Imperial College of Science, Technology and Medicine
Department of Computing

A Study of Bisimulation Theory for Session Types

Dimitrios Kouzapas

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College, April 2013

Abstract

Bisimulation theory is a co-inductive tool used as a tractable method for studying equivalence relations in process calculi. This dissertation studies bisimulation theory for session types. We define the Asynchronous Session π-calculus (ASP for short), which is a session type calculus with queue configurations acting as a communication medium at each session endpoint The semantics for ASP offer fine-grained communication that enjoys the non-blocking property of asynchrony and the order-preserving property of session types. The ASP typing system is shown to be sound to guarantee type safety in the presence of subtyping. A typed labelled transition system gives rise to a bisimilarity which is sound and complete with respect to typed reduction-closed congruence. The bisimilarity theory of ASP highlights the determinacy and confluence properties of session types.

Event-driven programming is one of the major paradigms that utilise the asynchronous nature of distributed systems, where events are recognised as the presence of messages and their typed information in the communication medium. To justify the design choices made, we develop a superset of ASP, called the Eventful Session π-calculus (ESP for short), equipped with the minimal session primitives for an expressive event-driven computational model. The eventful session type system introduces the session set type, which is a collection of session types used to type a set of possible events. The ESP typing system maintains its consistency with respect to the ASP session typing system up-to a subtyping relation for session set types. The straightforward extension from ASP to ESP offers behavioural transparency, making the bisimilarity theory for the ASP a special case for the ESP theory – the bisimilarity relation coincides with typed reduction-closed congruence and determinacy and confluence properties are shown to hold for session transitions.

Many studies regarding event-driven computation have identified the selector or its equivalent, the polling operator, as the key construct for describing an event-driven framework. The selector is defined as a higher level construct in ESP and it is used to implement the core event handling routine called the event loop. Following the empirical study by Lauer and Needham, we define a session-based transformation from a multi-threaded server to an event loop server. Confluence theory proves that the transformation is type- and semantics-preserving.

In the last part of the dissertation we extend the behavioural theory to multiparty session types, both in the synchronous and the asynchronous cases. For each case, we examine two different typed labelled transition systems. In the first case we examine a standard labelled transition system with respect to the local session typing of processes. In the second case a choreography specification governs the behaviour of a multiparty session process and its observer. Each labelled transition system defines a bisimilarity relation, which coincides with the corresponding reduction-closed congruence.

To the memory of Kohei.

Acknowledgments

The first person I would like to acknowledge is my supervisor Prof. Nobuko Yoshida. During my years as a PhD student, I always had the strong feeling of her guidance and support. As a token of my gratitude I would like to expose her virtues of patience, kindness and hard work.

This thesis is dedicated to the memory of Dr. Kohei Honda, who passed away a few weeks before the thesis was completed. Kohei, who is regarded as the father of session types, was not just a co-author and a friend; he was an on going source of motivation, excitement and inspiration for me. With this acknowledgement also comes a promise that we will continue his work.

Furthermore, I would like to thank my co-supervisor Dr. Iain Phillips for the time he spent and the advices he gave me, especially in the last months of my PhD studies.

I also appreciate the valuable contribution of my co-author and friend Dr. Raymond Hu.

I am gratefull to the Department of Computing of Imperial College London for giving me the opportunity to study at Imperial as a doctoral student and for granting me the Doctoral Training Award, which supported my study fees.

Last but not least, I am forever in debt to my family. Whithout their support, the completion of this thesis would never be possible.

Declaration

This thesis was composed by myself and the material used is my own unless otherwise referenced.

Contents

List of Figures

Chapter 1

Introduction

This dissertation is concerned with the study of bisimulation equivalences in the context of Session type theory. A session type system defines a class of π-calculus terms that have a well-defined communication behaviour. Bisimulation equivalences in session types present interest as an object of study, due to the fundamental notions of communication imposed by session types and the programming design principles that derive out of these foundations.

1.1 Introductory Notions

Distributed systems were evolved from the need of coordinating into efficient use the many concurrent resources in a computing system. Many individuals see a distributed system as a set of computations with communication as the meta-function that coordinates the entire system. On another perspective, distribution should be understood as a unit of computation, in the sense that there is only one computation taking place in the entire system and that communication is part of the computation. A step towards this latter direction is to semantically define communication. Communication as an operation, affects more than one computation entity and can be considered as the connector holding a distributed system together. The

duality of interactions between distributed entities lies at the core of communication semantics. When duality is clarified via the send/receive of computing entities, called messages, the communication is characterised as message passing.

