiv (\boldsymbol{\nu}\tilde{b})(c \blacktriangleleft [Q_1] \mid R_1)$. From $P_2 \xrightarrow{c\ l^{\downarrow}?(\tilde{a})} P_2'$ and considering Lemma A.2.13 we conclude $P_2 \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [l^{\downarrow}?(\tilde{x}).Q_2 + S_2] \mid R_2)$ and $P_2' \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [Q_2\{\tilde{x}/\tilde{a}\}] \mid R_2)$. We then have:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b})(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).Q_1 + S_1\right] \mid R_1) \mid (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_2 + S_2\right] \mid R_2)$$

and thus, considering the necessary α-renaming:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).Q_1' + S_1'\right] \mid c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_2' + S_2'\right] \mid R_2' \mid R_1')$$

Considering structural congruence rule (*StructCSplit*) we have:

$$P_1 \mid P_2 \equiv (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).Q_1' + S_1' \mid l^{\downarrow}?(\tilde{x}).Q_2' + S_2'\right] \mid R_2' \mid R_1')$$

Similarly we conclude:

$$P_1' \mid P_2' \equiv (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(c \blacktriangleleft \left[Q_1' \mid (Q_2'\{\tilde{x}/\tilde{a}\})\right] \mid R_2' \mid R_1')$$

We thus have:

$$\begin{array}{lll} P_1 \mid P_2 & \equiv & (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(c \blacktriangleleft [l^{\downarrow}!(\tilde{a}).Q_1' + S_1' \mid l^{\downarrow}?(\tilde{x}).Q_2' + S_2'] \mid R_2' \mid R_1') \\ & & \to (\boldsymbol{\nu}\tilde{b''})(\boldsymbol{\nu}\tilde{b'''})(c \blacktriangleleft [Q_1' \mid (Q_2'\{\tilde{x}/\tilde{a}\})] \mid R_2' \mid R_1') \qquad \equiv \quad P_1' \mid P_2' \end{array}$$

which completes the proof for this case.

(*Case* (*Close*))

Follows lines similar to the case of rule (*Comm*).

(*Case* (*ThisLoc*))

We have $c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [P']$ derived from $P \xrightarrow{c\ \mathtt{this}^{\downarrow}} P'$. Considering Lemma A.2.17 we have that either $P \equiv (\boldsymbol{\nu}\tilde{b})(\textbf{this}(x).Q{+}S \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q\{x/c\} \mid R)$; or $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1{+}S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$ and $P' \equiv (\boldsymbol{\nu}\tilde{b})(Q_1' \mid c \blacktriangleleft [Q_2'] \mid R)$ and either: $\alpha_1 = l^d?(\tilde{x})$ and $\alpha_2 = l^{\downarrow}!(\tilde{a})$ in

which case $Q_1' = Q_1\{\tilde{x}/\tilde{a}\}$ and $Q_2' = Q_2$; or $\alpha_1 = l^d!(\tilde{a})$ and $\alpha_2 = l^{\downarrow}?(\tilde{x})$ in which case $Q_1' = Q_1$ and $Q_2' = Q_2\{\tilde{x}/\tilde{a}\}$.

If $P \equiv (\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R)$ then we have that $c \blacktriangleleft [P] \equiv c \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(\textsf{this}(x).Q + S \mid R)\right]$. Then (up to some α-renaming) we have $c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})c \blacktriangleleft [\textsf{this}(x).Q' + S' \mid R']$ and considering (*StructCSplit*) we obtain $c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [\textsf{this}(x).Q' + S'] \mid c \blacktriangleleft [R'])$. Similarly we conclude $c \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [Q'\{x/c\}] \mid c \blacktriangleleft [R'])$. We thus have:

$$\begin{array}{rcl} c \blacktriangleleft [P] & \equiv & (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [\textsf{this}(x).Q' + S'] \mid c \blacktriangleleft [R']) \\ & & \to (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft [Q'\{x/c\}] \mid c \blacktriangleleft [R']) \qquad \equiv \quad c \blacktriangleleft [P'] \end{array}$$

which completes the proof for this case.

We now consider the case when $P \equiv (\boldsymbol{\nu}\tilde{b})(\alpha_1.Q_1 + S_1 \mid c \blacktriangleleft [\alpha_2.Q_2 + S_2] \mid R)$. Since the $c\,\texttt{this}^{\downarrow}$ label is $\downarrow$ directed we have that it must be the case that α_1 is defined on the $\downarrow$ direction. We show the case α_1 is the input and α_2 is the output. We have that:

$$c \blacktriangleleft [P] \equiv c \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(l^{\downarrow}?(\tilde{x}).Q_1 + S_1 \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a}).Q_2 + S_2\right] \mid R)\right]$$

We then have, up to some α-renaming, that:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_1' + S_1' \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a'}).Q_2' + S_2'\right] \mid R'\right])$$

Then considering (*StructCSplit*) we conclude that:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_1' + S_1'\right] \mid c \blacktriangleleft \left[c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a'}).Q_2' + S_2'\right]\right] \mid c \blacktriangleleft \left[R'\right])$$

We now consider (*StructOutHere*) and obtain:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_1' + S_1'\right] \mid c \blacktriangleleft \left[l^{\downarrow}!(\tilde{a'}).c \blacktriangleleft \left[c \blacktriangleleft \left[Q_2'\right]\right] + S_2''\right] \mid c \blacktriangleleft \left[R'\right])$$

for some S_2''. Then, again through (*StructCSplit*) we obtain:

$$c \blacktriangleleft [P] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_1' + S_1' \mid l^{\downarrow}!(\tilde{a'}).c \blacktriangleleft \left[c \blacktriangleleft \left[Q_2'\right]\right] + S_2''\right] \mid c \blacktriangleleft \left[R'\right]) \tag{A.2.18.1}$$

We also have that:

$$c \blacktriangleleft \left[P'\right] \equiv c \blacktriangleleft \left[(\boldsymbol{\nu}\tilde{b})(Q_1\{\tilde{x}/\tilde{a}\} \mid c \blacktriangleleft [Q_2] \mid R)\right]$$

As before, up to some α-renaming, we obtain:

$$c \blacktriangleleft \left[P'\right] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q_1'\{\tilde{x}/\tilde{a'}\} \mid c \blacktriangleleft \left[Q_2'\right] \mid R'\right])$$

And considering (*StructCSplit*) we obtain:

$$c \blacktriangleleft \left[P'\right] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q_1'\{\tilde{x}/\tilde{a'}\}\right] \mid c \blacktriangleleft \left[c \blacktriangleleft \left[Q_2'\right]\right] \mid c \blacktriangleleft \left[R'\right])$$

We now consider (*StructCNest*) and obtain:

$$c \blacktriangleleft \left[P'\right] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q_1'\{\tilde{x}/\tilde{a'}\}\right] \mid c \blacktriangleleft \left[c \blacktriangleleft \left[c \blacktriangleleft \left[Q_2'\right]\right]\right] \mid c \blacktriangleleft \left[R'\right])$$

Then, again through (*StructCSplit*) we have:

$$c \blacktriangleleft [P'] \equiv (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q_1'\{\tilde{x}/\tilde{a'}\} \mid c \blacktriangleleft [c \blacktriangleleft [Q_2']]\right] \mid c \blacktriangleleft [R']) \tag{A.2.18.2}$$

(A.2.18.1) and (A.2.18.2) allow us to state our intended result:

$$\begin{aligned} c \blacktriangleleft [P] \;\equiv\; & (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[l^{\downarrow}?(\tilde{x}).Q_1' + S_1' \mid l^{\downarrow}!(\tilde{a'}).c \blacktriangleleft [c \blacktriangleleft [Q_2']] + S_2''\right] \mid c \blacktriangleleft [R']) \\ & \to (\boldsymbol{\nu}\tilde{b'})(c \blacktriangleleft \left[Q_1'\{\tilde{x}/\tilde{a'}\} \mid c \blacktriangleleft [c \blacktriangleleft [Q_2']]\right] \mid c \blacktriangleleft [R']) \qquad \equiv \quad c \blacktriangleleft [P'] \end{aligned}$$

thus completing the proof for this case.

A.3 Chapter 5

Proposition 5.2.19 (Behavioral Type Merge Relation Properties)
(repetition of the statement in page 115)
The behavioral types merge relation is commutative and associative:

(1). If $B = B_1 \bowtie B_2$ then $B = B_2 \bowtie B_1$.

(2). If $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$ then there is B'' such that $B'' = B_2 \bowtie B_3$ and $B = B_1 \bowtie B''$.

Proof.

(*1*). Follows immediately from the definition.

(*2*). By induction on the derivation of $B' = B_1 \bowtie B_2$ and $B = B' \bowtie B_3$. We show the case when $B' = B_1 \bowtie B_2$ is derived using rule (*Plain-r*) and $B = B' \bowtie B_3$ is derived using rule (*Shuffle-l*), hence $B_1 \equiv \,!\, l(C).B_a$ and $B_2 \equiv \,?\, l(C).B_b$. We have:

$$\tau\, l(C).B' \mid B'' = \tau\, l(C).(B_a \bowtie B_b) \bowtie B_3 \tag{A.3.1.1}$$

derived from:

$$B' \mid B'' \equiv (B_a \bowtie B_b) \bowtie B_3 \tag{A.3.1.2}$$

and $\tau\, l(C) \# B_3$, $\tau\, l(C) \# B''$ and $\mathcal{I}(B') \subseteq \mathcal{I}(B_a \bowtie B_b)$ and $\mathcal{I}(B'') \subseteq \mathcal{I}(B_3)$. We also have that:

$$\tau\, l(C).(B_a \bowtie B_b) = \,!\, l(C).B_a \bowtie \,?\, l(C).B_b \tag{A.3.1.3}$$

We intend to prove: $\tau\, l(C).B' \mid B'' = \,!\, l(C).B_a \bowtie (?\, l(C).B_b \bowtie B_3)$. By induction hypothesis on (A.3.1.2) we have:

$$B' \mid B'' \equiv B_a \bowtie (B_b \bowtie B_3) \tag{A.3.1.4}$$

From (A.3.1.4) and $\tau\, l(C) \# B_3$ we have that there is B_c, B_d such that:

$$?\, l(C).B_c \mid B_d = \,?\, l(C).B_b \bowtie B_3 \tag{A.3.1.5}$$

derived from:

$$B_c \mid B_d \equiv B_b \bowtie B_3 \tag{A.3.1.6}$$

and $?\,l(C)\,\#\,B_d$ and $\mathcal{I}(B_c) \subseteq \mathcal{I}(B_b)$ and $\mathcal{I}(B_d) \subseteq \mathcal{I}(B_3)$. We then derive (using rule (*ShufflePar-r*)):

$$(!\,l(C).B_a \bowtie ?\,l(C).B_c) \bowtie B_d = !\,l(C).B_a \bowtie ?\,l(C).B_c \mid B_d \tag{A.3.1.7}$$

and then by rule (*Plain-r*) we derive:

$$\tau\,l(C).(B_a \bowtie B_c) \bowtie B_d = (!\,l(C).B_a \bowtie ?\,l(C).B_c) \bowtie B_d \tag{A.3.1.8}$$

We then have, by rule (*Shuffle-l*):

$$\tau\,l(C).B_e \mid B_f = \tau\,l(C).(B_a \bowtie B_c) \bowtie B_d \tag{A.3.1.9}$$

derived from:

$$B_e \mid B_f \equiv (B_a \bowtie B_c) \bowtie B_d \tag{A.3.1.10}$$

and $\tau\,l(C)\,\#\,B_f$ and $\mathcal{I}(B_e) \subseteq \mathcal{I}(B_a \bowtie B_c)$ and $\mathcal{I}(B_f) \subseteq \mathcal{I}(B_d)$. By induction hypothesis on (A.3.1.10) we conclude:

$$(B_a \bowtie B_c) \bowtie B_d \equiv B_a \bowtie (B_c \bowtie B_d) \tag{A.3.1.11}$$

From (A.3.1.6) we have that B_c and B_d are apart, hence:

$$B_a \bowtie (B_c \bowtie B_d) \equiv B_a \bowtie (B_c \mid B_d) \tag{A.3.1.12}$$

Also from (A.3.1.6) we conclude:

$$B_a \bowtie (B_c \mid B_d) \equiv B_a \bowtie (B_b \bowtie B_3) \tag{A.3.1.13}$$

From (A.3.1.13), (A.3.1.12), (A.3.1.11) and (A.3.1.10) we conclude:

$$B_e \mid B_f \equiv B_a \bowtie (B_b \bowtie B_3) \tag{A.3.1.14}$$

From (A.3.1.14) and (A.3.1.4) we conclude:

$$B_e \mid B_f \equiv B' \mid B'' \tag{A.3.1.15}$$

From $\mathcal{I}(B_f) \subseteq \mathcal{I}(B_d) \subseteq \mathcal{I}(B_3)$ and $\mathcal{I}(B'') \subseteq \{\tilde{B}\}$ we have $B_e \equiv B'$ and $B_f \equiv B''$, from which we conclude — rule (*Shuffle-l*):

$$\tau\,l(C).B' \mid B'' = \tau\,l(C).(B_a \bowtie B_c) \bowtie B_d \tag{A.3.1.16}$$

which completes the proof. ∎

Proposition 5.3.3 (Weakening)
(repetition of the statement in page 121)
Let P be a well-typed process such that $P :: T$. If exponential output located type $\star L^!$ is such that T and $\star L^!$ are apart, hence $T \# \star L^!$, then $P :: T \mid \star L^!$.

Proof. By induction on the length of the derivation of $P :: T$. The exponential output located typed is introduced at the level of the axioms (*Stop*) and (*RecVar*), for which we directly have:

$$\overline{\mathbf{0} : \star L_1^! \mid \star L^!}$$

and:

$$\overline{\mathcal{X} : \star L_1^! \mid \star L^! \mid \mathcal{X}}$$

The proof of the remaining rules follows directly from induction hypothesis, where some care needs to be taken (baring in mind that α-equivalent processes are identified) when considering the bound identifiers in rule (*Res*) and (*Input*). We show the cases for rules (*Par*) and (*Piece*).

(*Rule* (*Par*))

We have:

$$P_1 \mid P_2 :: T \tag{A.3.2.1}$$

derived from:

$$P_1 :: T_1 \qquad \text{and} \qquad P_2 :: T_2 \qquad \text{and} \qquad T = T_1 \bowtie T_2 \tag{A.3.2.2}$$

Let us consider $\star L^!$ such that $T \# \star L^!$. Considering (A.3.2.2), we then have:

$$T_1 \# \star L^! \qquad \text{and} \qquad T_2 \# \star L^! \tag{A.3.2.3}$$

By induction hypothesis on (A.3.2.2) and (A.3.2.3) we have:

$$P_1 :: T_1 \mid \star L^! \tag{A.3.2.4}$$

From (A.3.2.2) and (A.3.2.3) we conclude:

$$T \mid \star L^! = (T_1 \mid \star L^!) \bowtie T_2 \tag{A.3.2.5}$$

and thus:

$$P_1 \mid P_2 :: T \mid \star L^! \tag{A.3.2.6}$$

which completes the proof for this case.

(*Rule* (*Piece*))

We have:

$$n \blacktriangleleft [P] :: T \tag{A.3.2.7}$$

derived from:

$$P :: L \mid B \qquad \text{and} \qquad T = (L \bowtie n : [\downarrow B]) \mid loc(\uparrow B) \tag{A.3.2.8}$$

Let us consider $\star L^!$ such that $T \# \star L^!$. We then have that $L \# \star L^!$ and thus $L \mid B \# \star L^!$. By induction hypothesis we conclude:

$$P :: L \mid B \mid \star L^! \tag{A.3.2.9}$$

Since $T \# \star L^!$ we have that $n : [\downarrow B] \# \star L^!$:

$$T \mid \star L^! = ((L \mid \star L^!) \bowtie n : [\downarrow B]) \mid loc(\uparrow B) \tag{A.3.2.10}$$

which gives us our intended result:

$$n \blacktriangleleft [P] :: T \mid \star L^! \tag{A.3.2.11}$$

and completes the proof. ∎

Lemma 5.3.4 (Substitution Lemma)
(repetition of the statement in page 121)
Let P be a well typed process such that $P :: T \mid x : C$ and $x \notin dom(T)$. If there is T' such that $T' = T \bowtie a : C$ then $P\{x/a\} :: T'$.

Proof. By induction on the length of the derivation of $P :: T \mid x : C$. We show the cases of rule (*Piece*), when P of the form $x \blacktriangleleft [Q]$, and rule (*Output*), when P is of the form $l^d!(x).P$.

(*Rule* (*Piece*)) We have:

$$x \blacktriangleleft [Q] :: T \mid x : C \tag{A.3.3.1}$$

where $x \notin dom(T)$. Since (A.3.3.1) is a conclusion of rule (*Piece*) we have that:

$$T \mid x : C \equiv (L \bowtie x : [\downarrow B]) \mid loc(\uparrow B) \tag{A.3.3.2}$$

for some L, B such that:

$$Q :: L \mid B \tag{A.3.3.3}$$

In order to apply the induction hypothesis we first characterize types L, C and T. We separate L into two parts L' and $x : C'$ such that L' does not mention x (i.e., $x \notin dom(L')$):

$$L \equiv L' \mid x : C' \tag{A.3.3.4}$$

From (A.3.3.2) we have that $x : C$ is the result of the merge of $x : [\downarrow B]$ and the type that L specifies for x (which we identify in (A.3.3.4) as $x : C'$), hence:

$$x : C = x : C' \bowtie x : [\downarrow B] \tag{A.3.3.5}$$

Also, since L' does not mention x we have that $(L' \mid x : C') \bowtie x : [\downarrow B]$ yields the same type as $L' \mid (x : C' \bowtie x : [\downarrow B])$, hence from (A.3.3.2) we have that T is such that:

$$T \equiv L' \mid loc(\uparrow B) \tag{A.3.3.6}$$

We now assume the hypothesis in the statement of the Lemma: there is type T' such that:

$$T' = T \bowtie a : C \tag{A.3.3.7}$$

Since L' is a part of T (A.3.3.6) and C' is a partial view of C (A.3.3.5), from (A.3.3.7) we conclude there is type T'' such that $T'' = L' \bowtie a : C'$ and hence:

$$T'' \mid B = (L' \mid B) \bowtie a : C' \tag{A.3.3.8}$$

We rewrite (A.3.3.3) considering (A.3.3.4), using the subsumption rule, and obtain:

$$Q :: L' \mid B \mid x : C' \tag{A.3.3.9}$$

By induction hypothesis on (A.3.3.9) and (A.3.3.8) we conclude $Q\{x/a\} :: (L' \mid B) \bowtie a : C'$, from which we obtain:

$$Q\{x/a\} :: (L' \bowtie a : C') \mid B \tag{A.3.3.10}$$

From (A.3.3.7), considering L' is a part of T (A.3.3.6) and separating C in the partial views given by (A.3.3.5) we obtain there is type T''' such that:

$$T''' = L' \bowtie (a : C' \bowtie a : [\downarrow B]) \tag{A.3.3.11}$$

From (A.3.3.10) and (A.3.3.11) and considering rule (*Piece*) we derive:

$$a \blacktriangleleft [(Q\{x/a\})] :: ((L' \bowtie a : C') \bowtie a : [\downarrow B]) \mid loc(\uparrow B) \tag{A.3.3.12}$$

from which we conclude:

$$a \blacktriangleleft [(Q\{x/a\})] :: (L' \mid loc(\uparrow B)) \bowtie (a : C' \bowtie a : [\downarrow B]) \tag{A.3.3.13}$$

From (A.3.3.13), considering (A.3.3.6) and (A.3.3.5), we conclude:

$$a \blacktriangleleft [(Q\{x/a\})] :: T \bowtie a : C \tag{A.3.3.14}$$

which completes the proof for this case.

(*Rule* (*Output*))

We have:

$$l^d!(x).Q :: T \mid x : C \tag{A.3.3.15}$$

where $x \notin dom(T)$. We have (A.3.3.15) is derived from:

$$Q :: L \mid B \tag{A.3.3.16}$$

and:

$$T \mid x : C \equiv (L \bowtie x : C') \mid \oplus\{!\, l^d(C').B; \tilde{B}\} \tag{A.3.3.17}$$

In order to apply the induction hypothesis we first characterize types L, C and T. We separate L into two parts L' and $x : C''$ such that L' does not mention x (i.e., $x \notin dom(L')$):

$$L \equiv L' \mid x : C'' \tag{A.3.3.18}$$

From (A.3.3.17) we have that $x : C$ is the result of the merge of $x : C'$ with the type that L specifies for x (which we identify in (A.3.3.18) as C''), hence:

$$x : C = x : C'' \bowtie x : C' \tag{A.3.3.19}$$

Also, since L' does not mention x we have that $(L' \mid x : C'') \bowtie x : C'$ yields the same type as

$L' \mid (x : C'' \bowtie x : C')$, hence from (A.3.3.17) we have:

$$T \equiv L' \mid \oplus\{\,!\, l^d(C').B; \tilde{B}\} \qquad \text{(A.3.3.20)}$$

We now assume the hypothesis in the statement of the Lemma: there is type T' such that:

$$T' = T \bowtie a : C \qquad \text{(A.3.3.21)}$$

Since L' is a part of T (A.3.3.20) and C'' is a partial view of C (A.3.3.19), from (A.3.3.21) we conclude there is type T'' such that $T'' = L' \bowtie a : C''$ and hence:

$$T'' \mid B = (L' \mid B) \bowtie a : C'' \qquad \text{(A.3.3.22)}$$

We rewrite (A.3.3.16) considering (A.3.3.18), using the subsumption rule, and obtain:

$$Q :: L' \mid B \mid x : C'' \qquad \text{(A.3.3.23)}$$

By induction hypothesis on (A.3.3.23) and (A.3.3.22) we conclude $Q\{x/a\} :: (L' \mid B) \bowtie a : C''$, from which we obtain:

$$Q\{x/a\} :: (L' \bowtie a : C'') \mid B \qquad \text{(A.3.3.24)}$$

From (A.3.3.21), considering L' is a part of T (A.3.3.20) and separating C in the partial views given by (A.3.3.19) we obtain there is type T''' such that:

$$T''' = L' \bowtie (a : C'' \bowtie a : C') \qquad \text{(A.3.3.25)}$$

From (A.3.3.24) and (A.3.3.25) and considering rule (*Output*) we derive:

$$l^d!(a).(Q\{x/a\}) :: ((L' \bowtie a : C'') \bowtie a : C') \mid \oplus\{\,!\, l^d(C').B; \tilde{B}\} \qquad \text{(A.3.3.26)}$$

from which we conclude:

$$l^d!(a).(Q\{x/a\}) :: (L' \mid \oplus\{\,!\, l^d(C').B; \tilde{B}\}) \bowtie (a : C'' \bowtie a : C') \qquad \text{(A.3.3.27)}$$

From (A.3.3.27), considering (A.3.3.20) and (A.3.3.19), we conclude:

$$l^d!(a).(Q\{x/a\}) :: T \bowtie a : C \qquad \text{(A.3.3.28)}$$

which completes the proof for this case. ∎

We state some auxiliary results to the proof of Theorem 5.3.5. Such auxiliary results allows us to identify the typings that characterize processes that exhibit a determined behavior. Then, using such type information, we may study the several kinds of internal interactions in the proof of Theorem 5.3.5 along with the typing characterizations for such interactions. We start by some results that identify some invariant features of types related by subtyping.

Lemma A.3.4

Let P be a well-typed process $P :: T$. If $T \equiv T' \mid\, !\, l^d(C).B$ then $P :: T' \mid \oplus_{i \in I}\{\,!l^d(C).B; \tilde{B}\}$

where $\tilde{B} = {!}\, l_1^d(C_1).B_1; \ldots ; {!}\, l_k^d(C_k).B_k$. *Likewise if* $T \equiv T' \mid c : [{!}\, l^d(C).B]$ *then* $P :: T' \mid c : [\oplus_{i \in I}\{{!}\, l^d(C).B; \tilde{B}\}]$ *where* $\tilde{B} = {!}\, l_1^d(C_1).B_1; \ldots ; {!}\, l_k^d(C_k).B_k$.

Proof. Follows by induction on the derivation of $P :: T \mid {!}\, l^d(C).B$ in expected lines. Notice that in rule (*Output*) we may introduce the choice type instead of the message prefix type. ∎

Lemma A.3.5
Let T_1 *be a type such that* $T_1 \equiv T_1' \mid \oplus_{i \in I}\{M_i.B_i\}$. *If there is* T_2 *such that* $T_1 <: T_2$ *then there is* T_2' *and* B *and* B_i' *(for each* $i \in I$*) such that* $T_2 \equiv T_2' \mid B \mid \oplus_{i \in I}\{M_i.B_i'\}$ *and* $T_1' <: T_2'$ *and* $\oplus_{i \in I}\{M_i.B_i\} <: B \mid \oplus_{i \in I}\{M_i.B_i'\}$. *Furthermore if* $j \in I$ *and* $P :: T \mid B_j$ *then* $P :: T \mid B \mid B_j'$.

Proof. By induction on the length of the derivation of $T_1 <: T_2$. The proof of the first part of the lemma follows expected lines. We prove that given $\oplus_{i \in I}\{M_i.B_i\} <: B \mid \oplus_{i \in I}\{M_i.B_i'\}$ if $j \in I$ and $P :: T \mid B_j$ then $P :: T \mid B \mid B_j'$, considering the cases of rules (*SubParPref*) and (*SubProject*).

(*Rule* (*SubParPref*))

We have that $\oplus_{i \in I}\{M_i.B_i\} <: B \mid \oplus_{i \in I}\{M_i.B_i'\}$ derived using $M.(B_1 \mid B_2) <: M.B_1 \mid B_2$. This means that $\oplus_{i \in I}\{M_i.B_i\} \equiv M.(B_1 \mid B_2)$ and $B \mid \oplus_{i \in I}\{M_i.B_i'\} \equiv B_2 \mid M.B_1$, and thus the result is direct since $P :: T \mid B_1 \mid B_2$ directly gives us that $P :: T \mid B_2 \mid B_1$.

(*Rule* (*SubProject*))

We have that $\oplus_{i \in I}\{M_i.B_i\} <: B \mid \oplus_{i \in I}\{M_i.B_i'\}$ derived using $B <: \downarrow B \mid \uparrow B$. Let us assume that the M_i messages are defined on the $\downarrow$ direction (the proof for the $\uparrow$ case is analogous). We then have that $B \equiv \uparrow (\oplus_{i \in I}\{M_i.B_i\})$ and $\oplus_{i \in I}\{M_i.B_i'\} \equiv \downarrow (\oplus_{i \in I}\{M_i.B_i\})$. From the definition of direction projection (Definition 5.2.2) we have that $B \equiv \oplus_{i \in I}\{M_i'.B_i''\}$ where $\uparrow (B_i) = M_i'.B_i''$ for each $i \in I$. Let us now consider that $P :: T \mid B_j$. Since $B_j <: M_j'.B_j'' \mid B_j'$ (rule (*SubProject*)) we have that $P :: T \mid M_j'.B_j'' \mid B_j'$. Then considering Lemma A.3.4 we have that $P :: T \mid \oplus_{i \in I}\{M_i'.B_i''\} \mid B_j'$ which concludes the proof for this case. ∎

Lemma A.3.6
Let T_1 *be a type such that* $T_1 \equiv T_1' \mid \&_{i \in I}\{M_i.B_i\}$. *If there is* T_2 *such that* $T_1 <: T_2$ *then there is* T_2' *and* B *and* B_i' *(for each* $i \in I$*) such that* $T_2 \equiv T_2' \mid B \mid \&_{i \in I}\{M_i.B_i'\}$ *and* $T_1' <: T_2'$ *and* $\&_{i \in I}\{M_i.B_i\} <: B \mid \&_{i \in I}\{M_i.B_i'\}$. *Furthermore if* $j \in I$ *and* $P :: T \mid B_j$ *then* $P :: T \mid B \mid B_j'$.

Proof. Follows by induction on the length of the derivation of $T_1 <: T_2$, analogously to the proof of Lemma A.3.5. ∎

Lemma A.3.7
Let process P *be such that* $P :: T$. *If* $P \xrightarrow{l^d?(a)} Q$ *then there are* $T', C, B, \tilde{B}$ *such that either* $T \equiv T' \mid \&\{?\, l^d(C).B; \tilde{B}\}$ *or* $T \equiv T' \mid B$ *and* $\tau\, l^d(C) \in Msg(B)$. *Furthermore if* $T \equiv T' \mid \&\{?\, l^d(C).B; \tilde{B}\}$ *and there is* $T'' = (T' \mid B) \bowtie a : C$ *then* $Q :: T''$.

Proof. By induction on the derivation of the transition $P \xrightarrow{l^d?(a)} Q$. We show the cases when the transition results from a input summation, and from within a conversation piece, and from one component of a parallel composition.

(*Case* $\Sigma_{i \in I}\, l_i^d?(x_i).P_i \xrightarrow{l_j^d?(a)} P_j\{x_j/a\}$)

We have that:

$$\Sigma_{i\in I}\ l_i^d?(x_i).P_i :: T \qquad \text{and} \qquad \Sigma_{i\in I}\ l_i^d?(x_i).P_i \xrightarrow{l_j^d?(a)} P_j\{x_j/a\} \tag{A.3.7.1}$$

From (A.3.7.1) and considering rule (*Input*) we have there is $L, C_j, B_j, \tilde{B}$ such that:

$$L \mid \&\{?\, l_j^d(C_j).B_j; \tilde{B}\} <: T \tag{A.3.7.2}$$

and:

$$\Sigma_{i\in I}\ l_i^d?(x_i).P_i :: L \mid \&\{?\, l_j^d(C_j).B_j; \tilde{B}\} \qquad \text{and} \qquad P_j :: L \mid B_j \mid x_j : C_j \tag{A.3.7.3}$$

From (A.3.7.2) and considering Lemma A.3.6 we conclude there are $T', B', B_j', \tilde{B}'$ such that:

$$T \equiv T' \mid B' \mid \&\{?\, l_j^d(C_j).B_j'; \tilde{B}'\} \tag{A.3.7.4}$$

and:

$$L <: T' \qquad \text{and} \qquad \&\{?\, l_j^d(C_j).B_j; \tilde{B}\} <: B' \mid \&\{?\, l_j^d(C_j).B_j'; \tilde{B}'\} \tag{A.3.7.5}$$

Let us now consider there is T'' such that:

$$T'' \equiv (T' \mid B' \mid B_j') \bowtie a : C_j \tag{A.3.7.6}$$

Since $L <: T'$ from (A.3.7.6) we directly have that there is L' such that

$$L' \mid B_j \equiv (L \mid B_j) \bowtie a : C_j \tag{A.3.7.7}$$

Considering Lemma 5.3.4 we then have:

$$P_j\{x_j/a\} :: (L \mid B_j) \bowtie a : C_j \tag{A.3.7.8}$$

And considering Lemma A.3.6 we have:

$$P_j\{x_j/a\} :: (L \mid B' \mid B_j') \bowtie a : C_j \tag{A.3.7.9}$$

From $L <: T'$ we conclude:

$$P_j\{x_j/a\} :: (T' \mid B' \mid B_j') \bowtie a : C_j \tag{A.3.7.10}$$

which completes the proof for this case.

(*Case* $c \blacktriangleleft [P'] \xrightarrow{l^{\downarrow}?(a)} c \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] :: T \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \xrightarrow{l^{\downarrow}?(a)} c \blacktriangleleft \left[Q'\right] \tag{A.3.7.11}$$

From (A.3.7.11) and considering rule (*Piece*) we have there is L, B such that:

$$(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) <: T \tag{A.3.7.12}$$

and:

$$c \blacktriangleleft \left[P'\right] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.7.13}$$

We also have that (A.3.7.11) is derived from:

$$P' \xrightarrow{l^{\uparrow}?(a)} Q' \tag{A.3.7.14}$$

By induction hypothesis on (A.3.7.14) and (A.3.7.13) we have there is $T', C, B', \tilde{B}$ such that:

$$L \mid B \equiv T' \mid \&\{?\, l^{\uparrow}(C).B'; \tilde{B}\} \tag{A.3.7.15}$$

or:

$$\tau\, l^{d}(C) \in Msg(B) \tag{A.3.7.16}$$

In case of (A.3.7.15) we have that there is B_1 such that:

$$B \equiv B_1 \mid \&\{?\, l^{\uparrow}(C).B'; \tilde{B}\} \tag{A.3.7.17}$$

and there is L' such that $T' \equiv L' \mid B_1$ and $L \equiv L'$. We then have:

$$loc(\uparrow B) \equiv loc(\uparrow B_1) \mid \&\{?\, l^{\downarrow}(C).loc(\uparrow B'); loc(\uparrow \tilde{B})\} \tag{A.3.7.18}$$

From (A.3.7.12), (A.3.7.15) and (A.3.7.18) we conclude:

$$(L' \bowtie c : [\downarrow B_1 \mid \ \downarrow (\&\{?\, l^{\uparrow}(C).B'; \tilde{B}\})]) \mid loc(\uparrow B_1) \mid \&\{?\, l^{\downarrow}(C).loc(\uparrow B'); loc(\uparrow \tilde{B})\} <: T \tag{A.3.7.19}$$

From (A.3.7.19) and considering Lemma A.3.6 we have that there is $T_1', B_1', B'', \tilde{B}'$ such that:

$$T \equiv T_1' \mid B_1' \mid \&\{?\, l^{\downarrow}(C).B''; \tilde{B}'\} \tag{A.3.7.20}$$

where:

$$(L' \bowtie c : [\downarrow B_1 \mid \ \downarrow (\&\{?\, l^{\uparrow}(C).B'; \tilde{B}\})]) <: T_1' \tag{A.3.7.21}$$

and:

$$loc(\uparrow B_1) \mid \&\{?\, l^{\downarrow}(C).loc(\uparrow B'); loc(\uparrow \tilde{B})\} <: B_1' \mid \&\{?\, l^{\downarrow}(C).B''; \tilde{B}'\} \tag{A.3.7.22}$$

In case of (A.3.7.16) proof that $T \equiv T' \mid B'$ and $\tau\, l^{d}(C) \in Msg(B')$, from $\tau\, l^{d}(C) \in Msg(B)$ and (A.3.7.12) follows expect lines.

Let us now consider (A.3.7.20) and that there is T_2 such that:

$$T_2 \equiv (T_1' \mid B_1' \mid B'') \bowtie a : C \tag{A.3.7.23}$$

From (A.3.7.23), (A.3.7.21) and (A.3.7.15) we conclude there is T'' such that:

$$T'' \equiv (T' \mid B') \bowtie a : C \tag{A.3.7.24}$$

By induction hypothesis we then have:

$$Q' :: (T' \mid B') \bowtie a : C \tag{A.3.7.25}$$

and thus:

$$Q' :: (L' \mid B_1 \mid B') \bowtie a : C \tag{A.3.7.26}$$

From (A.3.7.26) we derive:

$$c \blacktriangleleft [Q'] :: ((L' \bowtie a : C) \bowtie c : [\downarrow B_1 \mid \ \downarrow B']) \mid loc(\uparrow (B_1 \mid B')) \tag{A.3.7.27}$$

From (A.3.7.21) and (A.3.7.22) and considering Lemma A.3.6 we conclude:

$$c \blacktriangleleft [Q'] :: (T_1' \mid B_1' \mid B'') \bowtie a : C \tag{A.3.7.28}$$

which completes the proof for this case.

($Case\ P' \mid R \xrightarrow{l^d?(a)} Q' \mid R$)

We have that:

$$P' \mid R :: T \qquad \text{and} \qquad P' \mid R \xrightarrow{l^d?(a)} Q' \mid R \tag{A.3.7.29}$$

which is derived from:

$$P' \xrightarrow{l^d?(a)} Q' \tag{A.3.7.30}$$

Considering (A.3.7.29) and rule (Par) we have that there is T_1, T_2 such that $T_1 \bowtie T_2 <: T$ and:

$$P' \mid R :: T_1 \bowtie T_2 \qquad \text{and} \qquad P' :: T_1 \qquad \text{and} \qquad R :: T_2 \tag{A.3.7.31}$$

By induction hypothesis on (A.3.7.30) and (A.3.7.31) we have that there is $T', C, B, \tilde{B}$ such that:

$$T_1 \equiv T' \mid \&\{?\, l^d(C).B; \tilde{B}\} \tag{A.3.7.32}$$

or:

$$T_1 \equiv T' \mid B \qquad \text{and} \qquad \tau\, l^d(C) \in Msg(B) \tag{A.3.7.33}$$

In case of (A.3.7.32), from $T_1 \bowtie T_2 <: T$ we conclude there is T'' such that either:

$$T \equiv T'' \mid \&\{?\, l^d(C).B; \tilde{B}\} \tag{A.3.7.34}$$

or:

$$T \equiv T'' \mid B'' \qquad \text{and} \qquad \tau\, l^d(C) \in Msg(B'') \tag{A.3.7.35}$$

In case of (A.3.7.33), from $T_1 \bowtie T_2 <: T$ we directly have that $T \equiv T'' \mid B''$ where $\tau\, l^d(C) \in Msg(B'')$. Let us now consider (A.3.7.34) and that there is T_3 such that:

$$T_3 \equiv (T'' \mid B) \bowtie a : C \tag{A.3.7.36}$$

From (A.3.7.36), (A.3.7.34) and $T_1 \bowtie T_2 <: T$ we have that there is $(T' \mid B) \bowtie a : C$, and thus by induction hypothesis we conclude:

$$Q' :: (T' \mid B) \bowtie a : C \tag{A.3.7.37}$$

From (A.3.7.37) and (A.3.7.31) we derive:

$$Q' \mid R :: ((T' \mid B) \bowtie a : C) \bowtie T_2 \tag{A.3.7.38}$$

From (A.3.7.38), (A.3.7.34), (A.3.7.32) and $T <: T_1 \bowtie T_2$ we conclude:

$$Q' \mid R :: (T'' \mid B) \bowtie a : C \tag{A.3.7.39}$$

which completes the proof for this case. ∎

Lemma A.3.8

Let T_1 be a type such that $T_1 \equiv T_1' \mid c : [\oplus_{i \in I}\{M_i.B_i\}]$. If there is T_2 such that $T_1 <: T_2$ then there is T_2' and B and B_i' (for each $i \in I$) such that $T_2 \equiv T_2' \mid c : [B \mid \oplus_{i \in I}\{M_i.B_i'\}]$ and $T_1' <: T_2'$ and $\oplus_{i \in I}\{M_i.B_i\} <: B \mid \oplus_{i \in I}\{M_i.B_i'\}$. Furthermore if $j \in I$ and $P :: T \mid c : [B_j]$ then $P :: T \mid c : [B \mid B_j']$.

Proof. By induction on the length of the derivation of $T_1 <: T_2$, following the lines of the proof of Lemma A.3.5. ∎

Lemma A.3.9

Let T_1 be a type such that $T_1 \equiv T_1' \mid c : [\&_{i \in I}\{M_i.B_i\}]$. If there is T_2 such that $T_1 <: T_2$ then there is T_2' and B and B_i' (for each $i \in I$) such that $T_2 \equiv T_2' \mid c : [B \mid \&_{i \in I}\{M_i.B_i'\}]$ and $T_1' <: T_2'$ and $\&_{i \in I}\{M_i.B_i\} <: B \mid \&_{i \in I}\{M_i.B_i'\}$. Furthermore if $j \in I$ and $P :: T \mid c : [B_j]$ then $P :: T \mid c : [B \mid B_j']$.

Proof. By induction on the length of the derivation of $T_1 <: T_2$, following the lines of the proof of Lemma A.3.5. ∎

Lemma A.3.10

Let P be a process such that $P :: T$. If $P \xrightarrow{c\ l^{\downarrow}?(a)} Q$ then there are $T', C, B, \tilde{B}$ such that $T \equiv T' \mid c : [\&\{?\ l^{\downarrow}(C).B; \tilde{B}\}]$ or $T \equiv T' \mid c : [B]$ and $\tau\, l^{\downarrow}(C) \in Msg(B)$. Furthermore if $T \equiv T' \mid c : [\&\{?\ l^{\downarrow}(C).B; \tilde{B}\}]$ and there is $T'' = (T' \mid c : [B]) \bowtie a : C$ then $Q :: T''$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c\ l^{\downarrow}?(a)} Q$, similarly to the proof of Lemma A.3.7. We show the case of $l^{\downarrow}?(a)$ transition originating from a context piece.

(*Case* $c \blacktriangleleft [P'] \xrightarrow{c\ l^{\downarrow}?(a)} c \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] :: T \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \xrightarrow{c\ l^{\downarrow}?(a)} c \blacktriangleleft \left[Q'\right] \tag{A.3.10.1}$$

From (A.3.10.1) and considering rule (*Piece*) we have there is L, B such that:

$$(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) <: T \tag{A.3.10.2}$$

and:

$$c \blacktriangleleft \left[P'\right] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.10.3}$$

We also have that (A.3.10.1) is derived from:

$$P' \xrightarrow{l^{\downarrow}?(a)} Q' \tag{A.3.10.4}$$

Considering Lemma A.3.7, from (A.3.10.4) and (A.3.10.3) we have there is $T', C, B', \tilde{B}$ such that either:

$$L \mid B \equiv T' \mid \&\{?\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.10.5}$$

or:

$$\tau\, l^{\downarrow}(C) \in Msg(B) \tag{A.3.10.6}$$

In case of (A.3.10.5) we have that there is B_1 such that:

$$B \equiv B_1 \mid \&\{?\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.10.7}$$

and there is L' such that $T' \equiv L' \mid B_1$ and $L \equiv L'$. We then have:

$$\downarrow B \equiv \downarrow B_1 \mid \&\{?\, l^{\downarrow}(C). \downarrow B'; \downarrow \tilde{B}\} \tag{A.3.10.8}$$

From (A.3.10.2), (A.3.10.5) and (A.3.10.8) we conclude:

$$(L' \bowtie c : [\downarrow B_1 \mid \&\{?\, l^{\downarrow}(C). \downarrow B'; \downarrow \tilde{B}\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\}) <: T \tag{A.3.10.9}$$

From (A.3.10.9) we have that either:

$$\begin{aligned}&(L' \bowtie c : [\downarrow B_1 \mid \&\{?\, l^{\downarrow}(C). \downarrow B'; \downarrow \tilde{B}\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\}) \\ &\equiv L'' \mid c : [\&\{?\, l^{\downarrow}(C).B''; \tilde{B}'\}] \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\})\end{aligned} \tag{A.3.10.10}$$

or:

$$\begin{aligned}&(L' \bowtie c : [\downarrow B_1 \mid \&\{?\, l^{\downarrow}(C). \downarrow B'; \downarrow \tilde{B}\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\}) \\ &\equiv L'' \mid c : [\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\}] \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\})\end{aligned} \tag{A.3.10.11}$$

Then considering Lemma A.3.9 we have that there is $T_1', B_1', B''', \tilde{B}''$ such that either:

$$T \equiv T_1' \mid c : [B_1' \mid \&\{?\, l^{\downarrow}(C).B'''; \tilde{B}''\}] \tag{A.3.10.12}$$

where:

$$L'' \mid loc(\uparrow B_1) \mid loc(\uparrow (\&\{?\, l^{\downarrow}(C).B'; \tilde{B}\})) <: T_1' \tag{A.3.10.13}$$

and:

$$c : [\&\{?\, l^{\downarrow}(C).B''; \tilde{B}'\}] <: c : [B_1' \mid \&\{?\, l^{\downarrow}(C).B'''; \tilde{B}''\}] \tag{A.3.10.14}$$

or:

$$T \equiv T_1' \mid c : [B_1' \mid \&\{?\, l^{\downarrow}(C).B'''; \tilde{B}''\}] \tag{A.3.10.15}$$

In case of (A.3.10.6) proof that $T \equiv T' \mid c : [B']$ and $\tau\, l^{\downarrow}(C) \in Msg(B')$ from $\tau\, l^{\downarrow}(C) \in Msg(B)$ follows expected lines.

Let us now consider (A.3.10.12) and that there is T_2 such that:

$$T_2 \equiv (T_1' \mid c : [B_1' \mid B''']) \bowtie a : C \tag{A.3.10.16}$$

From (A.3.10.16), (A.3.10.13) and (A.3.10.5) we conclude there is T'' such that:

$$T'' \equiv (T' \mid B') \bowtie a : C \tag{A.3.10.17}$$

From Lemma A.3.7 we then have:

$$Q' :: (T' \mid B') \bowtie a : C \tag{A.3.10.18}$$

and thus:

$$Q' :: (L' \mid B_1 \mid B') \bowtie a : C \tag{A.3.10.19}$$

From (A.3.10.19) we derive:

$$c \blacktriangleleft \left[Q'\right] :: ((L' \bowtie a : C) \bowtie c : [\downarrow B_1 \mid \ \downarrow B']) \mid loc(\uparrow (B_1 \mid B')) \tag{A.3.10.20}$$

From (A.3.10.10), (A.3.10.13) and (A.3.10.14), and considering Lemma A.3.9 we conclude:

$$c \blacktriangleleft \left[Q'\right] :: (T_1' \mid c : [B_1' \mid B''']) \bowtie a : C \tag{A.3.10.21}$$

which completes the proof for this case. ∎

Lemma A.3.11
Let P be a process such that $P :: T$. If $P \xrightarrow{l^d!(a)} Q$ then there are $T', C, B, \tilde{B}$ such that either $T \equiv T' \mid \oplus\{!\, l^d(C).B; \tilde{B}\}$ or $T \equiv T' \mid B$ and $\tau\, l^d(C) \in Msg(B)$. Furthermore if $T \equiv T' \mid \oplus\{!\, l^d(C).B; \tilde{B}\}$ or $l \in \mathcal{L}_\star$ then there is T'' such that $T' = T'' \bowtie a : C$ and $Q :: T'' \mid B$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{l^d!(a)} Q$. We show the cases when the transition results from a output prefix and from within a conversation piece.

(*Case* $l^d!(a).P' \xrightarrow{l^d!(a)} P'$)

We have that:

$$l^d!(a).P' :: T \qquad \text{and} \qquad l^d!(a).P' \xrightarrow{l^d!(a)} P' \tag{A.3.11.1}$$

From (A.3.11.1) and considering rule (*Output*) we have there is $L, C, B, \tilde{B}$ such that:

$$(L \bowtie a : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\} <: T \tag{A.3.11.2}$$

and:

$$l^d!(a).P' :: (L \bowtie a : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\} \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.11.3}$$

From (A.3.11.2) and considering Lemma A.3.5 we have there is $T', B', B'', \tilde{B}'$ such that:

$$T \equiv T' \mid B' \mid \oplus\{!\, l^d(C).B''; \tilde{B}'\} \tag{A.3.11.4}$$

where:

$$L \bowtie a : C <: T' \tag{A.3.11.5}$$

and:

$$\oplus\{!\, l^d(C).B; \tilde{B}\} <: B' \mid \oplus\{!\, l^d(C).B''; \tilde{B}'\} \tag{A.3.11.6}$$

From (A.3.11.5) we have that there is T'' such that $L <: T''$ and:

$$T' \equiv T'' \bowtie a : C \tag{A.3.11.7}$$

From (A.3.11.3) and (A.3.11.6) and considering Lemma A.3.5 we conclude:

$$P' :: L \mid B' \mid B'' \tag{A.3.11.8}$$

From $L <: T''$ we then have:

$$P' :: T'' \mid B' \mid B'' \tag{A.3.11.9}$$

which completes the proof for this case.

($Case\ c \blacktriangleleft [P'] \xrightarrow{l^{\downarrow}!(a)} c \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] :: T \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \xrightarrow{l^{\downarrow}!(a)} c \blacktriangleleft \left[Q'\right] \tag{A.3.11.10}$$

From (A.3.11.10) and considering rule (*Piece*) we have there is L, B such that:

$$(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) <: T \tag{A.3.11.11}$$

and:

$$c \blacktriangleleft \left[P'\right] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.11.12}$$

We also have that (A.3.11.10) is derived from:

$$P' \xrightarrow{l^{\uparrow}!(a)} Q' \tag{A.3.11.13}$$

By induction hypothesis on (A.3.11.13) and (A.3.11.12) we have there is $T', C, B', \tilde{B}$ such that:

$$L \mid B \equiv T' \mid \oplus\{!\, l^{\uparrow}(C).B'; \tilde{B}\} \tag{A.3.11.14}$$

or:

$$\tau\, l^{\uparrow}(C) \in Msg(B) \tag{A.3.11.15}$$

We show the case of (A.3.11.14). We have that there is B_1 such that:

$$B \equiv B_1 \mid \oplus\{!\, l^{\uparrow}(C).B'; \tilde{B}\} \tag{A.3.11.16}$$

and there is L' such that $T' \equiv L' \mid B_1$ and $L \equiv L'$. From (A.3.11.16) we have that:

$$loc(\uparrow B) \equiv loc(\uparrow B_1) \mid \oplus\{!\, l^{\downarrow}(C).loc(\uparrow B'); loc(\uparrow \tilde{B})\} \tag{A.3.11.17}$$

and:

$$\downarrow B \equiv \downarrow B_1 \mid \downarrow (\oplus\{!\, l^{\uparrow}(C).B'; \tilde{B}\}) \tag{A.3.11.18}$$

From (A.3.11.11) and (A.3.11.17) and (A.3.11.18) and considering Lemma A.3.5 we conclude there are $T_1', B'', B''', \tilde{B}'$ such that:

$$T \equiv T_1' \mid B'' \mid \oplus\{!\, l^{\downarrow}(C).B'''; \tilde{B}'\} \tag{A.3.11.19}$$

where:

$$(L \bowtie c : [\downarrow B_1 \mid \downarrow (\oplus\{ !\, l^{\uparrow}(C).B' ; \tilde{B}\})]) \mid loc(\uparrow B_1) <: T'_1 \tag{A.3.11.20}$$

and:

$$\oplus\{ !\, l^{\downarrow}(C).loc(\uparrow B') ; loc(\uparrow \tilde{B})\} <: B'' \mid \oplus\{ !\, l^{\downarrow}(C).B''' ; \tilde{B}'\} \tag{A.3.11.21}$$

Let us now consider (A.3.11.19) (proof when $l \in \mathcal{L}_\star$ follows similar lines). We then have (A.3.11.14) and by induction hypothesis we conclude there is T'' such that:

$$T' \equiv T'' \bowtie a : C \qquad \text{and} \qquad Q' :: T'' \mid B' \tag{A.3.11.22}$$

From (A.3.11.22) and $T' \equiv L' \mid B_1$ we have that there is L'' such that:

$$T'' \equiv L'' \mid B_1 \qquad \text{and} \qquad L' \equiv L'' \bowtie a : C \tag{A.3.11.23}$$

From (A.3.11.20), (A.3.11.22) and (A.3.11.14) we conclude there is T''_1 such that:

$$T'_1 \equiv T''_1 \bowtie a : C \tag{A.3.11.24}$$

and:

$$(L'' \bowtie c : [\downarrow B_1 \mid \downarrow (\oplus\{ !\, l^{\uparrow}(C).B' ; \tilde{B}\})]) \mid loc(\uparrow B_1) <: T''_1 \tag{A.3.11.25}$$

From (A.3.11.22) and (A.3.11.23) we conclude:

$$Q' :: L'' \mid B_1 \mid B' \tag{A.3.11.26}$$

from which we derive:

$$c \blacktriangleleft \left[Q'\right] :: (L'' \bowtie c : [\downarrow (B_1 \mid B')]) \mid loc(\uparrow B_1) \mid loc(\uparrow B') \tag{A.3.11.27}$$

Then, considering (A.3.11.25) and Lemma A.3.4 we have:

$$c \blacktriangleleft \left[Q'\right] :: T''_1 \mid loc(\uparrow B') \tag{A.3.11.28}$$

which together with (A.3.11.21) and considering Lemma A.3.5 leads to:

$$c \blacktriangleleft \left[Q'\right] :: T''_1 \mid B'' \mid B''' \tag{A.3.11.29}$$

which completes the proof for this case. ■

Lemma A.3.12

Let P be a process such that $P :: T$. If $P \xrightarrow{c\ l^{\downarrow}!(a)} Q$ then there are $T', C, B, \tilde{B}$ such that $T \equiv T' \mid c : [\oplus\{ !\, l^{\downarrow}(C).B ; \tilde{B}\}]$ or $T \equiv T' \mid c : [B]$ and $\tau\, l^{\downarrow}(C) \in Msg(B)$. Furthermore if $T \equiv T' \mid c : [\oplus\{ !\, l^{\downarrow}(C).B ; \tilde{B}\}]$ or $l \in \mathcal{L}_\star$ then there are T'' such that $T' \equiv T'' \bowtie a : C$ and $Q :: T'' \mid c : [B]$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c\ l^{\downarrow}!(a)} Q$, following similar lines of the proof of Lemma A.3.11 and of Lemma A.3.10. We show the case when the transition originates from within a conversation piece.

(*Case* $c \blacktriangleleft [P'] \xrightarrow{c\ l^{\downarrow}!(a)} c \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] :: T \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \xrightarrow{c\ l^{\downarrow}!(a)} c \blacktriangleleft \left[Q'\right] \tag{A.3.12.1}$$

From (A.3.12.1) and considering rule (*Piece*) we have there is L, B such that:

$$(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) <: T \tag{A.3.12.2}$$

and:

$$c \blacktriangleleft \left[P'\right] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.12.3}$$

We also have that (A.3.12.1) is derived from:

$$P' \xrightarrow{l^{\downarrow}!(a)} Q' \tag{A.3.12.4}$$

By induction hypothesis on (A.3.12.4) and (A.3.12.3) we have there is $T', C, B', \tilde{B}$ such that:

$$L \mid B \equiv T' \mid \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.12.5}$$

or:

$$\tau\, l^{\downarrow}(C) \in Msg(B) \tag{A.3.12.6}$$

We show the case of (A.3.12.5). We have that there is B_1 such that:

$$B \equiv B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.12.7}$$

and there is L' such that $T' \equiv L' \mid B_1$ and $L \equiv L'$. From (A.3.12.7) we have that:

$$loc(\uparrow B) \equiv loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) \tag{A.3.12.8}$$

and:

$$\downarrow B \equiv \downarrow B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\} \tag{A.3.12.9}$$

From (A.3.12.2) and (A.3.12.8) and (A.3.12.9) we have:

$$(L \bowtie c : [\downarrow B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) <: T \tag{A.3.12.10}$$

From (A.3.12.10) we conclude that either:

$$\begin{array}{l}(L \bowtie c : [\downarrow B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) \\ \equiv \\ L'' \mid c : [\oplus\{\,!\, l^{\downarrow}(C).B''; \tilde{B}'\}] \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\})\end{array} \tag{A.3.12.11}$$

or:

$$\begin{array}{l}(L \bowtie c : [\downarrow B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\}]) \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) \\ \equiv \\ L'' \mid c : [\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\}] \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\})\end{array} \tag{A.3.12.12}$$

Then considering Lemma A.3.8 we conclude there are $T_1', B_2, B_2', \tilde{B}''$ such that either:

$$T \equiv T_1' \mid c : [B_2' \mid \oplus\{\,!\, l^{\downarrow}(C).B_2; \tilde{B}''\}] \tag{A.3.12.13}$$

where:

$$L'' \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) <: T_1' \tag{A.3.12.14}$$

and:

$$\oplus\{\,!\, l^{\downarrow}(C).B''; \tilde{B}'\} <: B_2' \mid \oplus\{\,!\, l^{\downarrow}(C).B_2; \tilde{B}'\} \tag{A.3.12.15}$$

or:

$$T \equiv T_1' \mid c : [B_2' \mid \oplus\{\tau\, l^{\downarrow}(C).B_2; \tilde{B}''\}] \tag{A.3.12.16}$$

where:

$$L'' \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) <: T_1' \tag{A.3.12.17}$$

and:

$$\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\} <: B_2' \mid \oplus\{\tau\, l^{\downarrow}(C).B_2; \tilde{B}'\} \tag{A.3.12.18}$$

Let us now consider (A.3.12.16) and that $l \in \mathcal{L}_\star$. We then have (A.3.12.5) and by induction hypothesis we conclude there is T'' such that:

$$T' \equiv T'' \bowtie a : C \qquad \text{and} \qquad Q' :: T'' \mid B' \tag{A.3.12.19}$$

From (A.3.12.19) and $T' \equiv L' \mid B_1$ we have that there is L''' such that:

$$T'' \equiv L''' \mid B_1 \qquad \text{and} \qquad L' \equiv L''' \bowtie a : C \tag{A.3.12.20}$$

From (A.3.12.17), (A.3.12.19) and (A.3.12.5) we conclude there is T_1'' such that:

$$T_1' \equiv T_1'' \bowtie a : C \tag{A.3.12.21}$$

and:

$$L''' \mid loc(\uparrow B_1) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C).B'; \tilde{B}\}) <: T_1'' \tag{A.3.12.22}$$

From (A.3.12.19) and (A.3.12.20) we conclude:

$$Q' :: L''' \mid B_1 \mid B' \tag{A.3.12.23}$$

From (A.3.12.12) we have:

$$L \bowtie c : [\downarrow B_1 \mid \oplus\{\,!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\}] \equiv L'' \mid c : [\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\}] \tag{A.3.12.24}$$

Since $l \in \mathcal{L}_\star$ we have that there is L_1 such that:

$$L \equiv L_1 \mid c : [\star ?\, l^{\downarrow}(C)] \tag{A.3.12.25}$$

and:

$$(L_1 \bowtie c : [\downarrow B_1]) \bowtie (c : [\oplus\{!\, l^{\downarrow}(C).(\downarrow B'); (\downarrow \tilde{B})\} \bowtie \star?\, l^{\downarrow}(C)]) \equiv L'' \mid c : [\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\}] \tag{A.3.12.26}$$

which gives us that there is $L \bowtie c : [\downarrow B']$, since either $B'' = (\downarrow B')\{!\ l^{\downarrow}(C)/\tau\ l^{\downarrow}(C)\}$ or B'' is the result of applying rule (*Shuffle*) with some behavior specified in $L_1 \bowtie c : [\downarrow B_1]$. We then have, from (A.3.12.23) and (A.3.12.20) that:

$$c \blacktriangleleft \left[Q'\right] :: (L''' \bowtie c : [\downarrow (B_1 \mid B')]) \mid loc(\uparrow B_1) \mid loc(\uparrow B') \tag{A.3.12.27}$$

which gives us there is L_2 such that $L'' \equiv L_2 \bowtie a : C$ and:

$$c \blacktriangleleft \left[Q'\right] :: L_2 \mid c : [B''] \mid loc(\uparrow B_1) \mid loc(\uparrow B') \tag{A.3.12.28}$$

From (A.3.12.17) and (A.3.12.21) and considering Lemma A.3.4 we conclude:

$$c \blacktriangleleft \left[Q'\right] :: T_1'' \mid c : [B''] \tag{A.3.12.29}$$

which together with (A.3.12.18) and considering Lemma A.3.8 leads to:

$$c \blacktriangleleft \left[Q'\right] :: T_1'' \mid c : [B_2' \mid B_2] \tag{A.3.12.30}$$

which completes the proof for this case. ∎

Lemma A.3.13
Let P be a process such that $P :: T$. If $P \xrightarrow{(\nu a)l^d!(a)} Q$ then there are $T', C, B, \tilde{B}$ such that $T \equiv T' \mid \oplus\{!\ l^d(C).B; \tilde{B}\}$ or $T \equiv T' \mid B$ and $\tau\, l^d(C) \in Msg(B)$. Also if $T \equiv T' \mid \oplus\{!\ l^d(C).B; \tilde{B}\}$ or $l \in \mathcal{L}_\star$ then there are B', C' such that $closed(B')$, $a : [B'] = a : C' \bowtie a : C$ and $Q :: T' \mid B \mid a : C'$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{(\nu a)l^d!(a)} Q$. The proof follows similar lines to that of Lemma A.3.11. We show the case of (*Open*) (Figure 4.2).