Semantics for message passing communication are identified into two major classes: synchronous message passing and asynchronous message passing. In synchronous message passing the send/receive operators have a temporal meet to achieve message exchange, i.e. both send and receive operations happen at the same time. Asynchronous communication, on the other hand, does not rely on the exact temporal send/receive interaction, introducing the notion of the communication medium as an intermediate stage for message passing. A sender is always free to interact with the medium to store a message, while the receiver consumes the message from the medium at a later time.

In practice, distributed systems use forms of communication, where the send and receive operators do not require synchronisation. Asynchrony in concurrent systems uses intermediate memory buffers for storing data. Message presence in a memory buffer can be checked ahead of its reception, adding flexibility to asynchronous communication programming. The event-driven paradigm is one of the major frameworks for utilising asynchrony in distributed systems. The basic notion of event-driven programming is the event. An event can be recognised as the presence of a message in the communication medium. The event-driven framework has the facilities to detect and react on the presence of an event i.e. the event-driven mechanism can recognise the type of a message upon its reception and proceed with processing.

The semantics for message passing communication were described as mathematical objects in the context of process calculi. One such calculus is the π-calculus that was originally proposed in [MPW92]. The fundamental building block of the π-calculus processes is called name. A name models a communication link and exists as a process building block, in either the send or the receive mode. Communication semantics derive from the send/receive interaction on a name. The messages passed on send/receive interactions are also names giving rise to the notion of link mobility. The π-calculus uses names and message passing com-

munication as the only primitives to describe a mathematical framework for concurrency. A fragment of the π-calculus, called the asynchronous π-calculus was proposed in [HT91b]. The asynchronous π-calculus abstracts asynchronous communication semantics, using a simple restriction on the sequencing of the message send prefix on processes. An important result is the encoding of the synchronous π-calculus in terms of the asynchronous π-calculus.

Type theory and type systems for models of computation were developed to statically enforce well-defined properties to the computation and to study the dynamics of programs through typing abstractions. Session types is a typing system for the π-calculus proposed by Honda et al. in [HVK98], developed to abstract the fundamentals of message passing communication as a type theory. The basic type for session types is the session or session channel. The typing system for sessions is based on three basic principles: i) the send/receive duality of the communication sequence in a session; ii) the linearity of session names[1]; and iii) the type match between the messages carried by a send/receive interaction.

Session types gained attention by the concurrency research community, over the last years, due to the *good* properties they enforce on a distributed program: i) well type session programs do not suffer from communication deadlocks; ii) communication is handled as a linear resource iii) type soundness is a cornerstone property of type theories.

Session types were originally limited to the interactions of a binary set of session channels and presented limitations when applied to more than two processes. The idea of communication choreography influenced session type theory, for the development of multiparty session types, [HYC08, B$^+$08]. Choreography in communication requires the knowledge of the communication scenario for all computing participants to be specified in advance. Multiparty session types' main notion, is the global multiparty session type, that describes a global communication interaction between a set of computational participants. The projection of a global type to a local session type for each participant, allows for the local type checking of each

[1] The term linear is used to characterised a resource as finite, i.e. it is used a finite number of times, or it is used by a finite number of computation points at any given time

participant implementation.

A central point of study for process calculi is the behavioural equivalence theory for processes. Behavioural theory attempts to answer questions on how processes interact with, or better yet are observed by, their environment and how processes are related with respect to their behaviour. A labelled transition system on processes abstracts process transitions as a labelled graph. The bisimulation relation (developed originally for CCS [Mil80] following the intuitions from [Par81]) is a relation between labelled transition graphs and was emerged from the need of a fine-tuned equivalence between processes. Bisimulation has the property to be defined in the co-inductive framework. The application of the co-inductive method over co-directed process sets has lead to the definition of a universal closure called bisimilarity. A co-inductive subset of bisimilarity over a pair of processes is called a bisimulation. The existence of a bisimulation between processes is evidence of their equivalence.

1.2 Aim and Motivation

The aim of this dissertation is to study bisimulation relations for the π-calculus in the context of session types. Bisimulation was broadly studied in the setting of the untyped π-calculi, but less work has been done for equivalence relations in typed process calculi. A session type system is an excellent typing setting for studying typed bisimulations, since it offers a well-defined and desirable set of properties for message passing communication. The first motivation of this work is to exhibit a core theory for the session typed bisimulation. We further take care to include in the bisimulation theory an applied aspect of session typed behaviour.