(*Case* $(\boldsymbol{\nu} a)P' \xrightarrow{(\nu a)l^d!(a)} Q'$)

We have that:

$$(\boldsymbol{\nu} a)P' :: T \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' \xrightarrow{(\nu a)l^d!(a)} Q' \tag{A.3.13.1}$$

which is derived from:

$$P' \xrightarrow{l^d!(a)} Q' \tag{A.3.13.2}$$

From (A.3.13.1) and considering rule (*Res*) we have that there is T', B such that:

$$T' <: T \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' :: T' \qquad \text{and} \qquad P' :: T' \mid a : [B] \tag{A.3.13.3}$$

and $closed(B)$. Considering Lemma A.3.11 and (A.3.13.2) and (A.3.13.3) we have that there are $T_1, C, B', \tilde{B}$ such that either:

$$T' \mid a : [B] \equiv T_1 \mid \oplus\{!\ l^d(C).B'; \tilde{B}\} \tag{A.3.13.4}$$

or:

$$T' \mid a : [B] \equiv T_1 \mid B' \qquad \text{and} \qquad \tau\, l^d(C) \in \mathit{Msg}(B') \tag{A.3.13.5}$$

We show the case of (A.3.13.4). We have that there is T_1' such that $T_1 \equiv T_1' \mid a : [B]$ and:

$$T' \equiv T_1' \mid \oplus\{ !\, l^d(C).B'; \tilde{B}\} \tag{A.3.13.6}$$

From (A.3.13.6) and $T' <: T$ and considering Lemma A.3.5 we conclude there is $T'', B'', B''', \tilde{B}'$ such that:

$$T \equiv T'' \mid B'' \mid \oplus\{ !\, l^d(C).B'''; \tilde{B}'\} \tag{A.3.13.7}$$

where:

$$T_1' <: T'' \tag{A.3.13.8}$$

and:

$$\oplus\{ !\, l^d(C).B'; \tilde{B}\} <: B'' \mid \oplus\{ !\, l^d(C).B'''; \tilde{B}'\} \tag{A.3.13.9}$$

Let us now consider (A.3.13.6) (proof when $l \in \mathcal{L}_\star$ follows similar lines). We then have that must be the case of (A.3.13.4), hence, from Lemma A.3.11 we conclude:

$$T_1 \equiv T_1'' \bowtie a : C \qquad \text{and} \qquad Q' :: T_1'' \mid B' \tag{A.3.13.10}$$

From $T_1 \equiv T_1' \mid a : [B]$ we then have:

$$T_1' \mid a : [B] \equiv T_1'' \bowtie a : C \tag{A.3.13.11}$$

From (A.3.13.11) we conclude there are T_2, C' such that $T_2 \equiv T_1'$:

$$T_1'' \equiv T_2 \mid a : C' \qquad \text{and} \qquad a : [B] = a : C' \bowtie a : C \tag{A.3.13.12}$$

From (A.3.13.10) and (A.3.13.12) and $T_2 \equiv T_1'$ we have:

$$Q' :: T_1' \mid a : C' \mid B' \tag{A.3.13.13}$$

From (A.3.13.13) and $T_1' <: T''$ — (A.3.13.8) — we then have:

$$Q' :: T'' \mid a : C' \mid B' \tag{A.3.13.14}$$

Finally, from (A.3.13.14) and (A.3.13.9) and Lemma A.3.5 we conclude:

$$Q' :: T'' \mid a : C' \mid B'' \mid B''' \tag{A.3.13.15}$$

which completes the proof for this case. ■

Lemma A.3.14

Let P be a process such that $P :: T$. If $P \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q$ then there are $T', C, B, \tilde{B}$ such that $T \equiv T' \mid c : [\oplus\{ !\, l^{\downarrow}(C).B; \tilde{B}\}]$ or $T \equiv T' \mid c : [B]$ and $\tau\, l^{\downarrow}(C).B \in \mathit{Msg}(B)$. Furthermore if $T \equiv T' \mid c : [\oplus\{ !\, l^{\downarrow}(C).B; \tilde{B}\}]$ or $l \in \mathcal{L}_\star$ then there are B', C' such that $\mathit{closed}(B')$ and $a : [B'] = a : C' \bowtie a : C$ and $Q :: (T' \bowtie c : [B]) \mid a : C'$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q$. We show the case of a transition originating from within a context piece.

(*Case* $c \blacktriangleleft [P'] \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} c \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] :: T \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} c \blacktriangleleft \left[Q'\right] \tag{A.3.14.1}$$

From (A.3.14.1) and considering rule (*Piece*) we have there is L, B such that:

$$(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) <: T \tag{A.3.14.2}$$

and:

$$c \blacktriangleleft \left[P'\right] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \qquad \text{and} \qquad P' :: L \mid B \tag{A.3.14.3}$$

We also have that (A.3.14.1) is derived from:

$$P' \xrightarrow{(\nu a) l^{\downarrow}!(a)} Q' \tag{A.3.14.4}$$

Considering Lemma A.3.13, from (A.3.14.4) and (A.3.14.3) we have there is $T', C, B', \tilde{B}$ such that either:

$$L \mid B \equiv T' \mid \oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.14.5}$$

or:

$$\tau\, l^{\downarrow}(C) \in Msg(B) \tag{A.3.14.6}$$

We show the case of (A.3.14.5). We have that there is B_1 such that:

$$B \equiv B_1 \mid \oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\} \tag{A.3.14.7}$$

and there is L' such that $T' \equiv L' \mid B_1$ and $L \equiv L'$. From (A.3.14.7) we have that:

$$loc(\uparrow B) \equiv loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\})) \tag{A.3.14.8}$$

and:

$$\downarrow B \equiv \downarrow B_1 \mid \oplus\{!\, l^{\downarrow}(C).(\downarrow B'); \downarrow \tilde{B}\} \tag{A.3.14.9}$$

From (A.3.14.2), (A.3.14.8) and (A.3.14.9) we conclude:

$$(L' \bowtie c : [\downarrow B_1 \mid \oplus\{!\, l^{\downarrow}(C).(\downarrow B'); \downarrow \tilde{B}\}) \mid loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\})) <: T \tag{A.3.14.10}$$

From (A.3.14.10) we have that either:

$$\begin{aligned}&(L' \bowtie c : [\downarrow B_1 \mid \oplus\{!\, l^{\downarrow}(C).(\downarrow B'); \downarrow \tilde{B}\}) \mid loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\}))\\ &\equiv L'' \mid c : [\oplus\{!\, l^{\downarrow}(C).B''; \tilde{B}'\}) \mid loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\}))\end{aligned} \tag{A.3.14.11}$$

or:

$$\begin{aligned}&(L' \bowtie c : [\downarrow B_1 \mid \oplus\{!\, l^{\downarrow}(C).(\downarrow B'); \downarrow \tilde{B}\}) \mid loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\}))\\ &\equiv L'' \mid c : [\oplus\{\tau\, l^{\downarrow}(C).B''; \tilde{B}'\}) \mid loc(\uparrow B_1) \mid loc(\uparrow (\oplus\{!\, l^{\downarrow}(C).B'; \tilde{B}\}))\end{aligned} \tag{A.3.14.12}$$

Then considering Lemma A.3.8 we conclude there are $T_1', B_1', B''', \tilde{B}''$ such that either:

$$T \equiv T_1' \mid c : [B_1' \mid \oplus\{ !\, l^{\downarrow}(C).B'''; \tilde{B}''\})] \tag{A.3.14.13}$$

where:

$$L'' \mid \mathit{loc}(\uparrow B_1) \mid \mathit{loc}(\uparrow (\oplus\{ !\, l^{\downarrow}(C).B'; \tilde{B}\})) <: T_1' \tag{A.3.14.14}$$

and:

$$c : [\oplus\{ !\, l^{\downarrow}(C).B''; \tilde{B}'\}) <: c : [B_1' \mid \oplus\{ !\, l^{\downarrow}(C).B'''; \tilde{B}''\})] \tag{A.3.14.15}$$

or:

$$T \equiv T_1' \mid c : [B_1' \mid \oplus\{ \tau\, l^{\downarrow}(C).B'''; \tilde{B}''\})] \tag{A.3.14.16}$$

Let us now consider that (A.3.14.13) (proof when $l \in \mathcal{L}_\star$ follows similar lines). We then have (A.3.14.5) and by Lemma A.3.13 we conclude there are B'', C'' such that $\mathit{closed}(B'')$ and:

$$a : \left[B''\right] = a : C' \bowtie a : C \qquad \text{and} \qquad Q' :: T' \mid B' \mid a : C' \tag{A.3.14.17}$$

From $T' \equiv L \mid B_1$ we then have:

$$Q' :: L \mid B_1 \mid B' \mid a : C' \tag{A.3.14.18}$$

We then derive:

$$c \blacktriangleleft \left[Q'\right] :: ((L \mid a : C') \bowtie c : \left[\downarrow (B_1 \mid B')\right]) \mid \mathit{loc}(\uparrow (B_1 \mid B')) \tag{A.3.14.19}$$

From (A.3.14.19) and (A.3.14.11), we conclude:

$$c \blacktriangleleft \left[Q'\right] :: (L'' \mid c : \left[B''\right] \mid a : C') \mid \mathit{loc}(\uparrow (B_1 \mid B')) \tag{A.3.14.20}$$

Then from (A.3.14.14) and considering Lemma A.3.4 we have that:

$$c \blacktriangleleft \left[Q'\right] :: T_1' \mid c : \left[B''\right] \mid a : C' \tag{A.3.14.21}$$

From (A.3.14.21) and (A.3.14.15) and considering Lemma A.3.8 we conclude:

$$c \blacktriangleleft \left[Q'\right] :: T_1' \mid c : \left[B_1' \mid B'''\right] \mid a : C' \tag{A.3.14.22}$$

which completes the proof for this case. ∎

Lemma A.3.15

Let P be a well-typed process such that $P :: T$. If $P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q$ due to a **this** *prefix, then there are L, B_1, B_2 such that $T \equiv L \mid (B_1 \bowtie (\downarrow B_2))$. Furthermore if there is T' such that $T' \equiv (L \mid B_1) \bowtie (c : [\downarrow B_2])$ then $Q :: (L \mid B_1) \bowtie (c : [\downarrow B_2])$.*

Proof. By induction on the derivation of the transition $P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q$. We show the case of a **this** prefix transition.

(*Case* $\mathbf{this}(x).P' \xrightarrow{c\ \mathtt{this}^{\downarrow}} P'\{x/c\}$)

We have that:

$$\textsf{\textbf{this}}(x).P' \xrightarrow{c\ \texttt{this}^{\downarrow}} P'\{x/c\} \qquad \text{and} \qquad \textsf{\textbf{this}}(x).P' :: T \tag{A.3.15.1}$$

From (A.3.15.1) and considering rule (*This*) we have there are L, B_1, B_2 such that:

$$L \mid (B_1 \bowtie B_2) <: T \qquad \text{and} \qquad P' :: L \mid B_1 \mid x : [B_2] \tag{A.3.15.2}$$

We then have that there is L', B_1', B_2' such that $L <: L'$, $B_1 <: B_1'$ and $B_2 <: B_2'$ and:

$$T \equiv L' \mid (B_1' \bowtie B_2') \tag{A.3.15.3}$$

Let us consider there is T' such that:

$$T' = (L' \mid B_1') \bowtie c : [B_2'] \tag{A.3.15.4}$$

which directly gives us that there is T'' such that:

$$T'' = (L \mid B_1) \bowtie c : [B_2] \tag{A.3.15.5}$$

Then considering Lemma 5.3.4 we conclude:

$$P'\{x/c\} :: (L \mid B_1) \bowtie c : [B_2] \tag{A.3.15.6}$$

and hence:

$$P'\{x/c\} :: (L' \mid B_1') \bowtie c : [B_2'] \tag{A.3.15.7}$$

which completes the proof for this case. ■

Lemma A.3.16
Let P be a process such that $P :: T$. If $P \xrightarrow{\texttt{this}^{\downarrow\uparrow}} Q$ then there are $T', B, B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that $T \equiv T' \mid \{\mathsf{p}_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid \{\mathsf{p}_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}$ where $\mathsf{p}_i = \texttt{!}$ and $\mathsf{p}_j = \texttt{?}$ for $\{i, j\} = \{1, 2\}$, or $T \equiv T' \mid B$ and $\mathsf{p}_1\, l^{\uparrow}(C_1) \in \mathit{Msg}(B)$ and $\mathsf{p}_2\, l^{\downarrow}(C_2) \in \mathit{Msg}(B)$. Furthermore if $T \equiv T' \mid \{\mathsf{p}_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid \{\mathsf{p}_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}$ and $\mathsf{p}_i = \texttt{!}$ or $l \in \mathcal{L}_{\star}$ and $\mathsf{p}_j = \texttt{?}$ and $C_1 \equiv C_2$ then we have that $Q :: T' \mid B_1 \mid B_2$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{\texttt{this}^{\downarrow\uparrow}} Q$. We show the case of a (*Comm*) synchronization.

(*Case* $P_1 \mid P_2 \xrightarrow{\texttt{this}^{\downarrow\uparrow}} Q_1 \mid Q_2$)

We have that:

$$P_1 \mid P_2 \xrightarrow{\texttt{this}^{\downarrow\uparrow}} Q_1 \mid Q_2\ (i) \qquad \text{and} \qquad P_1 \mid P_2 :: T\ (ii) \tag{A.3.16.1}$$

(A.3.16.1)(i) is derived from:

$$P_1 \xrightarrow{l^{\downarrow}\texttt{!}(a)} Q_1 \qquad \text{and} \qquad P_2 \xrightarrow{l^{\uparrow}\texttt{?}(a)} Q_2 \tag{A.3.16.2}$$

From (A.3.16.1)(ii) we have there are T_1, T_2 such that:

$$T_1 \bowtie T_2 <: T \qquad \text{and} \qquad P_1 :: T_1 \qquad \text{and} \qquad P_2 :: T_2 \tag{A.3.16.3}$$

Considering Lemma A.3.11, (A.3.16.2) and (A.3.16.3) we have there are $T_1', C_1, B_1, \tilde{B}$ such that either:

$$T_1 \equiv T_1' \mid \oplus\{ !\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \tag{A.3.16.4}$$

or:

$$T_1 \equiv T_1' \mid B_1 \qquad \text{and} \qquad \tau\, l^{\downarrow}(C_1) \in Msg(B_1) \tag{A.3.16.5}$$

Considering Lemma A.3.7, (A.3.16.2) and (A.3.16.3) we have there are $T_2', C_2, B_2, \tilde{B}'$ such that either:

$$T_2 \equiv T_2' \mid \&\{ ?\, l^{\uparrow}(C_2).B_2; \tilde{B}'\} \tag{A.3.16.6}$$

or:

$$T_2 \equiv T_2' \mid B_2 \qquad \text{and} \qquad \tau\, l^{\uparrow}(C_2) \in Msg(B_2) \tag{A.3.16.7}$$

From $T_1 \bowtie T_2 <: T$ and (A.3.16.4), (A.3.16.5), (A.3.16.6) and (A.3.16.7) we directly have that $T \equiv T' \mid B_1$ such that $p_1\, l^{\downarrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\uparrow}(C_2) \in Msg(B_1)$.

Let us now consider it is the case of (A.3.16.4) and (A.3.16.6) when $!\, l^{\downarrow}(C_1) \# T_2$ and $?\, l^{\downarrow}(C_2) \# T_1$. Considering Lemma A.3.5 and Lemma A.3.6 we then have:

$$\begin{array}{rcl} T & \equiv & (T_1'' \mid B_1' \mid \oplus\{ !\, l^{\downarrow}(C_1).B_1''; \tilde{B}''\}) \bowtie (T_2'' \mid B_2' \mid \&\{ ?\, l^{\uparrow}(C_2).B_2''; \tilde{B}'''\}) \\ & \equiv & T' \mid \oplus\{ !\, l^{\downarrow}(C_1).B'; (\cdots)\} \mid \&\{ ?\, l^{\uparrow}(C_2).B''; (\cdots)\} \end{array} \tag{A.3.16.8}$$

where $T_1' <: T_1''$ and $T_2' <: T_2''$ and:

$$\oplus\{ !\, l^{\downarrow}(C_1).B_1; \tilde{B}\} <: B_1' \mid \oplus\{ !\, l^{\downarrow}(C_1).B_1''; \tilde{B}''\} \tag{A.3.16.9}$$

and:

$$\&\{ ?\, l^{\uparrow}(C_2).B_2; \tilde{B}'\} <: B_2' \mid \&\{ ?\, l^{\uparrow}(C_2).B_2''; \tilde{B}'''\} \tag{A.3.16.10}$$

Let us now consider that $(C \equiv) C_1 \equiv C_2$. From Lemma A.3.11 we have there is T_1''' such that:

$$T_1' = T_1''' \bowtie a : C \qquad \text{and} \qquad Q_1 :: T_1''' \mid B_1 \tag{A.3.16.11}$$

Then, via Lemma A.3.7, considering (A.3.16.11) and $T_1' <: T_1''$ and (A.3.16.8) and $T_2' <: T_2''$ we conclude that there is T_2''' such that:

$$T_2''' = (T_2' \mid B_2) \bowtie a : C \qquad \text{and} \qquad Q_2 :: T_2''' \tag{A.3.16.12}$$

From (A.3.16.11), (A.3.16.9) and considering Lemma A.3.5 we conclude:

$$Q_1 :: T_1'''' \mid B_1' \mid B_1'' \tag{A.3.16.13}$$

Likewise from (A.3.16.12), (A.3.16.10) and considering Lemma A.3.6 we conclude:

$$Q_2 :: (T_2' \bowtie a : C) \mid B_2' \mid B_2'' \tag{A.3.16.14}$$

Then from (A.3.16.13) and (A.3.16.14) we conclude:

$$Q_1 \mid Q_2 :: (T_1''' \mid B_1' \mid B_1'') \bowtie ((T_2' \bowtie a : C) \mid B_2' \mid B_2'') \quad \text{(A.3.16.15)}$$

which then leads to:

$$Q_1 \mid Q_2 :: (T_1' \mid B_1' \mid B_1'') \bowtie (T_2' \mid B_2' \mid B_2'') \quad \text{(A.3.16.16)}$$

Then, since $T_1' <: T_1''$ and $T_2' <: T_2''$ we derive:

$$Q_1 \mid Q_2 :: (T_1'' \mid B_1' \mid B_1'') \bowtie (T_2'' \mid B_2' \mid B_2'') \quad \text{(A.3.16.17)}$$

which, along with (A.3.16.8), gives us:

$$Q_1 \mid Q_2 :: T' \mid B' \mid B'' \quad \text{(A.3.16.18)}$$

which completes the proof for this case. ∎

Lemma A.3.17
Let P be a process such that $P :: T$. If $P \stackrel{c\ \mathtt{this}^{\uparrow}}{\longrightarrow} Q$ then there are $T', B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that $T \equiv T' \mid \{p_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$ where $p_i = \mathtt{!}$ and $p_j = \mathtt{?}$ for $\{i, j\} = \{1, 2\}$, or $T \equiv T' \mid B_1 \mid c : [B_2]$ and $p_1\, l^{\uparrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\downarrow}(C_2) \in Msg(B_2)$. Furthermore if $T \equiv T' \mid \{p_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$ and $p_i = \mathtt{!}$ or $l \in \mathcal{L}_\star$ and $p_j = \mathtt{?}$ and $C_1 \equiv C_2$ then we have that $Q :: T' \mid B_1 \mid c : [B_2]$.

Proof. By induction on the length of the derivation of the transition $P \stackrel{c\ \mathtt{this}^{\uparrow}}{\longrightarrow} Q$. We show the case of a (*Comm*) synchronization.

(*Case* $P_1 \mid P_2 \stackrel{c\ \mathtt{this}^{\uparrow}}{\longrightarrow} Q_1 \mid Q_2$)

We have that:

$$P_1 \mid P_2 \stackrel{c\ \mathtt{this}^{\uparrow}}{\longrightarrow} Q_1 \mid Q_2\ (i) \qquad \text{and} \qquad P_1 \mid P_2 :: T\ (ii) \quad \text{(A.3.17.1)}$$

(A.3.17.1)(i) is derived from:

$$P_1 \stackrel{c\ l^{\downarrow}!(a)}{\longrightarrow} Q_1 \qquad \text{and} \qquad P_2 \stackrel{l^{\uparrow}?(a)}{\longrightarrow} Q_2 \quad \text{(A.3.17.2)}$$

From (A.3.17.1)(ii) we have there are T_1, T_2 such that:

$$T_1 \bowtie T_2 <: T \qquad \text{and} \qquad P_1 :: T_1 \qquad \text{and} \qquad P_2 :: T_2 \quad \text{(A.3.17.3)}$$

Considering Lemma A.3.12, (A.3.17.2) and (A.3.17.3) we have there are $T_1', C_1, B_1, \tilde{B}$ such that either:

$$T_1 \equiv T_1' \mid c : [\oplus\{\mathtt{!}\ l^{\downarrow}(C_1).B_1; \tilde{B}\}] \quad \text{(A.3.17.4)}$$

or:

$$T_1 \equiv T_1' \mid c : [B_1] \qquad \text{and} \qquad \boldsymbol{\tau}\, l^{\downarrow}(C_1) \in Msg(B_1) \quad \text{(A.3.17.5)}$$

Considering Lemma A.3.7, (A.3.17.2) and (A.3.17.3) we have there are $T_2', C_2, B_2, \tilde{B}'$ such that:

$$T_2 \equiv T_2' \mid \&\{?\, l^{\uparrow}(C_2).B_2; \tilde{B}'\} \tag{A.3.17.6}$$

or:

$$T_2 \equiv T_2' \mid B_2 \qquad \text{and} \qquad \tau\, l^{\uparrow}(C_2) \in Msg(B_2) \tag{A.3.17.7}$$

From $T_1 \bowtie T_2 <: T$ and (A.3.17.4), (A.3.17.5), (A.3.17.6) and (A.3.17.7) we directly have that $T \equiv T' \mid c : [B_1] \mid B_2$ such that $p_1\, l^{\downarrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\uparrow}(C_2) \in Msg(B_2)$.

Let us now consider the case of (A.3.17.4) and (A.3.17.6 and also that $!\, l^{\downarrow}(C_1) \# T_2$ and $!\, l^{\uparrow}(C_2) \# T_1$. Considering Lemma A.3.8 and Lemma A.3.6 and Lemma A.3.5 we conclude:

$$\begin{array}{rcl} T & \equiv & (T_1'' \mid c : [B_1' \mid \oplus\{!\, l^{\downarrow}(C_1).B_1''; \tilde{B}''\}]) \bowtie (T_2'' \mid B_2' \mid \&\{?\, l^{\uparrow}(C_2).B_2''; \tilde{B}'''\}) \\ & \equiv & T' \mid c : [\oplus\{!\, l^{\downarrow}(C_1).B'; (\cdots)\}] \mid \&\{?\, l^{\uparrow}(C_2).B''; (\cdots)\} \end{array} \tag{A.3.17.8}$$

where $T_1' <: T_1''$ and $T_2' <: T_2''$ and:

$$\oplus\{!\, l^{\downarrow}(C_1).B_1; \tilde{B}\} <: B_1' \mid \oplus\{!\, l^{\downarrow}(C_1).B_1''; \tilde{B}''\} \tag{A.3.17.9}$$

and:

$$\&\{?\, l^{\uparrow}(C_2).B_2; \tilde{B}'\} <: B_2' \mid \&\{?\, l^{\uparrow}(C_2).B_2''; \tilde{B}'''\} \tag{A.3.17.10}$$

Let us now consider $(C \equiv) C_1 \equiv C_2$. Considering Lemma A.3.12 we have there is T_1''' such that:

$$T_1' \equiv T_1''' \bowtie a : C \qquad \text{and} \qquad Q_1 :: T_1''' \mid c : [B_1] \tag{A.3.17.11}$$

Then, via Lemma A.3.7, considering (A.3.17.11) and $T_1' <: T_1''$ and (A.3.17.8) and $T_2' <: T_2''$ we conclude that there is T_2''' such that:

$$T_2''' = (T_2' \mid B_2) \bowtie a : C \qquad \text{and} \qquad Q_2 :: T_2''' \tag{A.3.17.12}$$

From (A.3.17.11), (A.3.17.9) and considering Lemma A.3.8 we conclude:

$$Q_1 :: T_1''' \mid c : [B_1' \mid B_1''] \tag{A.3.17.13}$$

Likewise from (A.3.17.12), (A.3.17.10) and considering Lemma A.3.6 we conclude:

$$Q_2 :: (T_2' \mid a : C) \mid B_2' \mid B_2'' \tag{A.3.17.14}$$

Then from (A.3.17.13) and (A.3.17.14) we conclude:

$$Q_1 \mid Q_2 :: (T_1''' \mid c : [B_1' \mid B_1'']) \bowtie ((T_2' \bowtie a : C) \mid B_2' \mid B_2'') \tag{A.3.17.15}$$

which, considering (A.3.17.11) leads to:

$$Q_1 \mid Q_2 :: (T_1' \mid c : [B_1' \mid B_1'']) \bowtie (T_2' \mid B_2' \mid B_2'') \tag{A.3.17.16}$$

Since $T_1' <: T_1''$ and $T_2' <: T_2''$ we derive:

$$Q_1 \mid Q_2 :: (T_1'' \mid c : [B_1' \mid B_1'']) \bowtie (T_2'' \mid B_2' \mid B_2'') \tag{A.3.17.17}$$

which, along with (A.3.17.8), gives us:

$$Q_1 \mid Q_2 :: T' \mid c : [B'] \mid B'' \tag{A.3.17.18}$$

which completes the proof for this case. ∎

Lemma A.3.18
Let P be a process such that $P :: T$. If $P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q$ then there are $T', B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that $T \equiv T' \mid \{p_1\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$ and $p_i = \mathtt{!}$ and $p_j = \mathtt{?}$ where $\{i, j\} = \{1, 2\}$, or $T \equiv T' \mid B_1 \mid c : [B_2]$ and $p_1\, l^{\downarrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\downarrow}(C_2) \in Msg(B_2)$. Furthermore if $T \equiv T' \mid \{p_1\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$ and $p_i = \mathtt{!}$ or $l \in \mathcal{L}_\star$ and $p_j = \mathtt{?}$ and $C_1 \equiv C_2$ then we have $Q :: T' \mid B_1 \mid c : [B_2]$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q$. We show the case of transitions $\mathtt{this}^{\downarrow\uparrow}$ and $c\ \mathtt{this}^{\uparrow}$ originating from within a conversation piece.

(*Case* $n \blacktriangleleft [P'] \xrightarrow{c\ \mathtt{this}^{\downarrow}} n \blacktriangleleft [Q']$)

We have that:

$$n \blacktriangleleft [P'] \xrightarrow{c\ \mathtt{this}^{\downarrow}} n \blacktriangleleft [Q']\ (i) \qquad \text{and} \qquad n \blacktriangleleft [P'] :: T\ (ii) \tag{A.3.18.1}$$

(A.3.18.1)(ii) gives us there is L, B such that

$$(L \bowtie n : [\downarrow B]) \mid loc(\uparrow B) <: T \qquad \text{and} \qquad P :: L \mid B \tag{A.3.18.2}$$

(A.3.18.1)(i) is either derived from:

$$P' \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q' \tag{A.3.18.3}$$

or:

$$P' \xrightarrow{c\ \mathtt{this}^{\uparrow}} Q' \tag{A.3.18.4}$$

(*Case* $P' \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q'$) We have that $n = c$. Also, from (A.3.18.2) and considering Lemma A.3.16 we then have that there are $T', B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that:

$$L \mid B \equiv T' \mid \{p_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid \{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\} \tag{A.3.18.5}$$

and $p_i = \mathtt{!}$ and $p_j = \mathtt{?}$, or:

$$L \mid B \equiv T' \mid B' \tag{A.3.18.6}$$

and $p_1\, l^{\uparrow}(C_1) \in Msg(B')$ and $p_2\, l^{\downarrow}(C_2) \in Msg(B')$. Proof that $T \equiv T'' \mid B_1 \mid c : [B_2]$ and $p_1\, l^{\downarrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\downarrow}(C_2) \in Msg(B_2)$ follows expected lines.