A first applied aspect focuses on the development of a session type theory for the asynchronous π-calculus, using intermediate buffers as processes. Network transport protocols such as TCP use intermediate memory buffers to provide reliable and ordered delivery of

meaningful formatted messages, once a connection is established. The distinction between a possibly unordered communication outside a connection and an order-preserving connection inside an established connection is a key point of interest when handling communication. A session type system exhibits a natural fit towards order-preserving communication, since it allows a structured sequence of communication.

A message presence in an intermediate buffer, allows for the receiver to asynchronously inspect and consume messages. This fact gives rise to an event-driven discipline for session types. Event-driven programming is characterised by a reactive flow of control, driven by the occurrences of computation events. Primary motivations for event-driven programming include performance and scalability, particularly for highly concurrent web applications. Unfortunately, the flexibility and performance of traditional event-driven programming comes at the cost of more complex programs with obfuscated flow of control.

We use session types to type an asynchronous version of the π-calculus that uses: i) asynchrony to establish session connections; and ii) intermediate buffers to achieve asynchronous and order-preserving delivery of messages inside connections. Note that we assume perfect intermediate buffers, that have no message losses and have an unbounded capacity. Static session typing is adjusted to the dynamic nature of event-driven programming with the introduction of a message arrival expression and a statically checked type matching process construct. Session types provide a static and communication-centred perspective for event-driven programming, making reactive event-driven programs easier to write and understand.

We define session typed asynchronous bisimulation and show that it is the maximal reduction-closed congruence relation that preserves observation [HY95]. The motivation for reasoning about event-driven systems has led to the study of the properties of confluence [Mil89, PW97] on session transitions, which is defined using the session typed bisimulation. We show that session transitions preserve the confluence property, due to the linear and structured usage of session channels.

The event-driven session type framework can be used to abstract as processes and study the properties of event-driven programming primitives and routines. In this work we abstract as typed processes and study the behaviour of the selector primitive [Lea03, NIO] and the basic event-driven routine, the event-loop. Based on [LN79], the event-loop is used to define a typed transformation from a thread-based server and an event-based server. We then use confluence theory on session transitions to reason that the transformation results in a behaviourally indistinguishable process.

This work is completed with a study of bisimulation theory in the multiparty session type [HYC08, B$^+$08] setting. We use the principles developed previously in this dissertation to define a behavioural theory for both the synchronous and the asynchronous multiparty session calculi. The fact that all distributed processes follow a global multiparty protocol motivated a novel contribution of this part, which is to control the behaviour of a system based on the global multiparty session type.

1.3 Contribution

We develop a core process calculus called the Asynchronous Session π-calculus – ASP for short. Asynchrony for session channels in the ASP is achieved with the use of first-in first-out input and output queues, called session configurations, for each session endpoint, that define a fine-grained communication with the non-blocking property of asynchrony and the order-preserving property of session types. Similarly we use a first-in first-out queue as a shared name endpoint configuration to model asynchrony in shared names [Kou09].

We extend the standard session type theory (cf. [YV07]) in the presence of subtyping (cf. [GH05]) to develop a session type system with novel typing rules to type session endpoint configurations (cf. [MY09]). The soundness and safety of the typing system are proved via standard subject reduction and progress theorems.

A behavioural theory over session type processes should take into account the session typing. We develop a labelled transition system over the session type environment, which we use together with the standard labelled transition system for processes, to define a transition relation over typed processes. The typed transition is used as the monotone function to co-inductively define a weak bisimilarity relation, which is the largest reduction-closed congruence that preserves observation. We follow the intuition that session channels are linear resources to prove that session transitions (i.e. actions on session channels) enjoy the properties of confluence and determinacy.

Asynchronous communication is clarified with the definition of a superset of the ASP called the Eventful Session π-calculus (ESP) with the operational facilities to model the event-driven paradigm. More specifically we define the `arrive` construct used to check shared and session configuration endpoints for the arrival of messages and the `typecase` construct, that uses type matching on session names to decide process continuation. To type the `typecase` construct, the type system is extended with the definition of session set types. A subtyping theory over session set types makes the ESP typing system transparent up-to subtyping with respect to the ASP typing system. The behavioural theory for ESP shows the validity for all major results studied in ASP.

We demonstrate the properties of the ESP following simple examples. Specifically the equivalences, up-to session channel permutation, show the non-blocking and order-preserving properties of ESP communication. We demonstrate the counter-example that proves the lack of confluence of an `arrive`-guarded process. This result is useful when reasoning about event-driven ESP systems. A final example on the equivalence between the permutations of sequential `arrive`-inspections inside a recursive loop, gives a first and strong intuition about the behavioural nature of event-driven programming. The bisimulation equivalences for the ESP are distinguished by classic synchronous and asynchronous bisimulation theory. We demonstrate this distinction through a comparison of standard equivalence relations in different calculi.

An empirical study on event-driven programming identifies the selector construct as the key construct used to program event-driven asynchronous systems. The selector, also known as the polling operator in the context of operating systems, checks a set of communication channels for the arrival of messages. Upon a message arrival, the selector returns the corresponding communication channel for processing. The selector construct is used to build a basic event-driven programming construct called the event-loop. The event-loop is a recursive block of code that selects, identifies and processes events. We use the `arrive` construct to encode a type-safe selector primitive as a high-level construct on ESP. We then use the selector together with the `typecase` construct to define a type-safe event-loop process. We show that the behaviour of an event-loop that enjoys the confluence property, is not distinguished by the order the event-loop selector checks session channels for message arrival.

Lauer and Needham in their early work [LN79] argued that a concurrent program can be written equivalently in a thread-based programming style, or an event-based style. Following their work we define an ESP transformation from a thread-based server to an event-loop based server. We use the confluence theory to prove that both servers are semantically equivalent.

In the last part of the thesis we develop a multiparty session type behavioural theory on the basis of the binary session type theory. The main objective is to develop a modular theory for multiparty session types and explore its behavioural relations and properties. We develop the semantics for Synchronous Multiparty Session π-calculus (or Synchronous MSP for short) to use it as a core definition for the development of a set of Asynchronous Multiparty π-calculi (Asynchronous MSP for short) (cf. [B$^+$08]).

The Asynchronous MSP calculi are called: i) the output asynchronous MSP; ii) the input asynchronous MSP; and iii) the input/output asynchronous MSP and are defined by extending the synchronous MSP semantics with the session configuration construct. The distinction between them is based on the way the session configuration semantics are defined, in order to emulate asynchrony. Briefly output asynchrony respects asynchrony between the same sender

and different receiver, input asynchrony respects asynchrony between the same receiver and different senders and input/output asynchrony is inspired by the ASP definition.

We define the behavioural semantics on MSP calculus, based on the behavioural theory for binary session types, where we use local session types to define a typed LTS. For each MSP calculi, we prove that the proposed bisimulation is the maximal reduction-closed congruence that preserves observation.

As a last contribution, we propose a novel definition for controlling the typed LTS with the use of the multiparty global type. We use the typed LTS to define the globally governed bisimilarity that coincides with the corresponding reduction-closed observation-preserving congruence.

1.4 Publications and Detailed Contribution

The following papers were published as a result of the research done for the requirements of this dissertation and are its primary contributing sources. The papers are presented in chronological order. For each paper the author's contribution is given. The relevance of each work with this dissertation is given through the dissertation Chapter correspondence.

1. Raymond Hu, **Dimitrios Kouzapas**, Olivier Pernet, Nobuko Yoshida and Kohei Honda. Type-Safe Eventful Sessions in Java. In *ECOOP*, volume 6183 of LNCS, pages 329-353, 2010.

   Author's Contribution: Contribution on the development of the Eventful Session π-calculus syntax and typing system. Proof of the main subject reduction and progress safety theorem. Define and prove the properties of the selector construct.

   Chapter Correspondence: Chapter 3, Chapter 4 and Chapter 5.

2. **Dimitrios Kouzapas**, Nobuko Yoshida and Kohei Honda. On Asynchronous Session Semantics. In *FMOODS/FORTE*, volume 6722 of LNCS, pages 228-243, 2011.

   Author's Contribution: Development of the Eventful Session π-calculus, ESP, syntax and typing system. Studied bisimulation and confluence theory for the ESP. Proof of selector construct's properties and main result for the Lauer-Needham transformation. More specifically the definition of asynchronous session initiation, the runtime typing system, the device of the selector construct and using confluence to analyse the Lauer-Needham transformation were the author's innovation.

   Chapter Correspondence: Chapter 3, Chapter 4 and Chapter 5.

3. **Dimitrios Kouzapas**, Nobuko Yoshida, Raymond Hu and Kohei Honda. On Asynchronous Eventful Session Semantics. To appear in special issue: Behavioural Types of the MSCS.

   Author's Contribution: Development of the Eventful Session π-calculus, ESP, syntax and typing system. Studied bisimulation and confluence theory for the ESP. Proof of selector construct's properties and main result for the Lauer-Needham transformation. More specifically the definition of asynchronous session initiation, the runtime typing system, the device of the selector construct and using confluence to analyse the Lauer-Needham transformation were the author's innovation.