Let us consider that $L \mid B \equiv T' \mid \oplus\{\mathtt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid \&\{\mathtt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}$ and also that $c : [\mathtt{?}\, l^{\downarrow}(C_2)] \# T'$. We have there is B' such that:

$$B \equiv B' \mid \oplus\{\mathtt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid \&\{\mathtt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\} \tag{A.3.18.7}$$

and $T' \equiv L \mid B'$. From (A.3.18.7) we conclude that:

$$\downarrow B \equiv \downarrow B' \mid \downarrow (\oplus\{\,!\,l^{\uparrow}(C_1).B_1; \tilde{B}\}) \mid \&\{\,?\,l^{\downarrow}(C_2).(\downarrow B_2); \downarrow \tilde{B}\} \tag{A.3.18.8}$$

and also that:

$$loc(\uparrow B) \equiv loc(\uparrow B') \mid \oplus\{\,!\,l^{\downarrow}(C_1).loc(\uparrow B_1); loc(\uparrow \tilde{B})\} \mid loc(\uparrow (\&\{\,?\,l^{\downarrow}(C_2).B_2; \tilde{B}'\})) \tag{A.3.18.9}$$

From (A.3.18.2) and (A.3.18.8) and (A.3.18.9) we then have:

$$\begin{aligned}&(L \bowtie c : [\downarrow B' \mid \ \mid \downarrow (\oplus\{\,!\,l^{\uparrow}(C_1).B_1; \tilde{B}\}) \mid \&\{\,?\,l^{\downarrow}(C_2).(\downarrow B_2); \downarrow \tilde{B}\}])\\ &\mid loc(\uparrow B') \mid \oplus\{\,!\,l^{\downarrow}(C_1).loc(\uparrow B_1); loc(\uparrow \tilde{B})\} \mid loc(\uparrow (\&\{\,?\,l^{\downarrow}(C_2).B_2; \tilde{B}'\}))\\ &\equiv T' \mid c : [\&\{\,?\,l^{\downarrow}(C_2).B''; \tilde{B}''\}] \mid \oplus\{\,!\,l^{\downarrow}(C_1).B'''; \tilde{B}'''\} <: T\end{aligned} \tag{A.3.18.10}$$

and thus:

$$T \equiv T'' \mid c : [B_3' \mid \&\{\,?\,l^{\downarrow}(C_2).B_3''; (\cdots)\}] \mid B_4' \mid \oplus\{\,!\,l^{\downarrow}(C_1).B_4''; (\cdots)\} \tag{A.3.18.11}$$

where $T' <: T''$ and:

$$\&\{\,?\,l^{\downarrow}(C_2).B''; \tilde{B}''\} <: B_3' \mid \&\{\,?\,l^{\downarrow}(C_2).B_3''; (\cdots)\} \tag{A.3.18.12}$$

and:

$$\oplus\{\,!\,l^{\downarrow}(C_1).B'''; \tilde{B}'''\} <: B_4' \mid \oplus\{\,!\,l^{\downarrow}(C_1).B_4''; (\cdots)\} \tag{A.3.18.13}$$

Let us now consider that $(C \equiv)C_1 \equiv C_2$. From Lemma A.3.16 we conclude:

$$Q' :: T' \mid B_1 \mid B_2 \tag{A.3.18.14}$$

We then have that:

$$Q' :: L \mid B' \mid B_1 \mid B_2 \tag{A.3.18.15}$$

From (A.3.18.15) we derive:

$$c \blacktriangleleft [Q'] :: (L \bowtie c : [\downarrow (B' \mid B_1 \mid B_2)]) \mid loc(\uparrow (B' \mid B_1 \mid B_2)) \tag{A.3.18.16}$$

From (A.3.18.16) and (A.3.18.10) and considering Lemma A.3.4 we have:

$$c \blacktriangleleft [Q'] :: T' \mid c : [B''] \mid B''' \tag{A.3.18.17}$$

From (A.3.18.17) and $T' <: T''$ and (A.3.18.12) and (A.3.18.13) and considering Lemma A.3.9 and Lemma A.3.5 we conclude:

$$c \blacktriangleleft [Q'] :: T'' \mid c : [B_3' \mid B_3''] \mid B_4' \mid B_4'' \tag{A.3.18.18}$$

which completes the proof for this case.
(*Case* $P' \xrightarrow{c\ \mathtt{this}^{\uparrow}} Q'$) From (A.3.18.2) and considering Lemma A.3.17 we then have that there are

$T', B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that:

$$L \mid B \equiv T' \mid \{p_1\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}] \tag{A.3.18.19}$$

and $p_i = \texttt{!}$ and $p_j = \texttt{?}$, or:

$$L \mid B \equiv T' \mid B_1 \mid c : [B_2] \tag{A.3.18.20}$$

and $p_1\, l^{\uparrow}(C_1) \in \mathit{Msg}(B_1)$ and $p_2\, l^{\downarrow}(C_2) \in \mathit{Msg}(B_2)$. Proof that $T \equiv T'' \mid B_1' \mid c : [B_2']$ and $p_1\, l^{\downarrow}(C_1) \in \mathit{Msg}(B_1')$ and $p_2\, l^{\downarrow}(C_2) \in \mathit{Msg}(B_2')$ follows expected lines.

Let us consider the case when $L \mid B \equiv T' \mid \oplus\{\texttt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \mid c : [\&\{\texttt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$. We then have there is B' such that:

$$B \equiv B' \mid \oplus\{\texttt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\} \tag{A.3.18.21}$$

and $T' \equiv L' \mid B'$ and $L \equiv L' \mid c : [\&\{\texttt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]$. From (A.3.18.21) we conclude that:

$$\downarrow B \equiv \downarrow B' \mid \downarrow (\oplus\{\texttt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\}) \tag{A.3.18.22}$$

and also that:

$$\mathit{loc}(\uparrow B) \equiv \ \mathit{loc}(\uparrow B') \mid \oplus\{\texttt{!}\, l^{\downarrow}(C_1).\mathit{loc}(\uparrow B_1); \mathit{loc}(\uparrow \tilde{B})\} \tag{A.3.18.23}$$

From (A.3.18.2) and (A.3.18.22) and (A.3.18.23) we then have:

$$\begin{aligned}&((L' \mid c : [\&\{\texttt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]) \bowtie n : [\downarrow B' \mid \ \mid \downarrow (\oplus\{\texttt{!}\, l^{\uparrow}(C_1).B_1; \tilde{B}\})])\\ &\mid \mathit{loc}(\uparrow B') \mid \oplus\{\texttt{!}\, l^{\downarrow}(C_1).\mathit{loc}(\uparrow B_1); \mathit{loc}(\uparrow \tilde{B})\}\\ &\equiv T' \mid c : [\&\{\texttt{?}\, l^{\downarrow}(C_2).B''; \tilde{B}''\}] \mid \oplus\{\texttt{!}\, l^{\downarrow}(C_1).B'''; \tilde{B}'''\} <: T\end{aligned} \tag{A.3.18.24}$$

and thus:

$$T \equiv T'' \mid c : [B_3' \mid \&\{\texttt{?}\, l^{\downarrow}(C_2).B_3''; (\cdots)\}] \mid B_4' \mid \oplus\{\texttt{!}\, l^{\downarrow}(C_1).B_4''; (\cdots)\} \tag{A.3.18.25}$$

where $T' <: T''$ and:

$$\&\{\texttt{?}\, l^{\downarrow}(C_2).B''; \tilde{B}''\} <: B_3' \mid \&\{\texttt{?}\, l^{\downarrow}(C_2).B_3''; (\cdots)\} \tag{A.3.18.26}$$

and:

$$\oplus\{\texttt{!}\, l^{\downarrow}(C_1).B'''; \tilde{B}'''\} <: B_4' \mid \oplus\{\texttt{!}\, l^{\downarrow}(C_1).B_4''; (\cdots)\} \tag{A.3.18.27}$$

Let us now consider that $(C \equiv) C_1 \equiv C_2$. From Lemma A.3.17 we conclude:

$$Q' :: T' \mid B_1 \mid c : [B_2] \tag{A.3.18.28}$$

We then have that:

$$Q' :: L' \mid B' \mid B_1 \mid c : [B_2] \tag{A.3.18.29}$$

From (A.3.18.29) we derive:

$$c \blacktriangleleft [Q'] :: ((L' \mid c : [B_2]) \bowtie n : [\downarrow (B' \mid B_1)]) \mid \mathit{loc}(\uparrow (B' \mid B_1)) \tag{A.3.18.30}$$

From (A.3.18.30) and (A.3.18.24) and considering Lemma A.3.4 we have:

$$c \blacktriangleleft [Q'] :: T' \mid c : [B''] \mid B''' \tag{A.3.18.31}$$

From (A.3.18.31) and $T' <: T''$ and (A.3.18.26) and (A.3.18.27) and considering Lemma A.3.8 (if $p_2 = \mathtt{!}$ or Lemma A.3.9 if $p_2 = \mathtt{?}$) and Lemma A.3.6 (if $p_1 = \mathtt{?}$ or Lemma A.3.5 if $p_1 = \mathtt{!}$) we conclude:

$$c \blacktriangleleft [Q'] :: T'' \mid c : [B_3' \mid B_3''] \mid B_4' \mid B_4'' \tag{A.3.18.32}$$

which completes the proof for this case. ■

Lemma A.3.19
Let types T_1, T_1', T_2' be such that $T_1' <: T_1$ and $T_1' \to T_2'$. Then we have that there is type T_2 such that T_1 and $P :: T_2'$ implies $P :: T_2$.

Proof. By induction on the derivation of $T_1' <: T_1$, following expected lines. ■

Theorem 5.3.5 (Subject Reduction)
(repetition of the statement in page 122)
Let P be a well-typed process such that $P :: T$. If $P \to Q$ then there is $T \to T'$ such that $Q :: T'$.

Proof. By induction on the derivation of the reduction $P \to Q$. We show the case of a reduction derived from a synchronization on an unlocated — at the level of the current conversation — message, distinguishing between when the message is defined on a plain label and on a shared label, and the case of a τ derived from a $c\ \mathtt{this}^{\downarrow}$ transition originating from a conversation piece.

(*Case unlocated message synchronization*)

We have:

$$P_1 \mid P_2 \xrightarrow{\tau} Q_1 \mid Q_2 \tag{A.3.20.1}$$

derived from:

$$P_1 \xrightarrow{l^{\downarrow}!(a)} Q_1 \ (i) \qquad \text{and} \qquad P_2 \xrightarrow{l^{\downarrow}?(a)} Q_2 \ (ii) \tag{A.3.20.2}$$

Since $P_1 \mid P_2$ is a well-typed process, we have $P_1 \mid P_2 :: T$ for some T such that $T' <: T$ and:

$$P_1 \mid P_2 :: T' \tag{A.3.20.3}$$

where (A.3.20.3) is derived from (rule (Par)):

$$P_1 :: T_1 \ (i) \qquad \text{and} \qquad P_2 :: T_2 \ (ii) \qquad \text{and} \qquad T' = T_1 \bowtie T_2 \ (iii) \tag{A.3.20.4}$$

From (A.3.20.2)(i) and (A.3.20.4)(i) and considering Lemma A.3.11 we conclude that there are $T_1', C_1, B_1, \tilde{B}$ such that either:

$$T_1 \equiv T_1' \mid \oplus\{\mathtt{!}\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \tag{A.3.20.5}$$

or:

$$T_1 \equiv T_1' \mid B \qquad \text{and} \qquad \boldsymbol{\tau}\, l^{\downarrow}(C_1) \in Msg(B_1) \tag{A.3.20.6}$$

From (A.3.20.2)(ii) and (A.3.20.4)(ii), considering Lemma A.3.7, we conclude that there are $T_2', C_2, B_2, \tilde{B}'$ such that either:

$$T_2 \equiv T_2' \mid \&\{?\, l^{\downarrow}(C_2).B_2; \tilde{B}'\} \tag{A.3.20.7}$$

or:

$$T_2 \equiv T_2' \mid B \qquad \text{and} \qquad \tau\, l^{\downarrow}(C_2) \in Msg(B_2) \tag{A.3.20.8}$$

We consider the two possible cases: either the label is plain ($l \in \mathcal{L}_p$) or it is shared ($l \in \mathcal{L}_\star$).

(*Plain label*) If l is a plain label, from (A.3.20.4)(iii) we have that it must be the case that in T' there is a τ introduced by rule (*Plain*) for this synchronization which is only possible if (A.3.20.5) and (A.3.20.7) and also that $C_1 \equiv C_2$, otherwise the merge $T_1 \bowtie T_2$ (A.3.20.4)(iii) would not be defined. We use C such that $C \equiv C_1 \equiv C_2$. We then have, from Lemma A.3.11 that there is T'' such that:

$$T_1' = T_1'' \bowtie a : C \;\; (i) \qquad \text{and} \qquad Q_1 :: T_1'' \mid B_1 \;\; (ii) \tag{A.3.20.9}$$

From the merge in (A.3.20.4)(iii), considering (A.3.20.5), (A.3.20.9)(i) and (A.3.20.7) we conclude:

$$T' = ((T_1'' \bowtie a : C) \mid \oplus\{!\, l^{\downarrow}(C).B_1; \tilde{B}\}) \bowtie (T_2' \mid \&\{?\, l^{\downarrow}(C).B_2; \tilde{B}'\}) \tag{A.3.20.10}$$

From (A.3.20.10) we have that there is T'' such that: $T'' = (T_2' \mid B_2) \bowtie a : C$, hence from Lemma A.3.7 we conclude:

$$Q_2 :: (T_2' \mid B_2) \bowtie a : C \tag{A.3.20.11}$$

From (A.3.20.10) we also have that there is T''' such that $T''' = (T_1'' \bowtie B_1) \bowtie ((T_2' \bowtie B_2) \bowtie a : C)$, hence from (A.3.20.9)($ii$) and (A.3.20.11) we conclude:

$$Q_1 \mid Q_2 :: (T_1'' \mid B_1) \bowtie ((T_2' \mid B_2) \bowtie a : C) \tag{A.3.20.12}$$

The merge of plain label message types necessarily yields a τ message type. We can thus derive a reduction for type T' as follows:

$$\begin{array}{l}
((T_1'' \bowtie a : C) \mid \oplus\{!\, l^{\downarrow}(C).B_1; \tilde{B}\}) \bowtie (T_2' \mid \&\{?\, l^{\downarrow}(C).B_2; \tilde{B}'\}) \\
\equiv \\
((T_1'' \bowtie a : C) \bowtie T_2') \bowtie \oplus\{\tau\, l^{\downarrow}(C).(B_1 \bowtie B_2); (\cdots)\} \\
\rightarrow \\
((T_1'' \bowtie a : C) \bowtie T_2') \bowtie (B_1 \bowtie B_2) \\
\equiv \\
(T_1'' \mid B_1) \bowtie ((T_2' \mid B_2) \bowtie a : C)
\end{array}$$

Since $T' <: T$ and considering Lemma A.3.19 we have that there is T'' such that $T \rightarrow T''$ and $Q_1 \mid Q_2 :: T''$ which completes the proof for this case.

(*Shared label*) If l is a shared label then by conformance we have that $C_1 \equiv C_2 (\equiv C)$. From the definition of merge and (A.3.20.4)(iii) we conclude it must be the case of (A.3.20.7) and furthermore we have that $\&\{?\, l^{d}(C).B_2; \tilde{B}'\} \equiv \star ?\, l^{d}(C)$. Also, considering Lemma A.3.11, from

either (A.3.20.5) or (A.3.20.6), since $l \in \mathcal{L}_\star$, we have there is T'' such that:

$$T_1' = T_1'' \bowtie a : C_1 \;\; (i) \qquad \text{and} \qquad Q_1 :: T_1'' \mid B_1 \;\; (ii) \tag{A.3.20.13}$$

We show only the proof for (A.3.20.5), as the proof for (A.3.20.6) follows similar lines. From the merge in (A.3.20.4)(iii), and (A.3.20.13)(i) and (A.3.20.7) we conclude:

$$T' = ((T_1'' \bowtie a : C) \mid \oplus\{?\, l^{\downarrow}(C).B_1; \tilde{B}\}) \bowtie (T_2' \mid \star\, ?\, l^d(C)) \tag{A.3.20.14}$$

which leads to:

$$\begin{array}{l}((T_1'' \bowtie a : C) \bowtie T_2') \bowtie (\oplus\{?\, l^{\downarrow}(C).B_1; \tilde{B}\} \bowtie \star ?\, l^d(C)) \\ \equiv \\ ((T_1'' \bowtie a : C) \bowtie T_2') \bowtie (\oplus\{\tau\, l^{\downarrow}(C).(B_1\{!\; l^{\downarrow}(C)/\tau\; l^{\downarrow}(C)\}); (\tilde{B}\{!\; l^{\downarrow}(C)/\tau\; l^{\downarrow}(C)\})\} \mid \star\, ?\, l^d(C))\end{array} \tag{A.3.20.15}$$

From (A.3.20.15) we conclude:

$$(T_2' \mid \star\, ?\, l^d(C)) \bowtie a : C \tag{A.3.20.16}$$

Then from Lemma A.3.7 and (A.3.20.16) we have:

$$Q_2 :: (T_2' \mid \star\, ?\, l^d(C)) \bowtie a : C \tag{A.3.20.17}$$

From (A.3.20.13)(ii), (A.3.20.17) and (A.3.20.15) we derive:

$$Q_1 \mid Q_2 :: (T_1'' \mid B_1) \bowtie ((T_2' \mid \star\, ?\, l^d(C)) \bowtie a : C) \tag{A.3.20.18}$$

From (A.3.20.15) we have:

$$\begin{array}{l}((T_1'' \bowtie a : C) \bowtie T_2') \bowtie (\oplus\{\tau\, l^{\downarrow}(C).(B_1\{!\; l^{\downarrow}(C)/\tau\; l^{\downarrow}(C)\}); (\tilde{B}\{!\; l^{\downarrow}(C)/\tau\; l^{\downarrow}(C)\})\} \mid \star\, ?\, l^d(C)) \\ \to ((T_1'' \bowtie a : C) \bowtie T_2') \bowtie (B_1\{!\; l^{\downarrow}(C)/\tau\; l^{\downarrow}(C)\}) \mid \star\, ?\, l^d(C)) \\ \equiv (T_1'' \mid B_1) \bowtie ((T_2' \mid \star\, ?\, l^d(C)) \bowtie a : C)\end{array} \tag{A.3.20.19}$$

Considering Lemma A.3.19 and $T' <: T$, (A.3.20.14), (A.3.20.15) and (A.3.20.19) we then have that there is T'' such that $T \to T''$ and $Q_1 \mid Q_2 :: T''$ which completes the proof for this case.

(*Case* c `this`$^{\downarrow}$)

We have:

$$c \blacktriangleleft [P] \xrightarrow{\tau} c \blacktriangleleft [Q] \tag{A.3.20.20}$$

derived from:

$$P \xrightarrow{c\ \mathtt{this}^{\downarrow}} Q \tag{A.3.20.21}$$

We have that $c \blacktriangleleft [P] :: T$. Also we have that there is T' such that $T' <: T$ and:

$$c \blacktriangleleft [P] :: T' \tag{A.3.20.22}$$

where (A.3.20.22) is derived from (rule (*Piece*)):

$$P :: L \mid B \;\; (i) \qquad \text{and} \qquad T' = (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \;\; (ii) \tag{A.3.20.23}$$

We must consider the two distinct cases: either the transition originates from a **this** prefix or from a message synchronization.

(**this** *prefix*) We have that the transition originates in a **this** prefix. Considering Lemma A.3.15 and (A.3.20.23)(i) and (A.3.20.21) conclude there are L', B_1, B_2 such that:

$$L \mid B \equiv (L' \mid B_1) \bowtie (\downarrow B_2) \tag{A.3.20.24}$$

and also, considering (A.3.20.23)(ii), we have:

$$Q :: (L' \mid B_1) \bowtie (c : [\downarrow B_2]) \tag{A.3.20.25}$$

From (A.3.20.25) we derive:

$$c \blacktriangleleft [Q] :: ((L' \bowtie c : [\downarrow B_2]) \bowtie c : [\downarrow B_1]) \mid loc(\uparrow B_1) \tag{A.3.20.26}$$

From (A.3.20.24) we conclude that $L \equiv L'$ and:

$$\downarrow B \ = \downarrow B_1 \bowtie \downarrow B_2 \tag{A.3.20.27}$$

and:

$$\uparrow B \ \equiv \uparrow B_1 \tag{A.3.20.28}$$

From (A.3.20.26), (A.3.20.27), (A.3.20.28) we have:

$$c \blacktriangleleft [Q] :: (L \bowtie c : [\downarrow B]) \mid loc(\uparrow B) \tag{A.3.20.29}$$

From (A.3.20.29) and (A.3.20.23)(ii) and $T' <: T$ we then have:

$$c \blacktriangleleft [Q] :: T \tag{A.3.20.30}$$

which completes the proof for this case since $T \to T$.

(*Message synchronization*) We have that the transition originates in a message synchronization. Considering Lemma A.3.18 and from (A.3.20.23)(i) and (A.3.20.21) we conclude that there exist $T'', B_1, B_2, \tilde{B}, \tilde{B}', C_1, C_2, l$ such that:

$$L \mid B \equiv T'' \mid \{p_1\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \mid c : [\{p_2\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}] \tag{A.3.20.31}$$

and $p_i = \texttt{!}$ and $p_j = \texttt{?}$, or:

$$L \mid B \equiv T'' \mid B_1 \mid c : [B_2] \tag{A.3.20.32}$$

and $p_1\, l^{\downarrow}(C_1) \in Msg(B_1)$ and $p_2\, l^{\downarrow}(C_2) \in Msg(B_2)$. From $(L \bowtie c : [\downarrow B]) \mid loc(\uparrow B)$ we directly have that it must be the case of (A.3.20.31) otherwise the merge would be undefined. We show the case when $i = 1$ and $j = 2$ and l is a plain label (the proofs for the other cases follow similar lines). From (A.3.20.31) we conclude there exist L_1, B_3 such that $T'' \equiv L_1 \mid B_3$ and:

$$L \equiv L_1 \mid c : [\&\{\texttt{?}\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}] \tag{A.3.20.33}$$

and:

$$B \equiv B_3 \mid \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\} \tag{A.3.20.34}$$

From (A.3.20.34) we have that:

$$\downarrow B \equiv \downarrow B_3 \mid \oplus\{\,!\, l^{\downarrow}(C_1).(\downarrow B_1); (\downarrow \tilde{B})\} \tag{A.3.20.35}$$

and:

$$loc(\uparrow B) \equiv loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \tag{A.3.20.36}$$

From (A.3.20.23)(ii), (A.3.20.33) and (A.3.20.35) and (A.3.20.36) we have:

$$\begin{array}{rcl} T' & = & ((L_1 \mid c : [\&\{\,?\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]) \bowtie c : [\downarrow B_3 \mid \oplus\{\,!\, l^{\downarrow}(C_1).(\downarrow B_1); (\downarrow \tilde{B})\}]) \\ & & \mid loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \end{array} \tag{A.3.20.37}$$

We also have that $C_1 \equiv C_2$ $(\equiv C)$, otherwise the merge is undefined. From Lemma A.3.18 we then have:

$$Q :: T'' \mid B_1 \mid c : [B_2] \tag{A.3.20.38}$$

which, since $T'' \equiv L_1 \mid B_3$ gives us:

$$Q :: L_1 \mid B_3 \mid B_1 \mid c : [B_2] \tag{A.3.20.39}$$

From (A.3.20.39) we derive:

$$c \blacktriangleleft [Q] :: ((L_1 \mid c : [B_2]) \bowtie c : [\downarrow (B_3 \mid B_1)]) \mid loc(\uparrow (B_3 \mid B_1)) \tag{A.3.20.40}$$

Considering Lemma A.3.4 we conclude:

$$c \blacktriangleleft [Q] :: ((L_1 \mid c : [B_2]) \bowtie c : [\downarrow (B_3 \mid B_1)]) \mid loc(\uparrow B_3) \mid loc(\uparrow \{p_1\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \tag{A.3.20.41}$$

We may then derive the following type reduction:

$$\begin{array}{l} ((L_1 \mid c : [\&\{\,?\, l^{\downarrow}(C_2).B_2; \tilde{B}'\}]) \bowtie c : [\downarrow B_3 \mid \oplus\{\,!\, l^{\downarrow}(C_1).(\downarrow B_1); (\downarrow \tilde{B})\}]) \\ \;\mid loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \\ \equiv \\ ((L_1 \bowtie c : [\downarrow B_3]) \bowtie c : [\oplus\{\tau\, l^{\downarrow}(C).((\downarrow B_1) \bowtie B_2); \tilde{B}'''\}]) \mid loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \\ \to \\ ((L_1 \bowtie c : [\downarrow B_3]) \bowtie c : [(\downarrow B_1) \bowtie B_2]) \mid loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \\ \equiv \\ ((L_1 \mid c : [B_2]) \bowtie c : [\downarrow (B_3 \mid B_1)]) \mid loc(\uparrow B_3) \mid loc(\uparrow \oplus\{\,!\, l^{\downarrow}(C_1).B_1; \tilde{B}\}) \end{array} \tag{A.3.20.42}$$

From (A.3.20.37) and $T' <: T$ and considering Lemma A.3.19 we conclude there is T'' such that $T \to T''$ and $c \blacktriangleleft [Q] :: T''$ which completes the proof for this case. ■

Proposition 5.3.9 (Error Freeness)

(repetition of the statement in page 122)

If P is a well-typed process then P is not an error process.

Proof. The result follows from the definition of merge $\bowtie$. A parallel composition is well typed if the types of the parallel branches can be merged. Since it is not possible to synchronize message types defined on plain labels with the same polarity (which is the case for competing messages) and such types are not apart # (the label sets are not disjoint) it is not possible to merge them. Hence the composition of processes that exhibit competing plain messages is not typable, hence error processes are not typable.

From the definition of error process we there are $\mathcal{C}, \mathcal{C}', Q, R, d, l, \texttt{flag}$ such that $l \in \mathcal{L}_p$ and $P = \mathcal{C}[Q \mid R]$ and:

$$\mathcal{C}[Q \mid \mathcal{C}'[l^d-.\texttt{flag}^{\downarrow}!()]] \to \mathcal{C}[Q' \mid \mathcal{C}'[\texttt{flag}^{\downarrow}!()]]$$

and

$$\mathcal{C}[R \mid \mathcal{C}'[l^d-.\texttt{flag}^{\downarrow}!()]] \to \mathcal{C}[R' \mid \mathcal{C}'[\texttt{flag}^{\downarrow}!()]]$$

Let us consider the case when $l^d- = l^{\downarrow}?(x)$ and $\mathcal{C}'[l^{\downarrow}?(x).\texttt{flag}^{\downarrow}!()] = c \blacktriangleleft [l^{\downarrow}?(x).\texttt{flag}^{\downarrow}!()]$ and $\mathcal{C}[Q \mid R] = Q \mid R$. We then have that there is a such that either:

$$Q \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q' \tag{A.3.21.1}$$

or:

$$Q \xrightarrow{c\ l^{\downarrow}!(a)} Q' \tag{A.3.21.2}$$

and b such that either:

$$R \xrightarrow{(\nu b)c\ l^{\downarrow}!(b)} R' \tag{A.3.21.3}$$

or:

$$R \xrightarrow{c\ l^{\downarrow}!(b)} R' \tag{A.3.21.4}$$

We consider the case (A.3.21.1) and (A.3.21.4), and elide the proof for the remaining combinations as their proofs follow similar lines. Let us assume that there are T_1, T_2 such that:

$$Q :: T_1 \tag{A.3.21.5}$$

and:

$$R :: T_2 \tag{A.3.21.6}$$

From (A.3.21.1) and (A.3.21.5) and Lemma A.3.14 we conclude there is $T_1', B_1, C_1, \tilde{B}$ such that:

$$T_1 \equiv T_1' \mid c : [\oplus\{!\ l^{\downarrow}(C_1).B_1; \tilde{B}\}] \tag{A.3.21.7}$$

or:

$$T_1 \equiv T_1' \mid c : [\oplus\{\tau\ l^{\downarrow}(C_1).B_1; \tilde{B}\}] \tag{A.3.21.8}$$

and from (A.3.21.5) and (A.3.21.6) and Lemma A.3.12 we conclude there is $T_2', B_2, C_2, \tilde{B}'$ such that:

$$T_2 \equiv T_2' \mid c : [\oplus\{!\ l^{\downarrow}(C_2).B_2; \tilde{B}'\}] \tag{A.3.21.9}$$

or:

$$T_2 \equiv T_2' \mid c : [\oplus\{\tau\ l^{\downarrow}(C_2).B_2; \tilde{B}'\}] \tag{A.3.21.10}$$

In any of such cases, by definition of merge, it is not possible to synchronize such choice types.

Furthermore they are not apart as both types specify message $l^{\downarrow}$ in conversation c. Thus there is no type T' such that $T' = T_1 \bowtie T_2$ and therefore $Q \mid R$ is untypable. ∎

Lemma A.3.22
If $T_1 <: T_2$ and $T_2 <: T_3$ then $T_1 <: T_3$.

Proof. By induction on the derivation of $T_1 <: T_2$ and $T_2 <: T_3$. We show two interesting cases.

(*Case* (*SubExpRec*) *and* (*SubExpUnfold*))

We have that:

$$\texttt{rec}\,\mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\mathcal{X}\rangle) <: \star M_1^! \mid \ldots \mid \star M_k^! \mid \texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle \tag{A.3.22.1}$$

and:

$$\star M_1^! \mid \ldots \mid \star M_k^! \mid \texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle <: \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle\rangle \tag{A.3.22.2}$$

We derive that:

$$\texttt{rec}\,\mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\mathcal{X}\rangle) <: \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\mathcal{X}\rangle)\rangle \tag{A.3.22.3}$$

and:

$$\begin{array}{c}\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.(\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\mathcal{X}\rangle)\rangle \\ <: \\ \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\star M_1^! \mid \ldots \mid \star M_k^! \mid \texttt{rec}\,\mathcal{X}.(B\langle\mathcal{X}\rangle)\rangle\end{array} \tag{A.3.22.4}$$

and also:

$$\begin{array}{c}\star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\star M_1^! \mid \ldots \mid \star M_k^! \mid \texttt{rec}\,\mathcal{X}.(B\langle\mathcal{X}\rangle)\rangle \\ <: \\ \star M_1^! \mid \ldots \mid \star M_k^! \mid \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle\rangle\end{array} \tag{A.3.22.5}$$

and:

$$\star M_1^! \mid \ldots \mid \star M_k^! \mid \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle\rangle <: \star M_1^! \mid \ldots \mid \star M_k^! \mid B\langle\texttt{rec}\,\mathcal{X}.B\langle\mathcal{X}\rangle\rangle \tag{A.3.22.6}$$

which completes the proof for this case.

(*Case* (*SubProject*) *and* (*SubParPref*))

We have that:

$$M.(B_1 \mid B_2) <: \downarrow(B_1 \mid B_2) \mid M.\uparrow(B_1 \mid B_2) \tag{A.3.22.7}$$

and:

$$\downarrow(B_1 \mid B_2) \mid M.\uparrow(B_1 \mid B_2) <: \downarrow(B_1 \mid B_2) \mid M.(\uparrow B_1) \mid \uparrow B_2 \tag{A.3.22.8}$$

We derive that:

$$M.(B_1 \mid B_2) <: M.B_1 \mid B_2 \tag{A.3.22.9}$$

and:

$$M.B_1 \mid B_2 <: \downarrow B_1 \mid \downarrow B_2 \mid M.(\uparrow B_1) \mid \uparrow B_2 \tag{A.3.22.10}$$

which completes the proof for this case. ∎

A.4 Chapter 6

Proposition 6.3.5 (Event Ordering Weakening)
(repetition of the statement in page 146)
Let P be a process such that $\Gamma; \Delta \vdash_\ell P$. If $wf(\Gamma', \Delta')$ and $\Gamma \subseteq \Gamma'$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$ then $\Gamma'; \Delta' \vdash_\ell P$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_\ell P$. Intuitively if $\Gamma; \Delta$ already prove that events are well ordered in P then $\Gamma'; \Delta'$ describe extra events that do not pertain to P, and hence do not interfere in verifying the event ordering of P. We show the case when P is of the form $l^d!(n).P'$, of the form $\Sigma_{i \in I}\, l_i^d?(x_i).P_i$, of the form $(\boldsymbol{\nu} a)P'$, of the form $\mathbf{rec}\ \mathcal{X}.P$, of the form $\mathcal{X}$, and of the form $\mathbf{rec}_t\ \mathcal{X}.P$.