   Chapter Correspondence: Chapter 3, Chapter 4 and Chapter 5.

4. **Dimitrios Kouzapas** and Nobuko Yoshida. Globally Governed Session Semantics. To appear in CONCUR, 2013.

   Author's Contribution: Semantics for the Synchronous Multiparty Session π-calculus (MSP). Development of the behavioural theory for locally controlled and globally governed bisimulation and proof of the coincidence of each bisimilarity relation with the corresponding reduction-closed, barb-preserving congruences. Work on different use-case scenarios for the Ocean Oservatory Initiative framework in order to apply the be-

havioural theory for the MSP.

Chapter Correspondence: Chapter 6

1.5 Chapter Outline

In Chapter 2 we present the basic background theory for session types and π-calculus bisimulations. We also present a literature review for the event-driven programming paradigm. The rest of the thesis is divided into three parts. The first part has three chapters for the study of bisimulations for binary session types. Chapter 3 defines the theory for the Asynchronous Session π-calculus - ASP. We define the calculus for asynchronous buffer communication and a session type system for this calculus. The chapter concludes with the definition of the asynchronous session bisimilarity that is the maximum reduction-closed congruence that preserves observation. Based on the bisimulation we define a confluence theory for reasoning with session type systems. Chapter 4 extends the ASP to define the Eventful Session π-calculus-ESP, which is used to describe the event-driven programming framework. Together with the calculus, we extend the session type system and we prove that the corresponding bisimilarity is the maximum reduction-closed congruence that preserves observation. We also show that the properties for the confluence theory continue to hold for the ESP. An extensive study of the applications of the ESP is done in Chapter 5. We present the basic examples that characterise the ESP and we do a comparison of the ESP with other well known π-calculi. In the second part of the chapter we encode the *selector* construct and prove its basic properties. Based on the *selector* construct we define a transform from a multi-threaded server to a single-thread event-based server following the Lauer-Needham transform [LN79]. We show that the transform is type and semantics preserving. The second part studies the bisimulation theory for multiparty session types. In Chapter 6 we define a multiparty session type theory for both the synchronous and the asynchronous cases. Based on the multiparty session type we define two classes of bisimulations: i) the bisimulation that is controlled by the local session type; and ii)

the bisimulation that is controlled by the global session type. Each bisimilarity is shown to be the corresponding maximal reduction-closed congruence that preserves observation. The last part of the thesis (Chapter 7) compares the results of this dissertation with related literature and concludes the thesis.

Chapter 2

Background

In this chapter we introduce the definitions and the background work around the main concepts that concern this dissertation. In § 2.1 we present a basic theory for session types. We proceed in § 2.2 with the introductory notions for the bisimulation theory in the π-calculus. The last section of this chapter (§ 2.3) is concerned with a survey on the event-driven frameworks that are used for the requirements of this dissertation.

2.1 Session Types

Session types were originally developed as a tractable typing system, to offer structured communication with respect to communication duality and sort/type matching of the communicating data. They ensure a number of *good* properties regarding message passing communication. Despite their simplicity and ease of understanding, session types have a deep impact in distribution theory and message passing. Furthermore, session types can be applied in numerous ways to describe communication behaviour throughout the levels of computational abstraction.

The basics for typing message passing send/receive interactions were set in a 1993 paper by Honda called *"Types in Dyadic Interactions"* [Hon93]. Session types in their binary form were first proposed by Honda et al. in [HVK98], where session types are described as a tractable typing system that offers structured communication. A milestone work on session types, [GH05], introduces the notion of session channel polarities and proves the properties of session types in the presence of session subtyping. A revised version for session types that followed in [YV07] shows type soundness for two session type systems via a subject reduction theorem, with subject reduction being used to prove type safety and progress. Session types for object oriented languages were first studied in [DCMYD06] and a study of the capabilities of session types in the context of higher-order mobile processes can be found in [MY07] and in [MY09], with the latter work introducing asynchronous session typed communication with the use of FIFO queues.

The concept of communication choreography influenced session type theory for the development of multiparty session types [HYC08, B+08]. Multiparty session types are based in the concept the *global protocol*, that describes a communication consensus between different computation participants. The global protocol is then projected to local protocols able to describe the session interaction inside a single session participant.

Parametrised multiparty session types were proposed in [YDBH10], where a new session type primitive was developed to allow the definition of network topologies with a parametrized number of nodes. Dynamic multiparty session types [DY11] allow for an arbitrary number of parties to join or leave a session interaction. A multiparty session type system related with finite state automata is presented in [DY12].

A Curry Howard-like correspondence for session types can be found in the works by Caires and Pfening [CP10] and Wadler [Wad12]. These works abstract session types as linear logic propositions and the evaluation of session typed programs as cut-eliminated theorem proofs.

A session type system defines types on π-calculus names. The basic such type is called

session. The session typing system ensures three main properties for a session:

1. The linearity of usage. A session name is a linear resource used as a communication link between at most two endpoints. The term linearity is used here to identify a session name as a finite resource.

2. The duality of usage. Two session endpoints that implement the same session should have a dual send/receive correspondence, i.e. the send/receive sequence of one endpoint is dual to the send/receive sequence of the other endpoint.

3. Type matching. On a communication interaction, the type of the object being sent should be the same with the type of the object being received.

The above three main characteristics offer solutions for fundamental problems in concurrent computing. A linear typing system ensures the sound access to scarce and/or limited resources. The duality of communication on session types excludes the possibility of communication deadlocks inside a session. Furthermore, if we combine duality with linearity we can avoid other communication related erroneous situations such as starvation. Type matching ensures the soundness of the message exchange and ensures the safety of a program.

2.1.1 Session Types Semantics

This section presents the semantics for binary session types based on the second session π-calculus presented in [YV07].

Syntax: We define the syntax for a session π-calculus:

$$
\begin{array}{lcl}
P & ::= & \overline{u}(s).P \mid u(x).P \mid k!\langle v\rangle;P \mid k?(x);P \mid k \oplus l;P \mid k\&\{l_i : P_i\}_{i\in I} \\
 & \mid & (\nu\ s)P \mid (\nu\ a)P \mid P_1 \mid P_2 \mid \mathbf{0} \mid X \mid \texttt{def}\ D\ \texttt{in}\ P
\end{array}
$$

$$
\begin{array}{lcllcl}
u & ::= & a \mid x & v & ::= & \texttt{tt} \mid \texttt{ff} \mid a \mid s \\
k & ::= & s \mid \overline{s} \mid x & D & ::= & X_1 = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_2 = P_2
\end{array}
$$

The calculus syntax assumes shared names to range over $a, b, \ldots$ and session names to range over $s, s', \ldots$. We assume that a session name s exists in endpoint pairs denoted as s and $\overline{s}$ and let $\overline{\overline{s}} = s$. Labels include $l, \ldots$, constants include boolean $\texttt{tt}, \texttt{ff}$ and variables range over $x, y, z, \ldots$. Values v are either constants, shared names (a) or session names (s). u denotes a shared name a or a variable for a shared name, while k denotes a session name s or a variable for a session name. We use the symbol n to denote either a shared name a or a session name s.

Terms $\overline{u}(s).P$ and $u(s).P$ define the request (resp. the accept) prefix of process P used for the initiation of the fresh session s on the shared name u. Term $s!\langle v\rangle;P$ defines the send of value v via session channel s and then continuing with process P. Respectively $s?(x);P$ receives a value substituted on variable x and continues with process P. Terms $s \oplus l;P$ and $s\&\{l_i : P_i\}_{i\in I}$ describe the select and branch prefixes. The select prefix selects label l on session channel s and continues with process P. The branch prefix offers a set of labels $\{l_i\}_{i\in I}$ for branching. The rest of the terms are standard π-calculus terms. Terms $(\nu\ s)P$ and $(\nu\ a)P$ restrict session name s (resp. shared name a) in the scope of process P. Term $P_1 \mid P_2$ is the parallel composition between process P_1 and P_2. The inactive term is denoted as $\mathbf{0}$. We define recursion with the term $\texttt{def}\ D\ \texttt{in}\ P$ where X is the process variable term and D is a set of process definitions for process variables, that has the form $X_1 = P_1\ \texttt{and}\ \ldots\ \texttt{and}\ X_n = P_n$.

We define $\texttt{fn}(P), \texttt{bn}(P)$ and $\texttt{n}(P)$ as the free names, bound names and names in P respectively. Furthermore, closed terms (i.e. processes with no free names) are called programs.