(*Case* $l^d!(n).P'$)

We have that:

$$\Gamma; \Delta \vdash_\ell l^d!(n).P' \tag{A.4.1.1}$$

derived from (rule (*Output*)):

$$(\ell(d).l.(x)\Gamma' \bot \Gamma); \Delta \vdash_\ell P' \ \ (i) \qquad \text{and} \qquad \Gamma'\{x/n\} \subseteq (\ell(d).l.(y)\Gamma' \bot \Gamma) \ \ (ii) \tag{A.4.1.2}$$

and $wf(\Gamma, \Delta)$. Let us consider Γ'' and Δ' such that $wf(\Gamma'', \Delta')$ and $\Gamma \subseteq \Gamma''$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$. Then $(\ell(d).l.(x)\Gamma' \bot \Gamma'')$ is an event ordering. By induction hypothesis on (A.4.1.2)(i) we conclude:

$$(\ell(d).l.(x)\Gamma' \bot \Gamma''); \Delta' \vdash_\ell P' \tag{A.4.1.3}$$

We immediately have that $(\ell(d).l.(x)\Gamma' \bot \Gamma) \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma'')$, which together with (A.4.1.2)(ii) gives us:

$$\Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma'') \tag{A.4.1.4}$$

From (A.4.1.3) and (A.4.1.4) and $wf(\Gamma'', \Delta')$ we derive

$$\Gamma''; \Delta' \vdash_\ell l^d!(n).P' \tag{A.4.1.5}$$

which completes the proof.

(*Case* $\Sigma_{i \in I}\, l_i^d!(x_i).P_i$)

We have that:

$$\Gamma; \Delta \vdash_\ell \Sigma_{i \in I}\, l_i^d!(x_i).P_i \tag{A.4.1.6}$$

derived from (rule (*Input*)):

$$\forall_{i \in I}((\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma) \cup \Gamma_i'\{y_i/x_i\}; \Delta \vdash_\ell P_i \quad \Gamma_i' \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma)) \tag{A.4.1.7}$$

and $wf(\Gamma, \Delta)$. Let us consider Γ'' and Δ' such that $wf(\Gamma'', \Delta')$ and $\Gamma \subseteq \Gamma''$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$. From $\Gamma_i' \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma)$ we

directly have that:

$$\Gamma_i' \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma_i' \perp \Gamma'') \tag{A.4.1.8}$$

for all $i \in I$. Also, from the fact that Γ'' is an event ordering we have that $(\ell(d).l_i.(y_i)\Gamma_i' \perp \Gamma'')$ is an event ordering. Let us consider that $e(x_i) \notin dom(\Gamma'')$ (otherwise we α-convert x_i). Then, considering (A.4.1.8) we have that $(\ell(d).l_i.(y_i)\Gamma_i' \perp \Gamma'') \cup \Gamma_i'\{y_i/x_i\}$ is an event ordering. By induction hypothesis on (A.4.1.7) we conclude:

$$(\ell(d).l_i.(y_i)\Gamma_i' \perp \Gamma'') \cup \Gamma_i'\{y_i/x_i\}; \Delta' \vdash_\ell P_i \tag{A.4.1.9}$$

From (A.4.1.9) and (A.4.1.8) and $wf(\Gamma'', \Delta')$ we derive

$$\Gamma''; \Delta' \vdash_\ell \Sigma_{i \in I}\, l_i^d!(x_i).P_i \tag{A.4.1.10}$$

which completes the proof.

(*Case* $(\boldsymbol{\nu} a)P'$)

We have that:

$$\Gamma; \Delta \vdash_\ell (\boldsymbol{\nu} a)P' \tag{A.4.1.11}$$

derived from (rule (*Res*)):

$$\Gamma'; \Delta' \vdash_\ell P' \;\; (i) \qquad \text{and} \qquad \Gamma = \Gamma' \setminus a \;\; (ii) \qquad \text{and} \qquad \Delta = \Delta' \setminus a \;\; (iii) \tag{A.4.1.12}$$

Let us consider Γ'' and Δ'' such that $wf(\Gamma'', \Delta'')$ and $\Gamma \subseteq \Gamma''$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta''(\mathcal{X})$. Then considering $e(a) \notin dom(\Gamma'')$ and that for all $\mathcal{X}$ we have that $e(a) \notin dom(\Delta''(\mathcal{X}))$ (otherwise we α-convert a) we have that $wf(\Gamma' \cup \Gamma'', \Delta''')$ where $\Delta'''(\mathcal{X}) = \Delta(\mathcal{X})' \cup \Delta''(\mathcal{X})$. By induction hypothesis on (A.4.1.12)(i) we conclude:

$$\Gamma' \cup \Gamma''; \Delta''' \vdash_\ell P' \tag{A.4.1.13}$$

We then have that $\Gamma'' = (\Gamma' \cup \Gamma'') \setminus a$ and $\Delta'' = (\Delta''') \setminus a$ and hence:

$$\Gamma''; \Delta'' \vdash_\ell (\boldsymbol{\nu} a)P' \tag{A.4.1.14}$$

which completes the proof for this case.

(*Case* $\textsf{\textbf{rec}}\ \mathcal{X}.P'$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textsf{\textbf{rec}}\ \mathcal{X}.P' \tag{A.4.1.15}$$

derived from (rule (*Rec*)):

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P' \tag{A.4.1.16}$$

Let us consider Γ' and Δ' such that $wf(\Gamma', \Delta')$ and $\Gamma \subseteq \Gamma'$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$. By induction hypothesis on (A.4.1.16) we conclude:

$$\Gamma'; \Delta', (\mathcal{X} \to \Gamma') \vdash_\ell P' \tag{A.4.1.17}$$

from which we derive:

$$\Gamma'; \Delta' \vdash_\ell \textsf{\textbf{rec}}\ \mathcal{X}.P' \tag{A.4.1.18}$$

which completes the proof.

(*Case* $\mathcal{X}$)

We have that:

$$\Gamma; \Delta \vdash_\ell \mathcal{X} \tag{A.4.1.19}$$

derived from (rule (*RecVar*)):

$$\mathcal{X} \in dom(\Delta) \;\; (i) \qquad wf(\Gamma, \Delta) \;\; (ii) \tag{A.4.1.20}$$

Let us consider Γ' and Δ' such that $wf(\Gamma', \Delta')$ and $\Gamma \subseteq \Gamma'$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$. Then we immediately have:

$$\Gamma'; \Delta' \vdash_\ell \mathcal{X} \tag{A.4.1.21}$$

which completes the proof.

(*Case* $\textsf{rec}_t\ \mathcal{X}.P'$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textsf{rec}_t\ \mathcal{X}.P' \tag{A.4.1.22}$$

derived from (rule (*RecTerm*)):

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P' \tag{A.4.1.23}$$

and $wf(\Gamma, \Delta)$. Let us consider Γ' and Δ' such that $wf(\Gamma', \Delta')$ and $\Gamma \subseteq \Gamma'$ and for all $\mathcal{X}$ such that $\mathcal{X} \in dom(\Gamma)$ it is the case that $\Delta(\mathcal{X}) \subseteq \Delta'(\mathcal{X})$. By induction hypothesis on (A.4.1.16) we conclude:

$$\Delta'(\mathcal{X}); \Delta' \vdash_\ell P' \tag{A.4.1.24}$$

from which, considering $wf(\Gamma', \Delta')$, we derive:

$$\Gamma'; \Delta' \vdash_\ell \textsf{rec}_t\ \mathcal{X}.P' \tag{A.4.1.25}$$

which completes the proof. ■

Lemma A.4.2

Let P be a process such that $\Gamma; \Delta \vdash_\ell P$. If $n \notin fn(P) \cup fv(P)$ then $\Gamma \setminus n; \Delta \setminus n \vdash_\ell P$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_\ell P$, following expected lines. ■

Lemma 6.3.6 (Substitution)

(repetition of the statement in page 147)

Let P be such that $\Gamma; \Delta \vdash_\ell P$. If $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ then $\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} P\{x/n\}$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_\ell P$.

(*Case* $x \blacktriangleleft [P]$)

We have that:

$$\Gamma; \Delta \vdash_\ell x \blacktriangleleft [P] \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.1}$$

(A.4.3.1)(i) is derived from:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),x)} P \tag{A.4.3.2}$$

By induction hypothesis on (A.4.3.2) and (A.4.3.1)(ii) we have:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{(\ell(\downarrow)\{x/n\},n)} P\{x/n\} \tag{A.4.3.3}$$

From (A.4.3.3) we derive:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} n \blacktriangleleft [P\{x/n\}] \tag{A.4.3.4}$$

which completes the proof for this case.

(*Cases* $n \blacktriangleleft [P]$ *and* $m \blacktriangleleft [P]$)

Follow expected lines.

(*Case* $(\boldsymbol{\nu} a)P$)

We have that:

$$\Gamma; \Delta \vdash_{\ell} (\boldsymbol{\nu} a)P \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.5}$$

(A.4.3.5)(i) is derived from:

$$\Gamma'; \Delta' \vdash_{\ell} P \tag{A.4.3.6}$$

where $\Gamma = \Gamma' \setminus a$ and $\Delta = \Delta' \setminus a$. From (A.4.3.5)($ii$) and $n \neq a$ (otherwise we α-convert a) we have $\Gamma'\{x/n\} \subseteq \Gamma' \setminus x$ and $\Gamma \setminus x = (\Gamma' \setminus x) \setminus a$. Then by induction hypothesis on (A.4.3.6) we have:

$$\Gamma' \setminus x; \Delta' \setminus x \vdash_{\ell\{x/n\}} P\{x/n\} \tag{A.4.3.7}$$

From (A.4.3.7) we derive:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} (\boldsymbol{\nu} a)(P\{x/n\}) \tag{A.4.3.8}$$

which completes the proof for this case.

(*Case* $l^d!(x).P$)

We have that:

$$\Gamma; \Delta \vdash_{\ell} l^d!(x).P \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.9}$$

(A.4.3.9)(i) is derived from (rule (*Output*)):

$$(\ell(d).l.(y)\Gamma'' \bot \Gamma); \Delta \vdash_{\ell} P \;\; (i) \qquad \text{and} \qquad \Gamma''\{y/x\} \subseteq (\ell(d).l.(y)\Gamma'' \bot \Gamma) \;\; (ii) \tag{A.4.3.10}$$

and $wf(\Gamma, \Delta)$. From (A.4.3.10)(i), considering Lemma 6.3.5, we conclude $\Gamma; \Delta \vdash_{\ell} P$. Let us take $\ell' = \ell\{x/n\}$. Then by induction hypothesis we have:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell'} P\{x/n\} \tag{A.4.3.11}$$

We have that $\ell(d).l.(y)\Gamma'' \in dom(\Gamma)$ and since $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ we conclude:

$$\ell'(d).l.((y)\Gamma''\{x/n\}) \in dom(\Gamma \setminus x) \tag{A.4.3.12}$$

Also from (A.4.3.10)(i) we have that events in P are of greater order with respect to event

$\ell(d).l.(y)\Gamma''$, so events in $P\{x/n\}$ are of greater order with respect to $\ell'(d).l.(y)(\Gamma''\{x/n\})$:

$$(\ell'(d).l.((y)\Gamma''\{x/n\})\bot(\Gamma \setminus x)); \Delta \setminus x \vdash_{\ell'} P\{x/n\} \tag{A.4.3.13}$$

From (A.4.3.10)(ii) we conclude:

$$(\Gamma''\{y/x\})\{x/n\} \subseteq (\ell(d).l.(y)\Gamma''\bot\Gamma)\{x/n\} \subseteq (\ell'(d).l.((y)\Gamma''\{x/n\})\bot(\Gamma \setminus x)) \tag{A.4.3.14}$$

and hence $\Gamma''\{y/n\} \subseteq (\ell'(d).l.((y)\Gamma''\{x/n\})\bot(\Gamma \setminus x))$. Then, considering also (A.4.3.13), we have:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell'} l^d!(n).(P\{x/n\}) \tag{A.4.3.15}$$

which completes the proof for this case.

(*Cases* $l^d!(n).P$ *and* $l^d!(m).P$)

Follow similar lines to the previous case.

(*Case* $\Sigma_{i\in I}\, l_i^d?(z_i).P_i$)

We have that:

$$\Gamma; \Delta \vdash_{\ell} \Sigma_{i\in I}\, l_i^d?(z_i).P_i \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.16}$$

(A.4.3.16)(i) is derived from:

$$\forall_{i\in I}((\ell(d).l_i.(y_i)\Gamma'_i \bot \Gamma) \cup \Gamma'_i\{y_i/z_i\}; \Delta \vdash_{\ell} P_i \quad \Gamma'_i \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma'_i \bot \Gamma)) \tag{A.4.3.17}$$

and $wf(\Gamma, \Delta)$. From (A.4.3.17)(i), considering Proposition 6.3.5, we conclude $\Gamma \cup \Gamma'_i\{y_i/z_i\}; \Delta \vdash_{\ell} P$. Also since $y_i \neq x \neq z_i$ (otherwise we α-convert y_i or z_i) and $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ and $\Gamma'_i \setminus y_i \subseteq \Gamma$ we have that $(\Gamma \cup \Gamma'_i\{y_i/z_i\})\{x/n\} \subseteq (\Gamma \cup \Gamma'_i\{x/n\}\{y_i/z_i\}) \setminus x$. Let us take $\ell' = \ell\{x/n\}$. Then by induction hypothesis we have:

$$(\Gamma \cup \Gamma'_i\{x/n\}\{y_i/z_i\}) \setminus x; \Delta \setminus x \vdash_{\ell'} P_i\{x/n\} \tag{A.4.3.18}$$

We have that $\ell(d).l_i.(y_i)\Gamma'_i \in dom(\Gamma)$ and since $\Gamma\{x/n\} \subseteq \Gamma \setminus x$ we conclude:

$$\ell'(d).l_i.((y_i)\Gamma'_i\{x/n\}) \in dom(\Gamma \setminus x) \tag{A.4.3.19}$$

Then from (A.4.3.17) and (A.4.3.18) we have that:

$$(\ell'(d).l_i.((y_i)\Gamma'_i\{x/n\})\bot(\Gamma \setminus x)) \cup (\Gamma'_i\{x/n\}\{y_i/z_i\}); \Delta \setminus x \vdash_{\ell'} P_i\{x/n\} \tag{A.4.3.20}$$

and:

$$(\Gamma'_i\{x/n\}) \setminus y_i \subseteq (\ell'(d).l_i.((y_i)\Gamma'_i\{x/n\})\bot(\Gamma \setminus x)) \tag{A.4.3.21}$$

From (A.4.3.20) and (A.4.3.21) we conclude:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell'} \Sigma_{i\in I}\, l_i^d!(z_i).(P_i\{x/n\}) \tag{A.4.3.22}$$

which completes the proof for this case.

(*Case* $\textbf{this}(y).P$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textsf{\textbf{this}}(y).P \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.23}$$

(A.4.3.23)(i) is derived from:

$$\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{y/\ell(\downarrow)\} \prec_\Gamma e_2\{y/\ell(\downarrow)\})\}; \Delta \vdash_\ell P \tag{A.4.3.24}$$

and $(\ell(\downarrow) \neq \textsf{here})$. From (A.4.3.23)$(ii)$ and $x \neq y$ (otherwise we α-convert y) we have:

$$\begin{array}{l}(\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{y/\ell(\downarrow)\} \prec_\Gamma e_2\{y/\ell(\downarrow)\})\})\{x/n\} \\ \subseteq \\ (\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{y/\ell(\downarrow)\} \prec_\Gamma e_2\{y/\ell(\downarrow)\})\}) \setminus x\end{array} \tag{A.4.3.25}$$

Then by induction hypothesis on (A.4.3.24) we have:

$$(\Gamma \cup \{(e_1 \prec e_2) \mid (e_1\{y/\ell(\downarrow)\} \prec_\Gamma e_2\{y/\ell(\downarrow)\})\}) \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} P\{x/n\} \tag{A.4.3.26}$$

We have that $\ell'(\downarrow) \neq \textsf{here}$ for $\ell' = \ell\{x/n\}$. Then from (A.4.3.26) we derive:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} \textsf{\textbf{this}}(y).(P\{x/n\}) \tag{A.4.3.27}$$

which completes the proof for this case.

(*Case* $P \mid Q$)

Follows directly from induction hypothesis.

(*Case* $\mathbf{0}$)

Direct.

(*Case* $\mathcal{X}$)

We have that:

$$\Gamma; \Delta \vdash_\ell \mathcal{X} \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.28}$$

(A.4.3.28)(i) is derived from:

$$\mathcal{X} \in \mathit{dom}(\Delta) \qquad \text{and} \qquad \mathit{wf}(\Gamma, \Delta) \tag{A.4.3.29}$$

We directly have that $\mathcal{X} \in \mathit{dom}(\Delta \setminus x)$ and $\mathit{wf}(\Gamma \setminus x, \Delta \setminus x)$ and hence:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} \mathcal{X} \tag{A.4.3.30}$$

which completes the proof for this case.

(*Case* $\textsf{\textbf{rec}}\ \mathcal{X}.P$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textsf{\textbf{rec}}\ \mathcal{X}.P \;\; (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \;\; (ii) \tag{A.4.3.31}$$

(A.4.3.31)(i) is derived from:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P \tag{A.4.3.32}$$

By induction hypothesis we have that:

$$\Gamma \setminus x; (\Delta \setminus x, \mathcal{X} \rightarrow \Gamma \setminus x) \vdash_{\ell} P \tag{A.4.3.33}$$

from which we derive:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} \textbf{rec}\ \mathcal{X}.P \tag{A.4.3.34}$$

which completes the proof for this case.

(*Case* $\textbf{rec}_t\ \mathcal{X}.P$)

We have that:

$$\Gamma; \Delta \vdash_{\ell} \textbf{rec}_t\ \mathcal{X}.P \ \ (i) \qquad \text{and} \qquad \Gamma\{x/n\} \subseteq \Gamma \setminus x \ \ (ii) \tag{A.4.3.35}$$

(A.4.3.35)(i) is derived from:

$$\Delta(\mathcal{X}); \Delta \vdash_{\ell} P \tag{A.4.3.36}$$

and where $wf(\Gamma, \Delta)$ and $fv(\Delta(\mathcal{X})) = \emptyset$. From $fv(\Delta(\mathcal{X})) = \emptyset$ we have that:

$$\Delta(\mathcal{X})\{x/n\} = \Delta(\mathcal{X}) \setminus x = \Delta(\mathcal{X}) \tag{A.4.3.37}$$

By induction hypothesis we then have that:

$$\Delta(\mathcal{X}) \setminus x; \Delta \setminus x \vdash_{\ell} P \tag{A.4.3.38}$$

From $wf(\Gamma, \Delta)$ we directly have that $wf(\Gamma \setminus x, \Delta \setminus x)$ and hence:

$$\Gamma \setminus x; \Delta \setminus x \vdash_{\ell\{x/n\}} \textbf{rec}_t\ \mathcal{X}.P \tag{A.4.3.39}$$

which completes the proof for this case. ∎

Lemma A.4.4

If $\Gamma; \Delta \vdash_{\ell} P$ *then* $wf(\Gamma, \Delta)$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_{\ell} P$, following expected lines. Notice that each rule in the proof system that introduces a new ordering in the conclusion has a premise that verifies the well-foundedness condition on the event ordering plus recursion environment. ∎

Lemma A.4.5

Let Q *be a well-formed process. If* $\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma' \vdash_{\ell} Q$ *and* $\Gamma'; \Delta, \mathcal{X} \rightarrow \Gamma' \vdash_{\ell} P$ *and* $fv(\Gamma') = \emptyset$ *then* $\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma' \vdash Q\{\mathcal{X}/\textbf{rec}_t\ \mathcal{X}.P\}$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma' \vdash_{\ell} Q$.

(*Cases* (*Par*), (*Res*) *and* (*Stop*))

Direct from induction hypothesis.

(*Case* (*Rec*))

We have that:

$$\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma' \vdash_{\ell} \textbf{rec}\ \mathcal{Y}.Q \tag{A.4.5.1}$$

and $\Gamma'; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell P$. (A.4.5.1) is derived from:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma', \mathcal{Y} \to \Gamma \vdash_\ell Q \tag{A.4.5.2}$$

Considering Proposition 6.3.5 we have that $\Gamma'; \Delta, \mathcal{X} \to \Gamma', \mathcal{Y} \to \Gamma \vdash_\ell P$. Then, by induction hypothesis we conclude:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma', \mathcal{Y} \to \Gamma \vdash_\ell Q\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \tag{A.4.5.3}$$

from which we derive:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathbf{rec}\ \mathcal{Y}.(Q\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\}) \tag{A.4.5.4}$$

which completes the proof for this case.

(*Case* (*RecTerm*))

We have that:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathbf{rec}_t\ \mathcal{Y}.Q \tag{A.4.5.5}$$

and $\Gamma'; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell P$. (A.4.5.5) is derived from:

$$\Delta(\mathcal{Y}); \Delta, \mathcal{X} \to \Gamma' \vdash_\ell Q \tag{A.4.5.6}$$

and $wf(\Gamma, \Delta, \mathcal{X} \to \Gamma')$. By induction hypothesis we conclude:

$$\Delta(\mathcal{Y}); \Delta, \mathcal{X} \to \Gamma' \vdash_\ell Q\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \tag{A.4.5.7}$$

from which we derive:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathbf{rec}_t\ \mathcal{Y}.(Q\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\}) \tag{A.4.5.8}$$

which completes the proof for this case.

(*Case* (*RecVar*))

We have that:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathcal{Y} \tag{A.4.5.9}$$

and $\Gamma'; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell P$. (A.4.5.9) is derived from $\mathcal{Y} \in dom(\Delta, \mathcal{X} \to \Gamma')$ and $wf(\Gamma, \Delta, \mathcal{X} \to \Gamma')$. If $\mathcal{X} \neq \mathcal{Y}$ we directly have that:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathcal{Y} \tag{A.4.5.10}$$

We have that $\Gamma'; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell P$. and hence if $\mathcal{X} = \mathcal{Y}$ we have:

$$\Gamma; \Delta, \mathcal{X} \to \Gamma' \vdash_\ell \mathbf{rec}_t\ \mathcal{X}.P \tag{A.4.5.11}$$

which completes the proof for this case.

(*Case* (*Piece*))

Since Q is well-formed we have that $(n \blacktriangleleft [Q])\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} = n \blacktriangleleft [Q]$ hence the result is direct.

(*Cases* (*Input*), (*Output*) *and* (*This*))

Follow from induction hypothesis in expected lines. ■

Lemma A.4.6
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{l^d?(a)} Q$ and $(\ell(d).l.(x)\Gamma') \in dom(\Gamma)$ and $\Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma') \bot \Gamma$ then there are Γ'', Δ' such that $\Gamma''; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the derivation of the transition $P \xrightarrow{l^d?(a)} Q$. We show the case when P is an input summation, when P is a recursive process and when P is a terminal recursive process.

(*Case* $\Sigma_{i \in I}\ l_i^d?(x_i).P_i \xrightarrow{l_j^d?(a)} P_j\{x_j/a\}$)

We have that:

$$\Gamma; \Delta \vdash_\ell \Sigma_{i \in I}\ l_i^d?(x_i).P_i \qquad \text{and} \qquad \Sigma_{i \in I}\ l_i^d?(x_i).P_i \xrightarrow{l_j^d?(a)} P_j\{x_j/a\} \tag{A.4.6.1}$$

and:

$$(\ell(d).l_i.(y_j)\Gamma'_j) \in dom(\Gamma) \qquad \text{and} \qquad \Gamma'_j\{y_j/a\} \subseteq (\ell(d).l_i.(y_j)\Gamma'_j \bot \Gamma) \tag{A.4.6.2}$$

We have that (A.4.6.1) is derived from:

$$\forall_{i \in I}((\ell(d).l_i.(y_i)\Gamma'_i \bot \Gamma) \cup \Gamma'_i\{y_i/x_i\}; \Delta \vdash_\ell P_i) \tag{A.4.6.3}$$

in particular for j we have:

$$(\ell(d).l_j.(y_j)\Gamma'_j \bot \Gamma) \cup \Gamma'_j\{y_j/x_j\}; \Delta \vdash_\ell P_j \tag{A.4.6.4}$$

From Lemma 6.3.6 considering (A.4.6.4) and (A.4.6.2) we then have (where $\ell\{x_j/a\} = \ell$):

$$(\ell(d).l_j.(y_j)\Gamma'_j \bot \Gamma); \Delta \setminus x \vdash_\ell P_j\{x_j/a\} \tag{A.4.6.5}$$

From (A.4.6.5) and considering Proposition 6.3.5 we have:

$$\Gamma; \Delta \vdash_\ell P_j\{x_j/a\} \tag{A.4.6.6}$$

which along with $\Gamma \cup \bigcup \Delta \subseteq \Gamma \cup \bigcup \Delta$ completes the proof for this case.

(*Case* $\textbf{rec}\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textbf{rec}\ \mathcal{X}.P\ \ (i) \qquad \text{and} \qquad \textbf{rec}\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q\ \ (ii) \tag{A.4.6.7}$$

and:

$$(\ell(d).l.(x)\Gamma') \in dom(\Gamma) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma) \tag{A.4.6.8}$$

We have that (A.4.6.7)(i) is derived from:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P \tag{A.4.6.9}$$

which assuming $fv(\Gamma) = \emptyset$ (otherwise, since $fv(P) = \emptyset$ we use Lemma A.4.2 and trim down the ordering in such irrelevant names) and considering Lemma A.4.5 gives us that:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P\{\mathcal{X}/\textbf{rec}_t\ \mathcal{X}.P\} \tag{A.4.6.10}$$

(A.4.6.7)(ii) is derived from:

$$P\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \xrightarrow{l^d?(a)} Q \qquad \text{(A.4.6.11)}$$

By induction hypothesis we conclude:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell Q \qquad \text{(A.4.6.12)}$$

and $\Gamma \cup \bigcup(\Delta, \mathcal{X} \to \Gamma) \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case.

($Case$ $\mathbf{rec}_t\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q$)

We have that:

$$\Gamma; \Delta \vdash_\ell \mathbf{rec}_t\ \mathcal{X}.P\ \ (i) \qquad \text{and} \qquad \mathbf{rec}_t\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q\ \ (ii) \qquad \text{(A.4.6.13)}$$

and:

$$(\ell(d).l.(x)\Gamma') \in dom(\Gamma) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma) \qquad \text{(A.4.6.14)}$$

We have that (A.4.6.13)(i) is derived from:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P \qquad \text{(A.4.6.15)}$$

and $fv(\Delta(\mathcal{X})) = \emptyset$ and $wf(\Gamma, \Delta)$. Then considering Lemma A.4.5 we conclude:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \qquad \text{(A.4.6.16)}$$

from which, considering Proposition 6.3.5, we derive:

$$\Gamma \cup \Delta(\mathcal{X}); \Delta \vdash_\ell P\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \qquad \text{(A.4.6.17)}$$

(A.4.6.13)(ii) is derived from:

$$P\{\mathcal{X}/\mathbf{rec}_t\ \mathcal{X}.P\} \xrightarrow{l^d?(a)} Q \qquad \text{(A.4.6.18)}$$

By induction hypothesis we conclude:

$$\Gamma \cup \Delta(\mathcal{X}); \Delta \vdash_\ell Q \qquad \text{(A.4.6.19)}$$

and $\Gamma \cup \Delta(\mathcal{X}) \cup \bigcup \Delta \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case. ■

Lemma A.4.7

Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{c\ l^{\downarrow}?(a)} Q$ and $(c.l.(x)\Gamma') \in dom(\Gamma)$ and $\Gamma'\{x/a\} \subseteq (c.l.(x)\Gamma')\bot\Gamma$ then there are Γ'', Δ' such that $\Gamma''; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the derivation of the transition $P \xrightarrow{c\ l^{\downarrow}?(a)} Q$. We show the base case.

($Case$ $c \blacktriangleleft [P'] \xrightarrow{c\ l^{\downarrow}?(a)} c \blacktriangleleft [Q']$)

We have that:

$$\Gamma; \Delta \vdash_\ell c \blacktriangleleft \left[P'\right] \qquad \text{(A.4.7.1)}$$

Let us consider:

$$c \blacktriangleleft \left[P'\right] \xrightarrow{c\ l^{\downarrow}?(a)} c \blacktriangleleft \left[Q'\right] \qquad \text{and} \qquad (c.l.(x)\Gamma') \in dom(\Gamma) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (c.l.(x)\Gamma' \bot \Gamma) \tag{A.4.7.2}$$

We have that (A.4.7.1) is derived from:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),c)} P' \tag{A.4.7.3}$$

and (A.4.7.2) is derived from:

$$P' \xrightarrow{l^{\downarrow}?(a)} Q' \tag{A.4.7.4}$$

From Lemma A.4.6 considering (A.4.7.4), (A.4.7.3) and (A.4.7.2) we have:

$$\Gamma''; \Delta, \Delta' \vdash_{(\ell(\downarrow),c)} Q' \tag{A.4.7.5}$$

and $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$. From (A.4.7.5) we derive:

$$\Gamma''; \Delta, \Delta' \vdash_{\ell} c \blacktriangleleft \left[Q'\right] \tag{A.4.7.6}$$

which completes the proof for this case. ∎

Lemma A.4.8
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{l^d!(a)} Q$ then there are Γ'', Δ' such that $\Gamma''; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$ and $(\ell(d).l.(x)\Gamma') \in dom(\Gamma'')$ and $\Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma')\bot\Gamma''$.

Proof. By induction on length of the derivation of the transition $P \xrightarrow{l^d!(a)} Q$. We show the case when P is an output prefix, a recursion and a terminal recursion.

(*Case* $l^d!(a).P' \xrightarrow{l^d!(a)} P'$)

We have that:

$$\Gamma; \Delta \vdash_\ell l^d!(a).P' \ \ (i) \qquad \text{and} \qquad l^d!(a).P' \xrightarrow{l^d!(a)} P' \ \ (ii) \tag{A.4.8.1}$$

We have that (A.4.8.1)(i) is derived from:

$$(\ell(d).l.(x)\Gamma' \bot \Gamma); \Delta \vdash_\ell P' \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma) \tag{A.4.8.2}$$

and $wf(\Gamma, \Delta)$. From (A.4.8.2) and considering Proposition 6.3.5 we have:

$$\Gamma; \Delta \vdash_\ell P' \tag{A.4.8.3}$$

and $\Gamma \cup \bigcup \Delta \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case.

(*Case* $\textbf{rec}\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q$)

We have that:

$$\Gamma; \Delta \vdash_\ell \textbf{rec}\ \mathcal{X}.P \ \ (i) \qquad \text{and} \qquad \textbf{rec}\ \mathcal{X}.P \xrightarrow{l^d!(a)} Q \ \ (ii) \tag{A.4.8.4}$$

We have that (A.4.8.4)(i) is derived from:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P \tag{A.4.8.5}$$

which assuming $fv(\Gamma) = \emptyset$ (otherwise, since $fv(P) = \emptyset$ we use Lemma A.4.2 and trim down the ordering in such irrelevant names) and considering Lemma A.4.5 gives us that:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} \tag{A.4.8.6}$$

(A.4.8.4)(ii) is derived from:

$$P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} \xrightarrow{l^d!(a)} Q \tag{A.4.8.7}$$

By induction hypothesis we conclude:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell Q \tag{A.4.8.8}$$

and:

$$(\ell(d).l.(x)\Gamma') \in dom(\Gamma) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma) \tag{A.4.8.9}$$

and $\Gamma \cup \bigcup(\Delta, \mathcal{X} \to \Gamma) \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case.

(*Case* $\mathsf{rec}_t\ \mathcal{X}.P \xrightarrow{l^d?(a)} Q$)

We have that:

$$\Gamma; \Delta \vdash_\ell \mathsf{rec}_t\ \mathcal{X}.P\ \ (i) \qquad \text{and} \qquad \mathsf{rec}_t\ \mathcal{X}.P \xrightarrow{l^d!(a)} Q\ \ (ii) \tag{A.4.8.10}$$

We have that (A.4.8.10)(i) is derived from:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P \tag{A.4.8.11}$$

and $fv(\Delta(\mathcal{X})) = \emptyset$ and $wf(\Gamma, \Delta)$. Then considering Lemma A.4.5 we conclude:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} \tag{A.4.8.12}$$

(A.4.8.10)(ii) is derived from:

$$P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} \xrightarrow{l^d?(a)} Q \tag{A.4.8.13}$$

By induction hypothesis we conclude:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell Q \tag{A.4.8.14}$$

and:

$$(\ell(d).l.(x)\Gamma') \in dom(\Delta(\mathcal{X})) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Delta(\mathcal{X})) \tag{A.4.8.15}$$

and $\Delta(\mathcal{X}) \cup \bigcup \Delta \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case. ■

Lemma A.4.9

Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{c\ l^{\downarrow}!(a)} Q$ then there are Γ'', Δ' such

that $\Gamma''; \Delta, \Delta' \vdash_\ell Q$ *and* $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$ *and* $(c.l.(x)\Gamma') \in dom(\Gamma'')$ *and* $\Gamma'\{x/a\} \subseteq (c.l.(x)\Gamma')\bot\Gamma''$.

Proof. By induction on the derivation of the transition $P \stackrel{c\ l^{\downarrow}!(a)}{\longrightarrow} Q$. We show the base case.

($\mathit{Case}\ c \blacktriangleleft [P'] \stackrel{c\ l^{\downarrow}!(a)}{\longrightarrow} c \blacktriangleleft [Q']$)

We have that:

$$\Gamma; \Delta \vdash_\ell c \blacktriangleleft \left[P'\right] \quad (i) \qquad \text{and} \qquad c \blacktriangleleft \left[P'\right] \stackrel{c\ l^{\downarrow}!(a)}{\longrightarrow} c \blacktriangleleft \left[Q'\right] \quad (ii) \tag{A.4.9.1}$$

We have that (A.4.9.1)(i) is derived from:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),c)} P' \tag{A.4.9.2}$$

and (A.4.9.1)(ii) is derived from:

$$P' \stackrel{l^{\downarrow}!(a)}{\longrightarrow} Q' \tag{A.4.9.3}$$

From (A.4.9.2) and (A.4.9.3), considering Lemma A.4.8 we have:

$$\Gamma''; \Delta, \Delta' \vdash_{(\ell(\downarrow),c)} Q' \qquad \text{and} \qquad (c.l.(x)\Gamma') \in dom(\Gamma'') \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (c.l.(x)\Gamma'\bot\Gamma'') \tag{A.4.9.4}$$

and $\Gamma'' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$. From (A.4.9.4) we conclude:

$$\Gamma''; \Delta, \Delta' \vdash_\ell c \blacktriangleleft \left[Q'\right] \tag{A.4.9.5}$$

which completes the proof for this case. ∎

Lemma A.4.10

Let P *be a well-formed process such that* $\Gamma; \Delta \vdash_\ell P$. *If* $P \stackrel{(\nu a)l^d!(a)}{\longrightarrow} Q$ *then there are* $\Gamma'', \Delta', \Delta''$ *such that* $\Gamma'', \Delta', \Delta'' \vdash_\ell Q$ *and* $\Delta = \Delta' \setminus a$ *and* $\Gamma'' \setminus a \cup \bigcup(\Delta' \setminus a, \Delta'' \setminus a) \subseteq \Gamma \cup \bigcup \Delta$ *and* $(\ell(d).l.(x)\Gamma') \in dom(\Gamma'')$ *and* $\Gamma'\{x/a\} \subseteq ((\ell(d).l.(x)\Gamma')\bot\Gamma'')$.

Proof. By induction on the length of the derivation of the transition $P \stackrel{(\nu a)l^d!(a)}{\longrightarrow} Q$. We show the base case of restriction open (Figure 4.2 (*Open*)).

($\mathit{Case}\ (\boldsymbol{\nu} a)P' \stackrel{(\nu a)l^d!(a)}{\longrightarrow} Q'$)

We have that:

$$\Gamma; \Delta \vdash_\ell (\boldsymbol{\nu} a)P' \quad (i) \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' \stackrel{(\nu a)l^d!(a)}{\longrightarrow} Q' \quad (ii) \tag{A.4.10.1}$$

We have that (A.4.10.1)(i) is derived from:

$$\Gamma'; \Delta' \vdash_\ell P' \tag{A.4.10.2}$$

where $\Gamma = \Gamma' \setminus a$ and $\Delta = \Delta' \setminus a$. (A.4.10.1)$(ii)$ is derived from:

$$P' \stackrel{l^d!(a)}{\longrightarrow} Q' \tag{A.4.10.3}$$

From (A.4.10.2) and (A.4.10.3), considering Lemma A.4.8 we have:

$$\Gamma''; \Delta', \Delta'' \vdash_\ell Q' \qquad \text{and} \qquad (\ell(d).l.(x)\Gamma''') \in dom(\Gamma'') \qquad \text{and} \qquad \Gamma'''\{x/a\} \subseteq (\ell(d).l.(x)\Gamma''' \bot \Gamma'') \tag{A.4.10.4}$$

and $\Gamma'' \cup \bigcup(\Delta', \Delta'') \subseteq \Gamma' \cup \bigcup \Delta'$, thus, since $\Gamma = \Gamma' \setminus a$ and $\Delta = \Delta' \setminus a$ we have:

$$\Gamma'' \setminus a \cup \bigcup(\Delta' \setminus a, \Delta'' \setminus a) \subseteq \Gamma \cup \bigcup \Delta \tag{A.4.10.5}$$

which completes the proof. ∎

Lemma A.4.11
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q$ then there are $\Gamma'', \Delta', \Delta''$ such that $\Gamma'', \Delta', \Delta'' \vdash_\ell Q$ and $\Delta = \Delta' \setminus a$ and $\Gamma'' \setminus a \cup \bigcup(\Delta' \setminus a, \Delta'' \setminus a) \subseteq \Gamma \cup \bigcup \Delta$ and $(c.l.(x)\Gamma') \in dom(\Gamma'')$ and $\Gamma'\{x/a\} \subseteq ((c.l.(x)\Gamma')\bot\Gamma'')$.

Proof. By induction on the derivation of the transition $P \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q$. We show the cases of (*Open*) (Figure 4.2) and of a $(\nu a)l^{\downarrow}!(a)$ transition originating from a conversation piece.

(*Case* $c \blacktriangleleft [P'] \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} c \blacktriangleleft [Q']$)

We have that:

$$\Gamma; \Delta \vdash_\ell c \blacktriangleleft [P'] \ \ (i) \qquad \text{and} \qquad c \blacktriangleleft [P'] \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} c \blacktriangleleft [Q'] \ \ (ii) \tag{A.4.11.1}$$

We have that (A.4.11.1)(i) is derived from:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),c)} P' \tag{A.4.11.2}$$

and (A.4.11.1)(ii) is derived from:

$$P' \xrightarrow{(\nu a)l^{\downarrow}!(a)} Q' \tag{A.4.11.3}$$

From (A.4.11.2) and (A.4.11.3), considering Lemma A.4.10 we have that:

$$\Gamma'', \Delta', \Delta'' \vdash_{(\ell(\downarrow),c)} Q' \tag{A.4.11.4}$$

and $\Delta = \Delta' \setminus a$ and $\Gamma'' \setminus a \cup \bigcup(\Delta' \setminus a, \Delta'' \setminus a) \subseteq \Gamma \cup \bigcup \Delta$ and:

$$(c.l.(x)\Gamma' \in dom(\Gamma'')) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq (c.l.(x)\Gamma' \bot \Gamma'') \tag{A.4.11.5}$$

From (A.4.11.4) we conclude:

$$\Gamma''; \Delta', \Delta'' \vdash_\ell c \blacktriangleleft \left[Q'\right] \tag{A.4.11.6}$$

which completes the proof for this case.

(*Case* $(\boldsymbol{\nu} a)P' \xrightarrow{(\nu a)c\ l^{d}!(a)} Q'$)

We have that

$$\Gamma; \Delta \vdash_\ell (\boldsymbol{\nu} a)P' \ \ (i) \qquad \text{and} \qquad (\boldsymbol{\nu} a)P' \xrightarrow{(\nu a)c\ l^{\downarrow}!(a)} Q' \ \ (ii) \tag{A.4.11.7}$$

We have that (A.4.11.7)(i) is derived from:

$$\Gamma'; \Delta' \vdash_\ell P' \tag{A.4.11.8}$$

where $\Gamma = \Gamma' \setminus a$ and $\Delta = \Delta' \setminus a$. (A.4.11.7)$(ii)$ is derived from:

$$P' \xrightarrow{c\ l^{\downarrow}!(a)} Q' \tag{A.4.11.9}$$

From (A.4.11.8) and (A.4.11.9), considering Lemma A.4.9 we have that:

$$\Gamma'', \Delta', \Delta'' \vdash_\ell Q' \quad \text{and} \quad (c.l.(x)\Gamma''' \in dom(\Gamma'')) \quad \text{and} \quad \Gamma'''\{x/a\} \subseteq (c.l.(x)\Gamma''' \bot \Gamma'') \tag{A.4.11.10}$$

and $\Gamma'' \cup \bigcup(\Delta', \Delta'') \subseteq \Gamma' \cup \bigcup \Delta'$ from which, since $\Gamma = \Gamma' \setminus a$ and $\Delta = \Delta' \setminus a$, we conclude:

$$\Gamma'' \setminus a \cup \bigcup(\Delta' \setminus a, \Delta'' \setminus a) \subseteq \Gamma \cup \bigcup \Delta \tag{A.4.11.11}$$

which completes the proof for this case. ■

Lemma A.4.12
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q$ and $\ell(\downarrow) = \ell(\uparrow)$ then there are Γ', Δ' such that $\Gamma'; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q$. We show the case when the synchronization is derived via rule (*Comm*).

(*Case* $P_1 \mid P_2 \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q_1 \mid Q_2$)

We have that:

$$P_1 \mid P_2 \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q_1 \mid Q_2 \ \ (i) \quad \text{and} \quad \Gamma; \Delta \vdash_\ell P_1 \mid P_2 \ \ (ii) \tag{A.4.12.1}$$

From (A.4.12.1)(ii) we have:

$$\Gamma; \Delta \vdash_\ell P_1 \ \ (i) \quad \text{and} \quad \Gamma; \Delta \vdash_\ell P_2 \ \ (ii) \tag{A.4.12.2}$$

Let us consider (A.4.12.1)(i) is derived from:

$$P_1 \xrightarrow{l^{\downarrow}!(a)} Q_1 \ \ (i) \quad \text{and} \quad P_2 \xrightarrow{l^{\uparrow}?(a)} Q_2 \ \ (ii) \tag{A.4.12.3}$$

From Lemma A.4.8 and (A.4.12.2)(i) and (A.4.12.3)(i) we have:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma_1) \ \ (i) \quad \text{and} \quad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma') \bot \Gamma_1) \ \ (ii) \tag{A.4.12.4}$$

and:

$$\Gamma_1, \Delta, \Delta_1 \vdash_\ell Q_1 \tag{A.4.12.5}$$

and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$. Then, from (A.4.12.2)$(ii)$ and considering Proposition 6.3.5 we conclude:

$$\Gamma \cup \Gamma_1, \Delta, \Delta_1 \vdash_\ell P_2 \tag{A.4.12.6}$$

From (A.4.12.4) we directly have that:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma \cup \Gamma_1) \quad \text{and} \quad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma')\bot(\Gamma \cup \Gamma_1)) \tag{A.4.12.7}$$

From Lemma A.4.6 and (A.4.12.6) and (A.4.12.3)(ii) and (A.4.12.7) and since $\ell(\downarrow) = \ell(\uparrow)$ we have:

$$\Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.12.8}$$

and $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$. Then, considering Proposition 6.3.5, from (A.4.12.8) and (A.4.12.5) we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \quad \text{and} \quad \Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.12.9}$$

and hence:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \mid Q_2 \tag{A.4.12.10}$$

Also from $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$ and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$ we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2 \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \bigcup \Delta \tag{A.4.12.11}$$

which completes the proof for this case. ∎

Lemma A.4.13
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{c\ \mathtt{this}^\uparrow} Q$ and $\ell(\uparrow) = c$ then there are Γ', Δ' such that $\Gamma'; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the length of the derivation of the transition $P \xrightarrow{c\ \mathtt{this}^\uparrow} Q$. We show the case when the synchronization is derived via rule (*Comm*).

(*Case* $P_1 \mid P_2 \xrightarrow{c\ \mathtt{this}^\uparrow} Q_1 \mid Q_2$)

We have that:

$$P_1 \mid P_2 \xrightarrow{c\ \mathtt{this}^\uparrow} Q_1 \mid Q_2 \ \ (i) \quad \text{and} \quad \Gamma; \Delta \vdash_\ell P_1 \mid P_2 \ \ (ii) \tag{A.4.13.1}$$

From (A.4.13.1)(ii) we have:

$$\Gamma; \Delta \vdash_\ell P_1 \ \ (i) \quad \text{and} \quad \Gamma; \Delta \vdash_\ell P_2 \ \ (ii) \tag{A.4.13.2}$$

Let us consider (A.4.13.1)(i) is derived from:

$$P_1 \xrightarrow{c\ l^\downarrow!(a)} Q_1 \ \ (i) \quad \text{and} \quad P_2 \xrightarrow{l^\uparrow?(a)} Q_2 \ \ (ii) \tag{A.4.13.3}$$

From Lemma A.4.9 and (A.4.13.2)(i) and (A.4.13.3)(i) we have:

$$(c.l.(x)\Gamma') \in dom(\Gamma_1) \ \ (i) \quad \text{and} \quad \Gamma'\{x/a\} \subseteq ((c.l.(x)\Gamma')\bot\Gamma_1) \ \ (ii) \tag{A.4.13.4}$$

and:

$$\Gamma_1, \Delta, \Delta_1 \vdash_\ell Q_1 \tag{A.4.13.5}$$

and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$. Then, from (A.4.13.2)$(ii)$ and considering Proposition 6.3.5 we conclude:

$$\Gamma \cup \Gamma_1, \Delta, \Delta_1 \vdash_\ell P_2 \quad \text{(A.4.13.6)}$$

From (A.4.13.4) we directly have that:

$$(c.l.(x)\Gamma') \in dom(\Gamma \cup \Gamma_1) \quad \text{and} \quad \Gamma'\{x/a\} \subseteq ((c.l.(x)\Gamma')\bot(\Gamma \cup \Gamma_1)) \quad \text{(A.4.13.7)}$$

From Lemma A.4.6 and (A.4.13.6) and (A.4.13.3)(ii) and (A.4.13.7) and since $\ell(\uparrow) = c$ we have:

$$\Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \quad \text{(A.4.13.8)}$$

and $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$. Then, considering Proposition 6.3.5, from (A.4.13.8) and (A.4.13.5) we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \quad \text{and} \quad \Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \quad \text{(A.4.13.9)}$$

and hence:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \mid Q_2 \quad \text{(A.4.13.10)}$$

Also from $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$ and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$ we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2 \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \bigcup \Delta \quad \text{(A.4.13.11)}$$

which completes the proof for this case. ■

Lemma A.4.14
Let P be a well-formed process such that $\Gamma; \Delta \vdash_\ell P$. If $P \xrightarrow{c\ \mathtt{this}^\downarrow} Q$ and $\ell(\downarrow) = c$ then there are Γ', Δ' such that $\Gamma'; \Delta, \Delta' \vdash_\ell Q$ and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the derivation of the transition $P \xrightarrow{c\ \mathtt{this}^\downarrow} Q$. We show the base case when P is a **this** prefixed process and when the transition originates in a conversation piece.

($Case$ $\mathbf{this}(x).P' \xrightarrow{c\ \mathtt{this}^\downarrow} P'\{x/c\}$)

We have that:

$$\Gamma; \Delta \vdash_\ell \mathbf{this}(x).P' \ (i) \quad \text{and} \quad \mathbf{this}(x).P' \xrightarrow{c\ \mathtt{this}^\downarrow} P'\{x/c\} \ (ii) \quad \text{(A.4.14.1)}$$

and $\ell(\downarrow) = c$. We have that (A.4.14.1)(i) is derived from:

$$\Gamma \cup \Gamma'; \Delta \vdash_\ell P' \quad \text{(A.4.14.2)}$$

where $\Gamma'\{x/\ell(\downarrow)\} \subseteq \Gamma$, hence $\Gamma'\{x/c\} \subseteq \Gamma$. From Lemma 6.3.6 we then have:

$$\Gamma; \Delta \setminus x \vdash_\ell P'\{x/c\} \quad \text{(A.4.14.3)}$$

and considering Proposition 6.3.5 we conclude:

$$\Gamma; \Delta \vdash_\ell P'\{x/c\} \quad \text{(A.4.14.4)}$$

which completes the proof for this case.

($Case\ a \blacktriangleleft [P'] \xrightarrow{c\ \mathtt{this}^{\downarrow}} a \blacktriangleleft [Q']$)

We have that:

$$a \blacktriangleleft \left[P'\right] \xrightarrow{c\ \mathtt{this}^{\downarrow}} a \blacktriangleleft \left[Q'\right] \quad (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_{\ell} a \blacktriangleleft \left[P'\right] \quad (ii) \tag{A.4.14.5}$$

From (A.4.14.5)(ii) we have:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),a)} P' \tag{A.4.14.6}$$

(A.4.14.5)(i) is derived from either:

$$P' \xrightarrow{c\ \mathtt{this}^{\uparrow}} Q' \tag{A.4.14.7}$$

or:

$$P' \xrightarrow{\mathtt{this}^{\downarrow\uparrow}} Q' \qquad \text{and} \qquad c = a \tag{A.4.14.8}$$

($Case$ $\mathtt{this}^{\downarrow\uparrow}$) From Lemma A.4.12 and (A.4.14.6) and (A.4.14.8), $\ell(\downarrow) = c$ and $a = c$ we have:

$$\Gamma'; \Delta, \Delta' \vdash_{(\ell(\downarrow),c)} Q' \tag{A.4.14.9}$$

and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$. From (A.4.14.9) we derive:

$$\Gamma'; \Delta, \Delta' \vdash_{\ell} c \blacktriangleleft \left[Q'\right] \tag{A.4.14.10}$$

which completes the proof for this case.

($Case$ c $\mathtt{this}^{\uparrow}$) From $\ell(\downarrow) = c$ we have $\ell'(\uparrow) = c$ for $\ell' = (\ell(\downarrow), a)$. Then from Lemma A.4.13 and (A.4.14.6) and (A.4.14.7) we have:

$$\Gamma'; \Delta, \Delta' \vdash_{(\ell(\downarrow),a)} Q' \tag{A.4.14.11}$$

and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$. From (A.4.14.11) we derive:

$$\Gamma'; \Delta, \Delta' \vdash_{\ell} a \blacktriangleleft \left[Q'\right] \tag{A.4.14.12}$$

which completes the proof for this case. ∎

Theorem 6.3.7 (Preservation of Event Ordering)

(repetition of the statement in page 147)

Let process P be well-formed and $\Gamma; \Delta \vdash_{\ell} P$. If $P \to Q$ then $\Gamma'; \Delta, \Delta' \vdash_{\ell} Q$ and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the length of the derivation of the reduction $P \to Q$. We show the case for messages exchanged at the level of the current conversation, including when a bound name is carried in the message. The proof for the cases of located message exchanges follow analogous lines (notice the auxiliary lemmas for such cases are analogous). We also show the case of a τ originating from within a conversation piece, and the cases for the recursion and terminal recursion constructs. Remaining cases follow from induction hypothesis in expected lines.

($Case\ P_1 \mid P_2 \xrightarrow{\tau} Q_1 \mid Q_2$)

We have that:

$$P_1 \mid P_2 \xrightarrow{\tau} Q_1 \mid Q_2 \;\; (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_\ell P_1 \mid P_2 \;\; (ii) \tag{A.4.15.1}$$

From (A.4.15.1)(ii) we have:

$$\Gamma; \Delta \vdash_\ell P_1 \;\; (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_\ell P_2 \;\; (ii) \tag{A.4.15.2}$$

Let us consider (A.4.15.1)(i) is derived from:

$$P_1 \xrightarrow{l^{\downarrow}!(a)} Q_1 \;\; (i) \qquad \text{and} \qquad P_2 \xrightarrow{l^{\downarrow}?(a)} Q_2 \;\; (ii) \tag{A.4.15.3}$$

From Lemma A.4.8 and (A.4.15.2)(i) and (A.4.15.3)(i) we have:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma_1) \;\; (i) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma')\bot\Gamma_1) \;\; (ii) \tag{A.4.15.4}$$

and:

$$\Gamma_1, \Delta, \Delta_1 \vdash_\ell Q_1 \tag{A.4.15.5}$$

and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$. Then, from (A.4.15.2)$(ii)$ and considering Proposition 6.3.5 we conclude:

$$\Gamma \cup \Gamma_1, \Delta, \Delta_1 \vdash_\ell P_2 \tag{A.4.15.6}$$

From (A.4.15.4) we directly have that:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma \cup \Gamma_1) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma')\bot(\Gamma \cup \Gamma_1)) \tag{A.4.15.7}$$

From Lemma A.4.6 and (A.4.15.6) and (A.4.15.3)(ii) and (A.4.15.7) we have:

$$\Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.15.8}$$

and $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$. Then, considering Proposition 6.3.5, from (A.4.15.8) and (A.4.15.5) we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \qquad \text{and} \qquad \Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.15.9}$$

and hence:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta, \Delta_1, \Delta_2 \vdash_\ell Q_1 \mid Q_2 \tag{A.4.15.10}$$

Also from $\Gamma_2 \cup \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta, \Delta_1)$ and $\Gamma_1 \cup \bigcup(\Delta, \Delta_1) \subseteq \Gamma \cup \bigcup \Delta$ we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2 \bigcup(\Delta, \Delta_1, \Delta_2) \subseteq \Gamma \cup \bigcup \Delta \tag{A.4.15.11}$$

which completes the proof for this case.

($Case\ P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu} a)(Q_1 \mid Q_2)$)

We have that:

$$P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu} a)(Q_1 \mid Q_2) \;\; (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_\ell P_1 \mid P_2 \;\; (ii) \tag{A.4.15.12}$$

From (A.4.15.12)(ii) we have:

$$\Gamma;\Delta \vdash_\ell P_1\ \ (i) \qquad \text{and} \qquad \Gamma;\Delta \vdash_\ell P_2\ \ (ii) \tag{A.4.15.13}$$

Let us consider (A.4.15.12)(i) is derived from:

$$P_1 \xrightarrow{(\nu a)l^{\downarrow}!(a)} Q_1\ \ (i) \qquad \text{and} \qquad P_2 \xrightarrow{l^{\downarrow}?(a)} Q_2\ \ (ii) \tag{A.4.15.14}$$

From Lemma A.4.10 and (A.4.15.13)(i) and (A.4.15.14)(i) we have:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma_1)\ \ (i) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma')\bot\Gamma_1)\ \ (ii) \tag{A.4.15.15}$$

and:

$$\Gamma_1, \Delta', \Delta_1 \vdash_\ell Q_1 \tag{A.4.15.16}$$

and $\Delta = \Delta' \setminus a$ and $\Gamma_1 \setminus a \cup \bigcup(\Delta' \setminus a, \Delta_1 \setminus a) \subseteq \Gamma \cup \bigcup \Delta$. Then, from (A.4.15.13)(ii) and considering Proposition 6.3.5 we conclude:

$$\Gamma \cup \Gamma_1, \Delta', \Delta_1 \vdash_\ell P_2 \tag{A.4.15.17}$$

From (A.4.15.15) we directly have that:

$$(\ell(\downarrow).l.(x)\Gamma') \in dom(\Gamma \cup \Gamma_1) \qquad \text{and} \qquad \Gamma'\{x/a\} \subseteq ((\ell(\downarrow).l.(x)\Gamma')\bot(\Gamma \cup \Gamma_1)) \tag{A.4.15.18}$$

From Lemma A.4.6 and (A.4.15.17) and (A.4.15.14)(ii) and (A.4.15.18) we have:

$$\Gamma_2; \Delta', \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.15.19}$$

and $\Gamma_2 \cup \bigcup(\Delta', \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta', \Delta_1)$. Then, considering Proposition 6.3.5, from (A.4.15.19) and (A.4.15.16) we have:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta', \Delta_1, \Delta_2 \vdash_\ell Q_1 \qquad \text{and} \qquad \Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta', \Delta_1, \Delta_2 \vdash_\ell Q_2 \tag{A.4.15.20}$$

and hence:

$$\Gamma \cup \Gamma_1 \cup \Gamma_2; \Delta', \Delta_1, \Delta_2 \vdash_\ell Q_1 \mid Q_2 \tag{A.4.15.21}$$

from which we conclude:

$$\Gamma \cup \Gamma_1 \setminus a \cup \Gamma_2 \setminus a; \Delta, \Delta_1 \setminus a, \Delta_2 \setminus a \vdash_\ell (\boldsymbol{\nu} a)(Q_1 \mid Q_2) \tag{A.4.15.22}$$

Also since $\Delta = \Delta' \setminus a$ and $\Gamma = \Gamma \setminus a$ from $\Gamma_2 \cup \bigcup(\Delta', \Delta_1, \Delta_2) \subseteq \Gamma \cup \Gamma_1 \cup \bigcup(\Delta', \Delta_1)$ we have:

$$\Gamma_2 \setminus a \cup \bigcup(\Delta, \Delta_1 \setminus a, \Delta_2 \setminus a) \subseteq \Gamma \cup \Gamma_1 \setminus a \cup \bigcup(\Delta, \Delta_1 \setminus a) \tag{A.4.15.23}$$

which along with $\Gamma_1 \setminus a \cup \bigcup(\Delta, \Delta_1 \setminus a) \subseteq \Gamma \cup \bigcup \Delta$ we have:

$$\Gamma \cup \Gamma_1 \setminus a \cup \Gamma_2 \setminus a \cup \bigcup(\Delta, \Delta_1 \setminus a, \Delta_2 \setminus a) \subseteq \Gamma \cup \bigcup \Delta \tag{A.4.15.24}$$

which completes the proof for this case.

(*Case* $n \blacktriangleleft [P'] \xrightarrow{\tau} n \blacktriangleleft [Q']$)

We have that:

$$c \blacktriangleleft \left[P'\right] \xrightarrow{\tau} c \blacktriangleleft \left[Q'\right] \quad (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_{\ell} c \blacktriangleleft \left[P'\right] \quad (ii) \tag{A.4.15.25}$$

(A.4.15.25)(ii) is derived from:

$$\Gamma; \Delta \vdash_{(\ell(\downarrow),c)} P' \tag{A.4.15.26}$$

We have that (A.4.15.25)(i) is either derived from:

$$P' \overset{c\ \mathtt{this}^{\downarrow}}{\longrightarrow} Q' \tag{A.4.15.27}$$

or from:

$$P' \xrightarrow{\tau} Q' \tag{A.4.15.28}$$

In the latter case the result follows from induction hypothesis in expected lines. In the former case we have $\ell'(\downarrow) = c$ for $\ell' = (\ell(\downarrow), c)$. Then, considering Lemma A.4.14, from (A.4.15.26) and (A.4.15.27) we have:

$$\Gamma'; \Delta, \Delta' \vdash_{(\ell(\downarrow),c)} Q' \tag{A.4.15.29}$$

and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$. From (A.4.15.29) we derive:

$$\Gamma'; \Delta, \Delta' \vdash_{\ell} c \blacktriangleleft \left[Q'\right] \tag{A.4.15.30}$$

which completes the proof for this case.

(*Case* $\mathsf{rec}\ \mathcal{X}.P' \xrightarrow{\tau} Q'$)

We have that:

$$\mathsf{rec}\ \mathcal{X}.P' \xrightarrow{\tau} Q' \quad (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_{\ell} \mathsf{rec}\ \mathcal{X}.P' \quad (ii) \tag{A.4.15.31}$$

(A.4.15.31)(ii) is derived from:

$$\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma \vdash_{\ell} P' \tag{A.4.15.32}$$

From (A.4.15.32) and considering Lemma A.4.5 we conclude:

$$\Gamma; \Delta, \mathcal{X} \rightarrow \Gamma \vdash_{\ell} P'\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P'\} \tag{A.4.15.33}$$

We have that (A.4.15.31)(i) is derived from:

$$P'\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P'\} \xrightarrow{\tau} Q' \tag{A.4.15.34}$$

By induction hypothesis we conclude:

$$\Gamma'; \Delta, \mathcal{X} \rightarrow \Gamma, \Delta' \vdash_{\ell} Q' \tag{A.4.15.35}$$

and $\Gamma' \cup \bigcup(\Delta, \mathcal{X} \rightarrow \Gamma, \Delta') \subseteq \Gamma \cup \bigcup(\Delta, \mathcal{X} \rightarrow \Gamma) \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case.