Structural Congruence: Structural congruence is the least congruence relation, over the rules:

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad (P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$$

$$P_1 \equiv_a P_2 \qquad (\nu\ n)\mathbf{0} \equiv \mathbf{0}$$

$$(\nu\ n)P_1 \mid P_2 \equiv (\nu\ n)(P_1 \mid P_2) \qquad \text{if } n \notin \mathtt{fn}(P_2)$$

$$(\nu\ n)\mathtt{def}\ D\ \mathtt{in}\ P \equiv \mathtt{def}\ D\ \mathtt{in}\ (\nu\ n)P \qquad \text{if } \notin \mathtt{fn}(D)$$

$$(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q \equiv \mathtt{def}\ D\ (P \mid Q) \qquad \text{if } \mathtt{fpv}(D) \cap \mathtt{fpv}(Q) = \emptyset$$

$$\mathtt{def}\ D_1\ \mathtt{in}\ (\mathtt{def}\ D_2\ \mathtt{in}\ P) \equiv \mathtt{def}\ D_1\ \mathtt{and}\ D_2\ \mathtt{in}\ P \qquad \text{if } \mathtt{fpv}(D_1) \cap \mathtt{fpv}(D_2) = \emptyset$$

The $\mathbf{0}$ process has no structural effect when composed in parallel with another process. Furthermore, structural congruence respects the commutativity and associativity properties of the parallel operator. Restricting a name in the inactive process has no effect and alpha-conversion is included in the structural congruence. Rule $(\nu\ n)P_1 \mid P_2 \equiv (\nu\ n)(P_1 \mid P_2)$ if $n \notin \mathtt{fn}(P_2)$ says that the restriction scope of a name n in P_1 can be extended to a parallel process P_2 if n is not free in P_2. The last three rules define the structural congruence for the recursive term. The restriction of name n on a recursive term $\mathtt{def}\ D\ \mathtt{in}\ P$ can be limited to restrict the process P if n is not free in the body D. A recursive term composed in parallel with another process $(\mathtt{def}\ D\ \mathtt{in}\ P) \mid Q$ is structurally equivalent with $\mathtt{def}\ D\ \mathtt{in}\ (P \mid Q)$ if D and Q do not share any free process variables. Finally, term with a nested recursion can be written as one level recursive term if the bodies of the recursion do not share any free process variables.

Operational Semantics: The operational semantics clarify the syntax and the intuitions for the session typed π-calculus.

$$\begin{array}{rcl}
\overline{a}(s).P_1 \mid a(x).P_2 & \longrightarrow & (\nu\ s)(P_1 \mid P_2\{\overline{s}/x\}) \\
s!\langle v\rangle;P_1 \mid \overline{s}?(x);P_2 & \longrightarrow & P_1 \mid P_2\{v/x\} \\
s \oplus l_k;P \mid \overline{s}\&\{l_i : P_i\}_{i\in I} & \longrightarrow & P \mid P_k \quad k \in I \\
\texttt{def}\ D\ \texttt{in}\ (P_1 \mid X) & \longrightarrow & \texttt{def}\ D\ \texttt{in}\ (P_1 \mid P_2) \quad X = P_2 \in D
\end{array}$$

$$\frac{P \longrightarrow P'}{(\nu\ s)P \longrightarrow (\nu\ s)P'} \qquad \frac{P \longrightarrow P'}{(\nu\ a)P \longrightarrow (\nu\ a)P'}$$

$$\frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \frac{P \equiv P_1 \quad P_1 \longrightarrow P_2 \quad P_2 \equiv P'}{P \longrightarrow P'}$$

A fresh session s is created on the send/receive interaction on a shared name. The request process $\overline{a}(s).P_1$ sends a fresh session s to the accept process $\overline{a}(s).P_2$, which receives s via variable substitution. The exchange of value on a session send/receive interaction follows the standard π-calculus definition, where the receiver uses the value sent by the sender via substitution. The select/branch interaction defines the selection of a continuation in the branch prefixed process from a select prefixed process. The select/branch interaction uses labels to select and to offer selections respectively. In recursion semantics a process variable is instantiated through a reduction. Name restriction and parallel composition on a process P do not affect P's internal reductions. Finally the reduction relation is closed under the structural congruence relation. We define $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$

Session Syntax: We first introduce the syntax for session types.

$$\begin{array}{rcl}
U & ::= & \texttt{bool} \mid S \mid \langle S\rangle \\
S & ::= & !\langle U\rangle;S \mid ?(U);S \mid \oplus\{l_i : S_i\}_{i\in I} \mid \&\{l_i : S_i\}_{i\in I} \mid \mu t.S \mid t
\end{array}$$

Shared type U denotes boolean types `bool`, session types S (described next) and shared names channels $\langle S \rangle$. Session types S are included in type U, so we can allow the delegation of session names with type S. Session types S include the terms $!\langle U \rangle;S$ and $?(U);S$ to define the output (resp. the input) of type U and then continue with S. Type $\oplus\{l_i : S_i\}_{i \in I}$ defines the selection from a session type set $\{S_i\}_{i \in I}$ on labels $\{l_i\}_{i \in I}$ respectively. Similarly type $\&\{l_i : S_i\}_{i \in I}$ describes the session types $\{S_i\}_{i \in I}$ offered for branching on labels $\{l_i\}_{i \in I}$ respectively. Type `end` is the inactive type. Recursion is defined using the primitive recursor $\mu t.S$ with t as the recursive variable.