(*Case* $\mathsf{rec}_t\ \mathcal{X}.P' \xrightarrow{\tau} Q'$)

We have that:

$$\textsf{rec}_t\ \mathcal{X}.P' \xrightarrow{\tau} Q' \ (i) \qquad \text{and} \qquad \Gamma; \Delta \vdash_\ell \textsf{rec}_t\ \mathcal{X}.P' \ (ii) \tag{A.4.15.36}$$

(A.4.15.36)(ii) is derived from:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P' \tag{A.4.15.37}$$

and $wf(\Gamma, \Delta)$. From (A.4.15.37) and considering Lemma A.4.5 we conclude:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P'\{\mathcal{X}/\textsf{rec}_t\ \mathcal{X}.P'\} \tag{A.4.15.38}$$

We have that (A.4.15.36)(i) is derived from:

$$P'\{\mathcal{X}/\textsf{rec}_t\ \mathcal{X}.P'\} \xrightarrow{\tau} Q' \tag{A.4.15.39}$$

By induction hypothesis we conclude:

$$\Gamma'; \Delta, \Delta' \vdash_\ell Q' \tag{A.4.15.40}$$

and $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Delta(\mathcal{X}) \cup \bigcup \Delta \subseteq \Gamma \cup \bigcup \Delta$ which completes the proof for this case. ∎

We introduce some notions auxiliary to the proof of Theorem 6.3.10.

Notation A.4.16 *We say M^w is an initial message type of T if $T \equiv \oplus\{p\, l^d(C).B; \tilde{B}\} \mid T'$ and $M = p\, l^d(C)$ and $w = d$, or $T \equiv \&\{p\, l^d(C).B; \tilde{B}\} \mid T'$ and $M = p\, l^d(C)$ and $w = d$, or $T \equiv c : [\oplus\{M.B; \tilde{B}\}] \mid T'$ and $w = c$, or $T \equiv c : [\&\{M.B; \tilde{B}\}] \mid T'$ and $w = c$.*

Notation A.4.17 *We denote by $p_1\, l_1^d(C_1)^d \prec_\Gamma^\ell p_2\, l_2^{d'}(C_2)^{d'}$ that $\ell(d).l_1.(x)\Gamma' \prec_\Gamma \ell(d').l_2.(x)\Gamma''$ and by $p_1\, l_1^d(C_1)^a \prec_\Gamma^\ell p_2\, l_2^{d'}(C_2)^b$ that $a.l_1.(x)\Gamma' \prec_\Gamma b.l_2.(x)\Gamma''$.*

Notation A.4.18 *We denote by $P \xrightarrow{M^w} P'$ a transition $P \xrightarrow{\lambda} P'$ such that either:*

- *$\lambda = l^d!(a)$ (or $\lambda = (\nu a)l^d!(a)$) and $M = \,!\, l^d(C)$ and $w = d$ for some a, C*
- *$\lambda = l^d?(a)$ and $M = \,?\, l^d(C)$ and $w = d$ for some a, C*
- *$\lambda = c\ l^\downarrow!(a)$ (or $\lambda = (\nu a)c\ l^\downarrow!(a)$) and $M = \,!\, l^\downarrow(C)$ and $w = c$ for some a, C*
- *$\lambda = c\ l^\downarrow?(a)$ and $M = \,?\, l^\downarrow(C)$ and $w = c$ for some a, C*
- *$\lambda = \tau$ and $M = \boldsymbol{\tau}\, l^d(C)$*

Notation A.4.19 *We say M^w is an minimal initial message of T w.r.t. Γ, ℓ if M^w is an initial message of T and there is no $M'^{w'}$ initial message type of T such that $M'^{w'} \prec_\Gamma^\ell M^w$.*

Lemma A.4.20
If M^w be an initial message type of T and $T = T_1 \bowtie T_2$ then M^w is an initial message type of T_1 or M^w is an initial message type of T_2 or $M = \boldsymbol{\tau}\, l^d(C)$ and $p\, l^d(C)^w$ is an initial message of

T_k for $p \in \{?, !\}$ and $k \in \{1, 2\}$.

Proof. By induction on the length of the derivation of $T = T_1 \bowtie T_2$. Cases (*SharedInp*), (*Shared*), (*Plain*), (*Rec*), (*Var*), (*Apart*) and (*Par*) follow in expected lines. We show the cases for rules (*Shuffle*) and (*ShufflePar*).

(Case (*Shuffle*))

We have that:

$$B' \mid M'.B'' = B_1 \bowtie M'.B_2 \tag{A.4.20.1}$$

derived from:

$$M' \# B_1 \quad B' \mid B'' \equiv B_1 \bowtie B_2 \quad M' \# B' \quad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2) \quad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \tag{A.4.20.2}$$

We have that M^w is an initial message type of $B' \mid M'.B''$, hence either M^w is an initial message type of B' or of $M'.B''$. In the latter case we directly have that $M = M'$ and the proof is complete. In the former case, by induction hypothesis we have that either M^w is an initial message type of B_1 or M^w is an initial message type of B_2 or $M^w = \tau\, l^d(C)$ and $p\, l^d(C)$ is an initial message of B_1 or of B_2. Since $\mathcal{I}(B') \subseteq \mathcal{I}(B_1)$ we conclude that M^w is an initial message type of B_1 or $M^w = \tau\, l^d(C)$ and $p\, l^d(C)$ is an initial message type of B_1.

(Case (*ShufflePar*))

We have that:

$$B = B_1 \bowtie B_2 \mid B_3 \tag{A.4.20.3}$$

derived from:

$$B_1 \not\# B_2 \quad B' = B_1 \bowtie B_2 \quad B = B' \bowtie B_3 \tag{A.4.20.4}$$

We have that M^w is an initial message type of B. By induction hypothesis on $B = B' \bowtie B_3$ we conclude that either (1) M^w is an initial message type of B' or (2) of B_3 or (3) $M^w = \tau\, l^d(C)$ and $p\, l^d(C)$ is an initial message type of B' or B_3.

(1) We have that M^w is an initial message type of B'. By induction hypothesis on $B' = B_1 \bowtie B_2$ we conclude that either M^w is an initial message type of B_1 or of B_2 or $M^w = \tau\, l^d(C)$ and $p\, l^d(C)$ is an initial message type of B_1 or of B_2. In any case the result is immediate.

(2) Immediate.

(3) We have that $M^w = \tau\, l^d(C)$. If $p\, l^d(C)$ is an initial message type of B_3 then the result is immediate. If $p\, l^d(C)$ is an initial message type of B' then, by induction hypothesis and since $p \in \{!, ?\}$, we have that either $p\, l^d(C)$ is an initial message type of B_1 or of B_2 which completes the proof for this case. ∎

Notation A.4.21 *Given message type M and type T we write $M^w \in T$ to denote that $T \equiv n : [B] \mid T'$ and $M \in Msg_{\mathcal{L}}(B)$ if $w = n$, or $T \equiv L \mid B$ and $M \in Msg_{\mathcal{L}}(B)$ if $w = d$.*

Definition A.4.22 (Type and Event Ordering Domain Consistency)
We say type T and event ordering Γ are domain consistent with respect to ℓ if for all $p\, l^d(C)^w \in T$ we have that $n.l.(x)\Gamma' \in dom(\Gamma)$, for $n = \ell(d)$ if $w = d$ and $n = w$ otherwise, and $x : C$ and Γ' are domain consistent with respect to ℓ.

Notation A.4.23 *We say that M^d and Γ are domain consistent with respect to ℓ if M and Γ are domain consistent with respect to ℓ, and that M^n and Γ are domain consistent with respect to ℓ if $n : [M]$ and Γ are domain consistent with respect to ℓ.*

Lemma A.4.24
Let P be a process such that $P :: T$ and $\Gamma; \Delta \vdash_\ell P$ and for all $!\, l^d(C)^w \in T$ we have that $!\, l^d(C)^w$ and Γ are domain consistent with respect to ℓ. Then we have that T and Γ are domain consistent with respect to ℓ.
N.B. We assume all input message types are introduced by rule (Input).

Proof. By induction on the length of the derivation of $P :: T$. We show the cases for (*Par*), (*Output*) and (*Input*).

(Case (*Par*))

We have that:

$$P \mid Q :: T_1 \bowtie T_2 \tag{A.4.24.1}$$

and:

$$\Gamma; \Delta \vdash_\ell P \mid Q \tag{A.4.24.2}$$

and for all $!\, l^d(C)^w \in T_1 \bowtie T_2$ we have that $!\, l^d(C)^w$ and Γ are domain consistent with respect to ℓ. $(A.4.24.1)$ is derived from:

$$P :: T_1 \qquad Q :: T_2 \tag{A.4.24.3}$$

and $(A.4.24.2)$ is derived from:

$$\Gamma; \Delta \vdash_\ell P \qquad \Gamma; \Delta \vdash_\ell Q \tag{A.4.24.4}$$

Let us assume that for all $!\, l^d(C)^w \in T_1$ we have that $!\, l^d(C)^w$ and Γ are domain consistent with respect to ℓ. Then by induction hypothesis we have that T_1 and Γ are domain consistent with respect to ℓ. Likewise for T_2.

Let us take $?\, l^d(C)^w$ such that:

$$?\, l^d(C)^w \in T_1 \bowtie T_2 \tag{A.4.24.5}$$

We then have that either $?\, l^d(C)^w \in T_1$ or $?\, l^d(C)^w \in T_2$, from which, considering T_1 (and T_2) and Γ are domain consistent w.r.t. ℓ, gives us that $?\, l^d(C)^w$ and Γ are domain consistent.

Let us now take $\tau\, l^d(C)^w$ such that:

$$\tau\, l^d(C)^w \in T_1 \bowtie T_2 \tag{A.4.24.6}$$

We then have that either $\tau\, l^d(C)^w \in T_1$ or $\tau\, l^d(C)^w \in T_2$ or $!\, l^d(C)^w \in T_i$ and $?\, l^d(C)^w \in T_j$, for $\{i, j\} = \{1, 2\}$. The first two cases follow from the fact that T_1 (and T_2) and Γ are domain consistent w.r.t. ℓ. For the third and last case we have that T_j (where j in $\{1, 2\}$) and Γ are domain consistent w.r.t. ℓ, and hence $?\, l^d(C)^w$ and Γ are domain consistent which directly gives us that $\tau\, l^d(C)^w$ and Γ are domain consistent and completes the proof for this case.

(Case (*Input*))

We have that:

$$\Sigma_{i\in I}\ l_i^d?(x_i).P_i :: L \mid \&_{i\in I}\{?\, l_i^d(C_i).B_i\} \tag{A.4.24.7}$$

and:

$$\Gamma; \Delta \vdash_\ell \Sigma_{i\in I}\ l_i^d?(x_i).P_i \tag{A.4.24.8}$$

and for all $!\, l^d(C)^w \in L \mid \&_{i\in I}\{?\, l_i^d(C_i).B_i\}$ we have that $!\, l^d(C)^w$ and Γ are domain consistent with respect to ℓ. (A.4.24.7) is derived from:

$$\forall_{i\in I}(P_i :: L \mid B_i \mid x_i : C_i) \tag{A.4.24.9}$$

and (A.4.24.8) is derived from:

$$(\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma) \cup \Gamma_i'\{y_i/x_i\}; \Delta \vdash_\ell P_i \qquad \Gamma_i' \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma) \tag{A.4.24.10}$$

Let us assume that for all $!\, l^d(C)^w \in L \mid B_i \mid x_i : C_i$ we have that $!\, l^d(C)^w$ and $(\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma) \cup \Gamma_i'\{y_i/x_i\}$ are domain consistent with respect to ℓ. Then by induction hypothesis we have that $L \mid B_i \mid x_i : C_i$ and $(\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma) \cup \Gamma_i'\{y_i/x_i\}$ are domain consistent with respect to ℓ, from which we have that $L \mid B_i$ and $(\ell(d).l_i.(y_i)\Gamma_i' \bot \Gamma)$ are domain consistent with respect to ℓ. We conclude that $L \mid \&_{i\in I}\{?\, l_i^d(C_i).B_i\}$ and Γ are domain consistent.

(Case (*Output*))

We have that:

$$l^d!(n).P :: (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\} \tag{A.4.24.11}$$

and:

$$\Gamma; \Delta \vdash_\ell l^d!(n).P \tag{A.4.24.12}$$

and for all $!\, l^d(C)^w \in (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\}$ we have that $!\, l^d(C)^w$ and Γ are domain consistent with respect to ℓ. (A.4.24.11) is derived from:

$$P :: L \mid B \tag{A.4.24.13}$$

and (A.4.24.12) is derived from:

$$(\ell(d).l.(x)\Gamma' \bot \Gamma); \Delta \vdash_\ell P\ \ (i) \qquad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma)\ \ (ii) \tag{A.4.24.14}$$

By induction hypothesis we conclude $L \mid B$ and $\ell(d).l.(x)\Gamma' \bot \Gamma$ are domain consistent. Since $!\, l^d(C)^d$ and Γ are domain consistent we conclude that $\ell(d).l.(x)\Gamma' \in \mathit{dom}(\Gamma)$ and that $x : C$ and Γ' are domain consistent. We then have that $n : C$ and $\Gamma'\{x/n\}$ are domain consistent, and hence, considering (A.4.24.14)(ii), we conclude $n : C$ and $\ell(d).l.(x)\Gamma' \bot \Gamma$ are domain consistent.

Let us take $?\, l^d(C)^w$ such that:

$$?\, l^d(C)^w \in (L \bowtie n : C) \mid \oplus\{!\, l^d(C).B; \tilde{B}\} \tag{A.4.24.15}$$

We then have that either $?\, l^d(C)^w \in L \bowtie n : C$ or $?\, l^d(C)^w \in \oplus\{!\, l^d(C).B; \tilde{B}\}$. In the former case we have that either $?\, l^d(C)^w \in L$ or $?\, l^d(C)^w \in n : C$, which since L (and $n : C$) and Γ are domain consistent, we have that $?\, l^d(C)^w$ and Γ are domain consistent. In the latter case we have that $?\, l^d(C)^w \in \oplus\{!\, l^d(C).B; \tilde{B}\}$, which, since every input type occurs in B and B and Γ

are domain consistent gives us that $?\,l^d(C)^w$ and Γ are domain consistent.

Let us now take $\tau\,l^d(C)^w$ such that:

$$\tau\,l^d(C)^w \in (L \bowtie n : C) \mid \oplus\{!\,l^d(C).B; \tilde{B}\} \tag{A.4.24.16}$$

We then have that either $\tau\,l^d(C)^w \in L$ or $\tau\,l^d(C)^w \in n : C$ or $\tau\,l^d(C)^w \in \oplus\{!\,l^d(C).B; \tilde{B}\}$ or $p_1\,l^d(C)^w \in L$ and $p_2\,l^d(C)^w \in n : C$, for $\{p_1, p_2\} = \{?, !\}$. The first two cases follow from the fact that L (and $n : C$) and Γ are domain consistent w.r.t. ℓ. The third case follows from the fact that $\tau\,l^d(C)^w \in B$ and B and Γ are domain consistent. For the fourth and last case we have that T_j (where j in $\{1,2\}$) and Γ are domain consistent w.r.t. ℓ, and hence $?\,l^d(C)^w$ and Γ are domain consistent which directly gives us that $\tau\,l^d(C)^w$ and Γ are domain consistent and completes the proof for this case. ∎

Definition A.4.25 (Type Precedence)
We say type T justifies the precedence between $n.l_1$ and $n.l_2$ w.r.t. ℓ, noted by $T \blacktriangleright_\ell n.l_1 \rightsquigarrow n.l_2$, if $T \equiv n : [\{M.B; \tilde{B}\}] \mid T'$ and either $l_1^{\downarrow} \in dLab_{\mathcal{L}}(M)$ and $l_2^{\downarrow} \in dLab_{\mathcal{L}}(B)$, or $n : [B] \blacktriangleright_\ell n.l_1 \rightsquigarrow n.l_2$. Likewise we say type T justifies the precedence between $\ell(d).l_1$ and $\ell(d').l_2$ w.r.t. ℓ, denoted by $T \blacktriangleright_\ell \ell(d).l_1 \rightsquigarrow \ell(d').l_2$, if $T \equiv \{M.B; \tilde{B}\}] \mid T'$ and either $l_1^d \in dLab_{\mathcal{L}}(M)$ and $l_2^{d'} \in dLab_{\mathcal{L}}(B)$, or $B \blacktriangleright_\ell \ell(d).l_1 \rightsquigarrow \ell(d').l_2$.
N.B. We consider $\rightsquigarrow$ does not cross recursive type constructs and that types are maximum folded.

Definition A.4.26 (Type and Event Ordering Precedence Consistency)
We say type T and event ordering Γ are precedence consistent with respect to ℓ, if $T \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2$ implies $n_1.l_1.(x)\Gamma_1 \prec_\Gamma n_2.l_2.(y)\Gamma_2$.

Lemma A.4.27
Let T be a type such that $T = T_1 \bowtie T_2$ and $T \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$. Then $T_{i_1} \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2$ and $T_{i_2} \blacktriangleright_\ell n_2.l_2 \rightsquigarrow n_3.l_3$ and $\ldots$ and $T_{i_{k-1}} \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k$, where $i_1, \ldots, i_{k-1} \in \{1,2\}$.

Proof. By induction on the length of the derivation of $T = T_1 \bowtie T_2$. We show the cases for rules (*Shuffle*) and (*ShufflePar*).

(Case (*Shuffle*))

We have that $B' \mid M.B'' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$ and:

$$B' \mid M.B'' = B_1 \bowtie M.B_2 \tag{A.4.27.1}$$

which is derived from:

$$M \# B_1 \quad B' \mid B'' \equiv B_1 \bowtie B_2 \quad M \# B' \quad \mathcal{I}(B'') \subseteq \mathcal{I}(B_2) \quad \mathcal{I}(B') \subseteq \mathcal{I}(B_1) \tag{A.4.27.2}$$

We have that either (1) $B' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$ or (2) $M.B'' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$.

(Case (1)) By induction hypothesis on $B' \mid B'' \equiv B_1 \bowtie B_2$ we conclude that:

$$B_1 \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1} \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.3}$$

where $\{B_1, \ldots, B_{k-1}\} = \{B_1, B_2\}$, which directly gives us that:

$$B_1' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1}' \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.4}$$

where $\{B_1', \ldots, B_{k-1}'\} = \{B_1, M.B_2\}$.

(Case (2)) We have that either $n_1 = \ell(d)$ and $M = pl_1^d(C)$ or $B'' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$. The latter case follows lines similar to (1). In the former case we have that either $n_k.l_k$ is in the initial actions of B'' or not. If $n_k.l_k$ is in the initial actions of B'' since $\mathcal{I}(B'') \subseteq \mathcal{I}(B_2)$ we have that $n_k.l_k$ is in the initial actions of B_2 and hence $M.B_2 \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$. If $n_k.l_k$ is not in the initial actions of B'' we take $o.s$ in the initial actions of B'' such that $B'' \blacktriangleright_\ell o.s \rightsquigarrow n_k.l_k$. By induction hypothesis on $B' \mid B'' \equiv B_1 \bowtie B_2$ we conclude that:

$$B_1 \blacktriangleright_\ell o.s \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1} \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.5}$$

where $\{B_1, \ldots, B_{k-1}\} = \{B_1, B_2\}$, which gives us that:

$$B_1' \blacktriangleright_\ell o.s \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1}' \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.6}$$

where $\{B_1', \ldots, B_{k-1}'\} = \{B_1, M.B_2\}$. Since $o.s$ is in the initial actions of B'' and $\mathcal{I}(B'') \subseteq \mathcal{I}(B_2)$ we have that $o.s$ is in the initial actions of B_2, and hence:

$$M.B_2 \blacktriangleright n_1.l_1 \rightsquigarrow o.s \wedge B_1' \blacktriangleright_\ell o.s \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1}' \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.7}$$

which completes the proof for this case.

(Case (*ShufflePar*))

We have that $B \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$ and:

$$B = B_1 \bowtie B_2 \mid B_3 \tag{A.4.27.8}$$

which is derived from:

$$B_1 \# B_2 \qquad B' = B_1 \bowtie B_2 \qquad B = B' \bowtie B_3 \tag{A.4.27.9}$$

By induction hypothesis on $B \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_k.l_k$ and $B = B' \bowtie B_3$ we have that:

$$B_1 \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \wedge \ldots \wedge B_{k-1} \blacktriangleright_\ell n_{k-1}.l_{k-1} \rightsquigarrow n_k.l_k \tag{A.4.27.10}$$

where $\{B_1, \ldots, B_{k-1}\} = \{B', B_3\}$. Let us take $i \in 1, \ldots, k-1$ such that $B_i = B'$. By induction hypothesis on $B' \blacktriangleright_\ell n_i.l_i \rightsquigarrow n_{i+1}.l_{i+1}$ and $B' = B_1 \bowtie B_2$ we conclude:

$$B_1' \blacktriangleright_\ell n_i.l_i \rightsquigarrow o_2.s_2 \wedge \ldots \wedge B_{t-1}' \blacktriangleright_\ell o_{t-1}.s_{t-1} \rightsquigarrow n_{i+1}.l_{i+1} \tag{A.4.27.11}$$

where $\{B_1', \ldots, B_{t-1}'\} = \{B_1, B_2\}$. Since we can reproduce this reasoning for each $i \in 1, \ldots, k-1$ such that $B_i = B'$ we have there is a path that leads from $n_1.l_1$ to $n_k.l_k$ justified by either B_3, B_1 or B_2 and hence:

$$B_1'' \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_{i_1}.l_{i_1} \wedge \ldots \wedge B_r'' \blacktriangleright_\ell n_{i_{r-1}}.l_{i_{r-1}} \rightsquigarrow n_k.l_k \tag{A.4.27.12}$$

where $\{B''_1, \ldots, B''_r\} = \{B_1, B_2 \mid B_3\}$ which completes the proof for this case. ∎

Lemma A.4.28

Let P be a process such that $P :: T$, $\Gamma; \Delta \vdash_\ell P$, T and Γ are domain consistent, and for all $!\, l^d(C)^w \in T$ and $w.l.(x)\Gamma' \in dom(\Gamma)$ we have that $x : C \blacktriangleright_\ell o_1.s_1 \rightsquigarrow o_2.s_2$ implies $o_1.s_1.(x)\Gamma'_1 \prec_{\Gamma'} o_2.s_2.(x)\Gamma'_2$. Then T and Γ are precedence consistent with respect to ℓ.

Proof. By induction on the length of the derivation of $P :: T$. We show the cases for rules (*Par*), (*Input*) and (*Output*).

(Case (*Par*))

We have that:

$$P \mid Q :: T_1 \bowtie T_2 \qquad \text{(A.4.28.1)}$$

and:

$$\Gamma; \Delta \vdash_\ell P \mid Q \qquad \text{(A.4.28.2)}$$

and:

$$T_1 \bowtie T_2 \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \qquad \text{(A.4.28.3)}$$

(A.4.28.1) is derived from:

$$P :: T_1 \qquad Q :: T_2 \qquad \text{(A.4.28.4)}$$

and (A.4.28.2) is derived from:

$$\Gamma; \Delta \vdash_\ell P \qquad \Gamma; \Delta \vdash_\ell Q \qquad \text{(A.4.28.5)}$$

From (A.4.28.3) and considering Lemma A.4.27 we have that $T_{i_1} \blacktriangleright_\ell n_1.l_1 \rightsquigarrow o_1.s_1$ and ... and $T_{i_k} \blacktriangleright_\ell o_{k-1}.s_{k-1} \rightsquigarrow n_2.l_2$, where $i_1, \ldots, i_k \in \{1, 2\}$. By induction hypothesis on (A.4.28.4) and (A.4.28.5), for any $j \in 2, \ldots, k-1$, we conclude that $o_{j-1}.s_{j-1}.(x)\Gamma_{j-1} \prec_\Gamma o_j.s_j.(x)\Gamma_j$ and also that $n_1.l_1.(x)\Gamma' \prec_\Gamma o_1.s_1.(x)\Gamma'$ and $o_{k-1}.s_{k-1}.(x)\Gamma_{k-1} \prec_\Gamma n_2.l_2.(x)\Gamma''$, and hence we have that $n_1.l_1.(x)\Gamma' \prec_\Gamma n_2.l_2.(x)\Gamma''$ which completes the proof for this case.

(Case (*Input*))

We have that:

$$\Sigma_{i \in I}\, l_i^d(x_i).P_i :: L \mid \&_{i \in I}\{?\, l_i^d(C_i).B_i\} \qquad \text{(A.4.28.6)}$$

and:

$$\Gamma; \Delta \vdash_\ell \Sigma_{i \in I}\, l_i^d(x_i).P_i \qquad \text{(A.4.28.7)}$$

and:

$$L \mid \&_{i \in I}\{?\, l_i^d(C_i).B_i\} \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \qquad \text{(A.4.28.8)}$$

(A.4.28.6) is derived from:

$$P_i :: L \mid B_i \mid x_i : C_i \qquad \text{(A.4.28.9)}$$

and (A.4.28.7) is derived from:

$$(\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y_i/x_i\}; \Delta \vdash_\ell P_i \qquad \Gamma'_i \setminus y_i \subseteq (\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \qquad \text{(A.4.28.10)}$$

From (A.4.28.8) we conclude that either $L \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2$ or $\&_{i \in I}\{?\, l_i^d(C_i).B_i\} \blacktriangleright_\ell n_1.l_1 \rightsquigarrow$

$n_2.l_2$. We prove the latter case, as the former follows by induction hypothesis in expected lines. We have that there is j such that either (1) $B_j \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2$, or (2) $n_1 = \ell(d)$ and $l_2 = l_j$.

(1) Follows by induction hypothesis in expected lines.

(2) Let us take $o.s$ in the initial messages of B_j, for which we either have $B_j \blacktriangleright_\ell o.s \rightsquigarrow n_2.l_2$ or $o = n_2$ and $s = l_2$. We prove the former case as the latter follows similar lines, using the domain consistency hypothesis. In the former case by induction hypothesis we have that:

$$o.s.(x)\Gamma' \prec_{(\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y_i/x_i\}} n_2.l_2.(x)\Gamma_2 \tag{A.4.28.11}$$

from which we conclude:

$$o.s.(x)\Gamma' \prec_\Gamma n_2.l_2.(x)\Gamma_2 \tag{A.4.28.12}$$

Since $o.s.(x)\Gamma' \in dom(\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma) \cup \Gamma'_i\{y_i/x_i\}$ and $o \neq x_i$ we have that $o.s.(x)\Gamma' \in dom(\ell(d).l_i.(y_i)\Gamma'_i \perp \Gamma)$ and hence:

$$n_1.l_1.(x)\Gamma_1 \prec_\Gamma o.s.(x)\Gamma' \tag{A.4.28.13}$$

From (A.4.28.12) and (A.4.28.13) we conclude:

$$n_1.l_1.(x)\Gamma_1 \prec_\Gamma n_2.l_2.(x)\Gamma_2 \tag{A.4.28.14}$$

which completes the proof for this case.

(Case (*Output*))

We have that:

$$l^d!(n).P :: (L \bowtie n : C) \mid \oplus\{!\ l^d(C).B; \tilde{B}\} \tag{A.4.28.15}$$

and:

$$\Gamma; \Delta \vdash_\ell l^d!(n).P \tag{A.4.28.16}$$

and:

$$(L \bowtie n : C) \mid \oplus\{!\ l^d(C).B; \tilde{B}\} \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2 \tag{A.4.28.17}$$

(A.4.28.15) is derived from:

$$P :: L \mid B \tag{A.4.28.18}$$

and (A.4.28.16) is derived from:

$$(\ell(d).l.(x)\Gamma' \perp \Gamma); \Delta \vdash_\ell P \qquad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \perp \Gamma) \tag{A.4.28.19}$$

From (A.4.28.17) we have that either $L \bowtie n : C \blacktriangleright_\ell n_1.l_1 \rightsquigarrow n_2.l_2$ or $\{!\ l^d(C).B; \tilde{B}\} \rightsquigarrow n_2.l_2$. We show the proof for the first case. Considering Lemma A.4.27 we conclude that:

$$T_1 \blacktriangleright_\ell n_1.l_1 \rightsquigarrow o_1.s_1 \wedge \ldots \wedge T_k \blacktriangleright_\ell o_{k-1}.s_{k-1} \rightsquigarrow n_2.l_2 \tag{A.4.28.20}$$

where $T_1, \ldots, T_k \in \{L, n : C\}$. For each i we have that either $T_i = L$, in which case by induction hypothesis we have:

$$o_{i-1}.s_{i-1}.(x)\Gamma'_{i-1} \prec_{(\ell(d).l.(x)\Gamma' \perp \Gamma)} o_i.s_i.(x)\Gamma'_i \tag{A.4.28.21}$$

and hence $o_{i-1}.s_{i-1}.(x)\Gamma'_{i-1} \prec_\Gamma o_i.s_i.(x)\Gamma'_i$, or $T_i = n : C$, which considering $!\ l^d(C)^d \in T$ and

$\ell(d).l.(x)\Gamma' \in dom(\Gamma)$ gives us that $x : C \blacktriangleright_\ell x.s_{i-1} \rightsquigarrow x.s_i$ implies:

$$x.s_{i-1}.(x)\Gamma'_{i-1} \prec_{\Gamma'} x.s_i.(x)\Gamma'_i \tag{A.4.28.22}$$

from which we have that $n : C \blacktriangleright_\ell n.s_{i-1} \rightsquigarrow n.s_i$ implies:

$$n.s_{i-1}.(x)\Gamma'_{i-1} \prec_{\Gamma'\{x/n\}} n.s_i.(x)\Gamma'_i \tag{A.4.28.23}$$

and hence, since $\Gamma'\{x/n\} \subseteq \Gamma$, we have $n.s_{i-1}.(x)\Gamma'_{i-1} \prec_\Gamma n.s_i.(x)\Gamma'_i$. We then have that:

$$n_1.l_1.(x)\Gamma_1 \prec_\Gamma o_1.s_1.(x)\Gamma'_1 \prec_\Gamma \ldots \prec_\Gamma o_{k-1}.s_{k-1}.(x)\Gamma'_{k-1} \prec_\Gamma n_2.l_2.(x)\Gamma_2 \tag{A.4.28.24}$$

and hence $n_1.l_1.(x)\Gamma_1 \prec_\Gamma n_2.l_2.(x)\Gamma_2$ which completes the proof for this case. ■

Lemma A.4.29
Let P be a process such that $\Gamma; \Delta \vdash_\ell P$. We have that there is $\Gamma_1 \subseteq \Gamma$ such that $\Gamma_1; \Delta \vdash_\ell P$ and there is no $\Gamma_2 \subset \Gamma_1$ such that $\Gamma_2; \Delta \vdash_\ell P$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_\ell P$.

(*Case* (*Par*))

We have that $\Gamma; \Delta \vdash_\ell P \mid Q$ derived from $\Gamma; \Delta \vdash_\ell P$ and $\Gamma; \Delta \vdash_\ell Q$. By induction hypothesis we have that there is Γ_1 such that $\Gamma_1; \Delta \vdash_\ell P$ and $\Gamma_1 \subseteq \Gamma$ and there is no Γ_2 such that $\Gamma_2 \subset \Gamma_1$ and $\Gamma_2; \Delta \vdash_\ell P$. Likewise for Q we have that there is Γ_3 such that $\Gamma_3; \Delta \vdash_\ell Q$ such that $\Gamma_3 \subseteq \Gamma$ and there is no Γ_4 such that $\Gamma_4 \subset \Gamma_3$ and $\Gamma_4; \Delta \vdash_\ell Q$. From $\Gamma_1 \subseteq \Gamma$ and $\Gamma_3 \subseteq \Gamma$ we have that $wf(\Gamma_1 \cup \Gamma_3, \Delta)$. Then, considering Proposition 6.3.5 we conclude $\Gamma_1 \cup \Gamma_3; \Delta \vdash_\ell P$ and $\Gamma_1 \cup \Gamma_3; \Delta \vdash_\ell Q$ and thus $\Gamma_1 \cup \Gamma_3; \Delta \vdash_\ell P \mid Q$, which completes the proof for this case.

(*Cases* (*Stop*), (*RecVar*) (*Rec*) *and* (*RecTerm*))

Follow expected lines.

(*Case* (*Res*))

We have that $\Gamma; \Delta \vdash_\ell (\boldsymbol{\nu} a)P$ derived from $\Gamma_1; \Delta' \vdash_\ell P$ where $\Gamma = \Gamma_1 \setminus a$ and $\Delta = \Delta' \setminus a$. By induction hypothesis we have that there is Γ_2 such that $\Gamma_2; \Delta' \vdash_\ell P$ such that $\Gamma_2 \subseteq \Gamma_1$ and there is no Γ_3 such that $\Gamma_3 \subset \Gamma_2$ and $\Gamma_3; \Delta' \vdash_\ell P$. We then have that $\Gamma_2 \setminus a; \Delta \vdash_\ell (\boldsymbol{\nu} a)P$.

(*Case* (*Piece*))

Follows directly from induction hypothesis.

(*Case* (*Output*))

We have that $\Gamma; \Delta \vdash_\ell l^d!(n).P$ which is derived from $(\ell(d).l.(x)\Gamma' \bot \Gamma); \Delta \vdash_\ell P$ and $\Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma)$. By induction hypothesis we have there is Γ_1 such that $\Gamma_1; \Delta \vdash_\ell P$ and $\Gamma_1 \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma)$ and there is no Γ_2 such that $\Gamma_2 \subset \Gamma_1$ and $\Gamma_2; \Delta \vdash_\ell P$. Considering $\Gamma_1 \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma)$ and $\Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma)$ and Proposition 6.3.5 we have that $\Gamma_1 \cup \Gamma'\{x/n\}; \Delta \vdash_\ell P$. Let us now take $\Gamma_2 = \Gamma_1 \cup \Gamma'\{x/n\} \cup \{(\ell(d).l.(x)\Gamma' \prec e') \mid (\ell(d).l.(x)\Gamma' \prec_\Gamma e')\}$, for which we have $\Gamma_2 \subseteq \Gamma$ and $\Gamma_2; \Delta \vdash_\ell l^d!(n).P$. which completes the proof.

(*Case* (*Input*))

Follows lines similar to the case for rule (*Output*).

(*Case* (*This*))

Follows expected lines. ■

Notation A.4.30 *We denote by* $min(\Gamma)$ *the set of minimal events in ordering* Γ.

Definition A.4.31 (Safe Unfolding)
The safe unfolding of process P *with respect to* Γ *and* Δ*, noted* $su_{(\Gamma,\Delta)}(P)$*, is defined as follows:*

$$\begin{array}{lcll}
su_{(\Gamma,\Delta)}(\mathbf{0}) & \triangleq & \mathbf{0} & \\
su_{(\Gamma,\Delta)}(P \mid Q) & \triangleq & su_{(\Gamma,\Delta)}(P) \mid su_{(\Gamma,\Delta)}(Q) & \\
su_{(\Gamma,\Delta)}((\boldsymbol{\nu} a)P) & \triangleq & (\boldsymbol{\nu} a)(su_{(\Gamma,\Delta)}(P)) & \\
su_{(\Gamma,\Delta)}(\mathsf{rec}\ \mathcal{X}.P) & \triangleq & P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} & \\
su_{(\Gamma,\Delta)}(\mathsf{rec}_t\ \mathcal{X}.P) & \triangleq & P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} & (\textit{if } min(\Gamma) \subseteq min(\Gamma \cup \Delta(\mathcal{X}))) \\
su_{(\Gamma,\Delta)}(\mathsf{rec}_t\ \mathcal{X}.P) & \triangleq & \mathsf{rec}_t\ \mathcal{X}.P & (\textit{if } min(\Gamma) \not\subseteq min(\Gamma \cup \Delta(\mathcal{X}))) \\
su_{(\Gamma,\Delta)}(\mathcal{X}) & \triangleq & \mathcal{X} & \\
su_{(\Gamma,\Delta)}(n \blacktriangleleft [P]) & \triangleq & n \blacktriangleleft [su_{(\Gamma,\Delta)}(P)] & \\
su_{(\Gamma,\Delta)}(l^d!(n).P) & \triangleq & l^d!(n).P & \\
su_{(\Gamma,\Delta)}(\Sigma_{i\in I} l_i^d?(x_i).P_i) & \triangleq & \Sigma_{i\in I} l_i^d?(x_i).P_i & \\
su_{(\Gamma,\Delta)}(\mathsf{this}(x).P) & \triangleq & \mathsf{this}(x).P &
\end{array}$$

Lemma A.4.32
Let P, Q *be such that* $\Gamma; \Delta \vdash_\ell P$ *and* $Q = su_{(\Gamma,\Delta)}(P)$*. Then there is* Γ', Δ' *such that* $\Gamma'; \Delta, \Delta' \vdash_\ell Q$ *and* $\Gamma' \cup \bigcup(\Delta, \Delta') \subseteq \Gamma \cup \bigcup \Delta$.

Proof. By induction on the length of the derivation of $\Gamma; \Delta \vdash_\ell P$. Cases (*This*), (*Input*), (*Output*), (*RecVar*) and (*Stop*) are direct, and cases (*Par*), (*Res*) and (*Piece*) follow directly from induction hypothesis. We show the cases for (*Rec*) and (*RecTerm*).

(Case (*Rec*))

We have that $P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} = su_{(\Gamma,\Delta)}(P)$ and:

$$\Gamma; \Delta \vdash_\ell \mathsf{rec}\ \mathcal{X}.P \qquad \text{(A.4.32.1)}$$

which is derived from:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P \qquad \text{(A.4.32.2)}$$

From (A.4.32.2) and Lemma A.4.5 we have that:

$$\Gamma; (\Delta, \mathcal{X} \to \Gamma) \vdash_\ell P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} \qquad \text{(A.4.32.3)}$$

which completes the proof for this case.

(Case (*RecTerm*))

We have that:

$$\Gamma; \Delta \vdash_\ell \mathsf{rec}_t\ \mathcal{X}.P \qquad \text{(A.4.32.4)}$$

which is derived from:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P \qquad \text{(A.4.32.5)}$$

We consider the two distinct cases: either $min(\Gamma) \subseteq min(\Gamma \cup \Delta(\mathcal{X}))$ or $min(\Gamma) \not\subseteq min(\Gamma \cup \Delta(\mathcal{X}))$. In the latter case we have that $\mathsf{rec}_t\ \mathcal{X}.P = su_{(\Gamma,\Delta)}(\mathsf{rec}_t\ \mathcal{X}.P)$ and the proof is complete. In the former case we have that $P\{\mathcal{X}/\mathsf{rec}_t\ \mathcal{X}.P\} = su_{(\Gamma,\Delta)}(\mathsf{rec}_t\ \mathcal{X}.P)$. From (A.4.32.5) and

Lemma A.4.5 we conclude:

$$\Delta(\mathcal{X}); \Delta \vdash_\ell P\{\mathcal{X}/\mathsf{rec}_t\, \mathcal{X}.P\} \tag{A.4.32.6}$$

which completes the proof for this case. ∎

Notation A.4.33 *We denote by* Π *a non-empty sequence of communication prefixes* $\alpha_1.\ldots.\alpha_k$.

Lemma A.4.34
Let P *be a process such that* $\Gamma; \Delta \vdash_\ell P$ *and if* $e \in dom(\Gamma)$ *we have that* P *is not of the form* $\mathcal{C}[\Pi.\mathsf{rec}_t\, \mathcal{X}.R]$ *where there is a prefix relative to* e *in* R *and none in* Π*. If* $Q = su_{(\Gamma,\Delta)}(P)$ *and* $\Gamma'; \Delta' \vdash_\ell Q$ *then for all* $e \in \Gamma'$ *we have that* Q *is not of the form* $\mathcal{C}[\Pi.\mathsf{rec}_t\, \mathcal{X}.R]$ *where there is a prefix relative to* e *in* R *and none in* Π.

Proof. By contradiction. Let us assume there is $e \in dom(\Gamma')$ such that Q is of the form $\mathcal{C}[\Pi.\mathsf{rec}_t\, \mathcal{X}.R]$ where there is a prefix relative to e in R and none in Π. We consider two distinct cases: either $e \in dom(\Gamma)$ or $e \notin dom(\Gamma)$.

If $e \in dom(\Gamma)$ then we have that P is not of the form $\mathcal{C}'[\Pi'.\mathsf{rec}_t\, \mathcal{X}.R']$ where there is a prefix relative to e in R' and none in Π'. We conclude that P is of the form $\mathcal{C}[P']$ where $\Pi.\mathsf{rec}_t\, \mathcal{X}.R = su_{(\Gamma,\Delta)}(P)$ from which we have our intended contradiction as there is a prefix in Π relative to e.

If $e \notin dom(\Gamma)$ either P is of the form $\mathcal{C}[\Pi.\mathsf{rec}_t\, \mathcal{X}.R]$ where there is a prefix relative to e in R and none in Π, or not. In the latter case case the proof follows the same lines as the case when $e \in dom(\Gamma)$. In the former case we have that that either the initial action of Π is relative to an event e' which is lesser than e, in which case Π holds a prefix relative to e and we have our intended contradiction, or the initial action of Π is relative to an event e' greater than e: we have that $e \prec_{\Gamma'} e'$ and, by definition of safe unfolding, that $min(\Gamma) \subseteq min(\Gamma')$; if e' is in the set of minimal events of Γ then we have our contradiction as e' is not is the set of minimal events of Γ'; if e' is not in the set of minimal events of Γ then there is e'' minimal in Γ such that $e'' \prec_\Gamma e'$ and since e is not in the domain of Γ we have $e \prec_{\Gamma'} e''$ which contradicts $min(\Gamma) \subseteq min(\Gamma')$ and completes the proof. ∎

Definition A.4.35 (Event Ordering Unfolding Bound)
We say a process P *is unfolding bound by* Γ *if* P *is the result of a safe unfolding and for all* $e \in dom(\Gamma)$ *we have that* P *is not of the form* $\mathcal{C}[\Pi.\mathsf{rec}_t\, \mathcal{X}.R]$ *where there is a prefix relative to* e *in* R *and none in* Π.

Lemma A.4.36
Let P *be a process such that* $P :: T$ *and there is* T' *such that* $closed(T \bowtie T')$ *and* $\Gamma \vdash_\ell P$ *and* M_1^w *and* $M_2^{w'}$ *initial messages of* T*. If* $M_1^w \prec_\Gamma{}^\ell M_2^{w'}$ *and* $M_1 = ?\, l_1^d(C_1)$ *and* $l_1 \in \mathcal{L}_\star$ *and* $M_2 = \tau\, l_2^d(C_2)$ *then there is* $\tau\, l_1^d(C_1)$ *initial of* T.

Proof. By induction on the length of the derivation of $P :: T$ following expected lines. Notice that shared inputs expose a shared message interface (outputs on shared labels) and hence any greater τ initial message type is introduced by an output that matches the shared input, which type is initial and minimal since the shared input is initial and minimal. ∎

Notation A.4.37 *We say process P is final if it has no active* **this** *prefixes, hence if there is no $\mathcal{C}$ and Q such that $P = \mathcal{C}[\textbf{this}(x).P]$.*

Lemma A.4.38

Let P be a final process such that $P :: T$ and $\Gamma \vdash_\ell P$ and P is unfolding bound by Γ and T and Γ are precedence consistent. We have that either there is $P \to P'$ or for all M^w minimal initial messages of T there is $P \xrightarrow{M^w} P'$.

Proof. By induction on the length of the derivation of $P :: T$. We show the cases when the last rule applied is (*Par*), (*Res*) and (*Output*).

(*Case* (*Par*))

We have that:

$$P_1 \mid P_2 :: T_1 \bowtie T_2 \;\; (i) \qquad \text{and} \qquad \Gamma \vdash_\ell P_1 \mid P_2 \;\; (ii) \tag{A.4.38.1}$$

and that M^w is a minimal initial message of $T_1 \bowtie T_2$. (A.4.38.1)(i) is derived from:

$$P_1 :: T_1 \;\; (i) \qquad \text{and} \qquad P_2 :: T_2 \;\; (ii) \tag{A.4.38.2}$$

(A.4.38.1)(ii) is derived from:

$$\Gamma \vdash_\ell P_1 \;\; (i) \qquad \text{and} \qquad \Gamma \vdash_\ell P_2 \;\; (ii) \tag{A.4.38.3}$$

Let us take M^w such that M^w is a minimal initial message of $T_1 \bowtie T_2$, from which we have that M^w is a initial message type of $T_1 \bowtie T_2$. Considering Lemma A.4.20 we conclude that either: (1) M^w is an initial type of T_1; (2) M^w is an initial type of T_2; or (3) $M = \tau\, l^d(C)$ and $p\, l^d(C)^w$ is an initial message type of T_k, for $p \in \{\texttt{!}, \texttt{?}\}$ and $k \in \{1, 2\}$.

(Case (1)) We have that M^w is an initial type of T_1 and we prove that M^w is a minimal initial message of T_1 by contradiction. Let us thus assume that M^w is not a minimal initial message of T_1. We have that there is $M'^{w'}$ initial message type of T_1 such that $M'^{w'} \prec^\ell_\Gamma M^w$. We also have that $M'^{w'}$ is not an initial message of T, otherwise M^w would not be a minimal initial message of T, from which we have that either: (a) there is $M''^{w''}$ initial message of T such that $T \blacktriangleright M''^{w''} \rightsquigarrow M'^{w'}$ or (b) $M'^{w'}$ is guarded by a $\texttt{rec}$ in T.

(Case (a)) Since T and Γ are precedence consistent we conclude that $M''^{w''} \prec^\ell_\Gamma M'^{w'}$ and hence $M''^{w''} \prec^\ell_\Gamma M'^{w'} \prec^\ell_\Gamma M^w$. Since we have that $M''^{w''}$ is an initial message of T and is less than M^w, we conclude M^w is not a minimal initial message which gives us our intended contradiction, and hence M^w is a minimal initial message of T_1.

(Case (b)) From the fact that $M'^{w'}$ is guarded by a $\texttt{rec}$ in T we conclude that $M'^{w'}$ is guarded by a $\texttt{rec}$ in T_1, and we also have that $M'^{w'}$ is initial to T_1. We may show that $M'^{w'}$ is either guarded by a **rec** or by a $\textbf{rec}_t$ in P_1 by inspecting the typing derivation and the merge relation. If $M'^{w'}$ is guarded by a **rec** process which is at the top level of P_1 then we have a contradiction since P_1 is safely unfolded. If $M'^{w'}$ is guarded by a **rec** process which is not at the top level of P_1 then we have there is an initial action of T which is lesser than $M'^{w'}$ and the proof proceeds as in case (a). If $M'^{w'}$ is guarded by a $\textbf{rec}_t$ process then we have a contradiction as If $M'^{w'}$ is not in the domain of Γ, and hence M^w is a minimal initial message of T_1.

We thus have that M^w is a minimal initial message of T_1. By induction hypothesis on

(A.4.38.2)(i) and (A.4.38.3)(i) we have that there is P_1' such that either:

$$P_1 \to P_1' \qquad \text{or} \qquad P_1 \xrightarrow{M^w} P_1' \tag{A.4.38.4}$$

We then directly have that:

$$P_1 \mid P_2 \to P_1' \mid P_2 \qquad \text{or} \qquad P_1 \mid P_2 \xrightarrow{M^w} P_1' \mid P_2 \tag{A.4.38.5}$$

respectively, which completes the proof for this case.

(Case (2)) Analogous to (1).

(Case (3)) The proof that $?\, l^d(C)^w$ is a minimal initial message of T_i and $!\, l^d(C)^w$ is a minimal initial message of T_j, for $\{i, j\} \in \{1, 2\}$ follows lines similar to (1). Let us consider $i = 1$ and $j = 2$, hence, $?\, l^d(C)^w$ is a minimal initial message of T_1 and $!\, l^d(C)^w$ is a minimal initial message of T_2. By induction hypothesis on (A.4.38.2)(i) and (A.4.38.3)(i) and $?\, l^d(C)^w$ is a minimal initial message of T_1 we conclude that either:

$$P_1 \to P_1' \;\; (i) \qquad \text{or} \qquad P_1 \xrightarrow{?\, l^d(C)^w} P_1' \;\; (ii) \tag{A.4.38.6}$$

Likewise by induction hypothesis on (A.4.38.2)(ii) and (A.4.38.3)(ii) and $!\, l^d(C)^w$ is a minimal initial message of T_2 we conclude that either:

$$P_2 \to P_2' \;\; (i) \qquad \text{or} \qquad P_2 \xrightarrow{!\, l^d(C)^w} P_2' \;\; (ii) \tag{A.4.38.7}$$

If either (A.4.38.6)(i) or (A.4.38.7)(i) we have that:

$$P_1 \mid P_2 \to P' \tag{A.4.38.8}$$

and the proof is complete. If (A.4.38.6)(ii) and (A.4.38.7)(ii) we derive:

$$P_1 \mid P_2 \to P' \tag{A.4.38.9}$$

and, given that $M^w = \boldsymbol{\tau}\, l^d(C)^w$, we conclude:

$$P_1 \mid P_2 \xrightarrow{M^w} P' \tag{A.4.38.10}$$

which completes the proof for this case.

(*Case* (*Res*))

We have that:

$$(\boldsymbol{\nu} a)P :: T \;\; (i) \qquad \text{and} \qquad \Gamma \vdash_\ell (\boldsymbol{\nu} a)P \;\; (ii) \tag{A.4.38.11}$$

and that M^w is a minimal initial message of T. (A.4.38.11)(i) is derived from:

$$P :: T \mid a : [B] \tag{A.4.38.12}$$

and *closed*(B). (A.4.38.11)(ii) is derived from:

$$\Gamma' \vdash_\ell P \tag{A.4.38.13}$$

and $\Gamma = \Gamma' \setminus a$. Since M^w is a minimal initial message of T we have that either (*1*) M^w is a minimal initial message of $T \mid a : [B]$ (w.r.t. Γ', ℓ) or (*2*) there is M'^a minimal initial message of $T \mid a : [B]$ (w.r.t. Γ', ℓ). If (*1*) then the result follows from induction hypothesis. If (*2*) we have, since *closed*(B), that $M'^a = \tau\, l^{\downarrow}(C)^a$. By induction hypothesis on (A.4.38.12) and (A.4.38.13) and (*2*) we conclude:

$$P \to P' \qquad \text{or} \qquad P \xrightarrow{M'^a} P' \tag{A.4.38.14}$$

Since $M'^a = \tau\, l^{\downarrow}(C)^a$ in either case we have:

$$P \to P' \tag{A.4.38.15}$$

and hence:

$$(\boldsymbol{\nu} a)P \to (\boldsymbol{\nu} a)P' \tag{A.4.38.16}$$

which completes the proof for this case.

mininitialtype

(*Case* (*Output*))

We have that:

$$l^d!(n).P :: (L \bowtie n : C) \mid \oplus_{i \in I}\{M_i.B_i\} \;\; (i) \qquad \text{and} \qquad \Gamma \vdash_\ell l^d!(n).P \;\; (ii) \tag{A.4.38.17}$$

and there is $j \in I$ such that $M_j.B_j = \,!\, l^d(C).B$ and that M^w is a minimal initial message of $(L \bowtie n : C) \mid \oplus_{i \in I}\{M_i.B_i\}$. (A.4.38.17)($i$) is derived from:

$$P :: L \mid B \tag{A.4.38.18}$$

(A.4.38.17)(ii) is derived from:

$$(\ell(d).l.(x)\Gamma' \bot \Gamma) \vdash_\ell P \qquad \text{and} \qquad \Gamma'\{x/n\} \subseteq (\ell(d).l.(x)\Gamma' \bot \Gamma) \tag{A.4.38.19}$$

From (A.4.38.18) and (A.4.38.19) we conclude $M^w = \,!\, l^d(C)^d$ and we have that:

$$l^d!(n).P \xrightarrow{!\, l^d(C)^d} P \tag{A.4.38.20}$$

which completes the proof for this case. ∎

Lemma A.4.39

Let P be final process such that $P :: T$, where closed(T), and $\Gamma \vdash_{(z',z)} P$. If P is not a finished process then there are $\mathcal{C}, Q, T', l, d, C$ such that $P = \mathcal{C}[Q]$ and $Q :: T'$ and $\tau\, l^d(C)^w$ is minimal to T'.

Proof. Follows by induction on the length of the derivation of $P :: T$ in expected lines. Since *closed*(T) all message types in T are either of polarity τ or are of polarity ? and defined on a shared label. If all initial message types are of the second type then the process is finished, otherwise if there is an initial τ message type which is not minimal then Lemma A.4.36 gives us there is a minimal initial message type. ∎

Lemma A.4.40
Let P be a process such that $P :: T$, where closed(T) and $\Gamma; \Delta \vdash_\ell P$. We have that T and Γ are domain consistent and precedence consistent.

Proof. Follows directly from Lemma A.4.24 and Lemma A.4.28. ∎

Remark A.4.41 *Lemma A.4.34 characterizes the fact that if an event ordering talks only about the events of the current recursion iteration, then also the event ordering for the safe unfolding of the process talks only about the events of the new current recursion iteration. Furthermore, this property is direct for processes which do not contain* $\mathbf{rec}_t$ *processes. We assume, without any loss of generality, that reductions do not involve unfoldings if processes are always safely unfolded by their minimal event ordering — in the light of Lemma A.4.29. In such a way we are able to focus the analysis on the current recursion iteration and abstract away from further recursion iterations. We focus on processes that are unfolding bound by their minimal ordering.*

Proof of Theorem 6.3.10 (Lock Freeness)

Let P be a well-formed process such that $P :: T$, where closed(T), and $\Gamma; \Delta \vdash_\ell P$, where $\ell = (\mathsf{up}, \mathsf{here})$, then either P is a finished process or there is $P \to Q$.
Proof. Follows directly from Lemma A.4.40, Lemma A.4.38 and Lemma A.4.39. If P is has an active **this** prefix we directly have that $P \to P'$. Otherwise P is final and Lemma A.4.39 gives us that there is a minimal initial τ message type and hence from Lemma A.4.40 and Lemma A.4.38, considering Remark A.4.41, we conclude $P \to P'$. ∎

A.5 Chapter 7

Theorem 7.4.3 (Congruence)
(repetition of the statement in page 162)
Strong bisimilarity is a congruence.

1. *If $P \sim Q$ then $l^d!(\tilde{n}).P \sim l^d!(\tilde{n}).Q$.*
2. *If $P\{\tilde{x}/\tilde{n}\} \sim Q\{\tilde{x}/\tilde{n}\}$ for all $\tilde{n}$ then $l^d?(\tilde{x}).P \sim l^d?(\tilde{x}).Q$.*
3. *If $P\{x/n\} \sim Q\{x/n\}$ for all n then* **this**$(x).P \sim$ **this**$(x).Q$.
4. *If $\alpha_i.P_i \sim \alpha'_i.Q_i$, for all $i \in I$, then $\Sigma_{i\in I}\, \alpha_i.P_i \sim \Sigma_{i\in I}\, \alpha'_i.Q_i$.*
5. *If $P \sim Q$ then $n \blacktriangleleft [P] \sim n \blacktriangleleft [Q]$.*
6. *If $P \sim Q$ then $(\boldsymbol{\nu} a)P \sim (\boldsymbol{\nu} a)Q$.*
7. *If $P \sim Q$ then $P \mid R \sim Q \mid R$.*
8. *If $P\{\mathcal{X}/R\} \sim Q\{\mathcal{X}/R\}$, for all R, then* **rec** $\mathcal{X}.P \sim$ **rec** $\mathcal{X}.Q$.
9. *If $P \sim Q$ then* **throw**$.P \sim$ **throw**$.Q$.
10. *If $P \sim Q$ then* **try** P **catch** $R \sim$ **try** Q **catch** R.

11. If $P \sim Q$ then $\textsf{\textbf{try}}\ R\ \textsf{\textbf{catch}}\ P \sim \textsf{\textbf{try}}\ R\ \textsf{\textbf{catch}}\ Q$.

Proof. By coinduction on the definition of strong bisimulation. We show the proof for the axioms involving the new constructs, that exploits the other axioms already listed in Theorem 4.4.6, namely Theorem 4.4.6(7):

$$\text{If } P \sim Q \text{ then } P \mid R \sim Q \mid R. \tag{A.5.1.1}$$

(*If $P \sim Q$ then* $\textsf{\textbf{throw}}.P \sim \textsf{\textbf{throw}}.Q$)

Follows by a standard coinductive argument. Notice `throw` is the only observation that can be performed by $\textsf{\textbf{throw}}.P$ and by $\textsf{\textbf{throw}}.Q$, leading to bisimilar processes P and Q.

(*If $P \sim Q$ then* $\textsf{\textbf{try}}\ P\ \textsf{\textbf{catch}}\ R \sim \textsf{\textbf{try}}\ Q\ \textsf{\textbf{catch}}\ R$)

The proof proceeds by witnessing relation $\mathcal{R}$ defined as:

$$\mathcal{R} \triangleq \{(\textsf{\textbf{try}}\ P\ \textsf{\textbf{catch}}\ R, \textsf{\textbf{try}}\ Q\ \textsf{\textbf{catch}}\ R) \mid P \sim Q\} \cup \sim \tag{A.5.1.2}$$

is contained in $\sim$ by coinduction on the definition of strong bisimulation. We must show for every transition of $\textsf{\textbf{try}}\ P\ \textsf{\textbf{catch}}\ R$ there is a matching transition of $\textsf{\textbf{try}}\ Q\ \textsf{\textbf{catch}}\ R$ leading to states related in $\mathcal{R}$, and conversely.

Let us consider P_1, λ_1 such that:

$$\textsf{\textbf{try}}\ P\ \textsf{\textbf{catch}}\ R \xrightarrow{\lambda_1} P_1 \tag{A.5.1.3}$$

We have there are λ_2, P_2 such that (A.5.1.3) is derived from:

$$P \xrightarrow{\lambda_2} P_2 \tag{A.5.1.4}$$

We consider the two distinct cases: either λ_2 is `throw` or λ_2 is different from `throw`.

($\lambda_2 = \texttt{throw}$) We have that $\lambda_1 = \tau$ and:

$$P_1 = P_2 \mid R \tag{A.5.1.5}$$

Since $P \sim Q$ (A.5.1.2) we have that there is Q' such that:

$$Q \xrightarrow{\lambda_2} Q' \tag{A.5.1.6}$$

and $P_2 \sim Q'$. From (A.5.1.6) we derive:

$$\textsf{\textbf{try}}\ Q\ \textsf{\textbf{catch}}\ R \xrightarrow{\tau} Q' \mid R \tag{A.5.1.7}$$

Considering (A.5.1.1) we have:

$$P_2 \mid R \sim Q' \mid R \tag{A.5.1.8}$$

which completes the proof for this case, given $(P_2 \mid R, Q' \mid R) \in \mathcal{R}$.

($\lambda_2 \neq \texttt{throw}$) We have that $\lambda_1 = \lambda_2$ and:

$$P_1 = \textsf{\textbf{try}}\ P_2\ \textsf{\textbf{catch}}\ R \tag{A.5.1.9}$$

Since $P \sim Q$ (A.5.1.2) we have that there is Q' such that:

$$Q \xrightarrow{\lambda_2} Q' \tag{A.5.1.10}$$

and $P_2 \sim Q'$. From (A.5.1.10) we derive:

$$\textsf{\textbf{try}}\ Q\ \textsf{\textbf{catch}}\ R \xrightarrow{\lambda_1} \textsf{\textbf{try}}\ Q'\ \textsf{\textbf{catch}}\ R \tag{A.5.1.11}$$

Since $P_2 \sim Q'$ we have:

$$(\textsf{\textbf{try}}\ P_2\ \textsf{\textbf{catch}}\ R, \textsf{\textbf{try}}\ Q'\ \textsf{\textbf{catch}}\ R) \in \mathcal{R}$$

which completes the proof for this case.

(*If* $P \sim Q$ *then* **try** R **catch** $P \sim$ **try** R **catch** Q)

Follows expected lines. ∎