Before we proceed with the definition of a type system we define the duality relation between sessions. Session duality is used by the typing system to ensure the duality of interactions between session endpoints.

$$\overline{\texttt{end}} = \texttt{end} \qquad \overline{t} = t \qquad \overline{\mu t.S} = \mu T.\overline{S} \qquad \overline{!\langle U \rangle;S} = ?(U);\overline{S} \quad \overline{?(U);S} = !\langle U \rangle;\overline{S}$$
$$\overline{\oplus\{l_i : S_i\}_{i \in I}} = \&\{l_i : \overline{S_i}\}_{i \in I} \qquad \overline{\&\{l_i : S_i\}_{i \in I}} = \oplus\{l_i : \overline{S_i}\}_{i \in I}$$

The duality relation relates opposing send/receive types. The dual of the output prefixed session type is the input prefixed dual session type. The dual of the input prefixed session type is symmetric. Similarly the dual of the select type is the branch type with the dual set of session types. The dual of the branch type is symmetric. Duality for `end` and t is the identity and finally duality of the recursion implies the duality of the session type inside the body of the recursion.

Typing System: The typing system defines judgements of the forms:

$$\Gamma \vdash v : U \qquad \text{and} \qquad \Gamma \vdash P \triangleright \Delta$$

with

$$\Gamma \ ::= \ \Gamma \cdot v : U \ \mid \ \Gamma \cdot X : \Delta \ \mid \ \emptyset \qquad\qquad \Delta \ ::= \ \Delta \cdot s : S \ \mid \ \emptyset$$

where Γ is a type environment that maps values v to shared types U and process variables X to session type environments Δ and Δ is a session type environment that maps session endpoints to session types.

Judgement $\Gamma \vdash v : U$ is read as value v has type U under type environment Γ. Judgement $\Gamma \vdash P \triangleright \Delta$ is read as process P has session typing Δ under environment Γ.

The session typing system is defined in Figure 2.1.

Boolean values true `tt` and false `ff` are always typed with the boolean `bool` type. An environment $\Gamma \cdot v : U$ judges value v with the U type.

Request and accept term check if the shared environment maps the shared channel to the output (resp. the input) shared channel type. Send and receive on a session name require that a value being sent (resp. received) on a session channel is typed according to the shared environment Γ. We call delegation the action of sending a session channel through another session channel. Delegation respects the linear properties of the carried session. When a session channel is being sent, the typing rule requires that the sent is present with the correct type in the linear environment Δ. Similarly when a session s is being received the typing rule requires that the type of s is present in the linear environment Δ after the reception. Selection is typed similarly to send using the select session type. The branching type on a session channel requires the typing of the set of the branching processes. Parallel composition concatenates two disjoint session environments. Not disjoint session environments will lead to the occurence of more than one endpoint with the same name and thus break session type linerity. The restriction of a session name checks the duality of its two session endpoints, while the restriction of a shared name has no effect in the typing judgement. The inactive process is typed with a complete session typing (i.e. all session names are mapped to the `end` session type). Finally recursion variables are typed according to their mapping in the shared environment Γ. The recursive term typing checks for the definition of a process variable to agree with the process variable typing.

$$\Gamma \vdash \texttt{tt}, \texttt{ff} : \texttt{bool} \qquad \Gamma \cdot v : U \vdash v : U$$

$$\frac{\Gamma \vdash u : \langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : S}{\Gamma \vdash u(x).P \triangleright \Delta} \qquad \frac{\Gamma \vdash u : \langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : S}{\Gamma \vdash \overline{u}(x).P \triangleright \Delta}$$

$$\frac{\Gamma \vdash v : U \quad \Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k!\langle v \rangle; P \triangleright \Delta \cdot k : !\langle U \rangle; S} \qquad \frac{\Gamma \cdot x : U \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(U); S}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k!\langle k' \rangle; P \triangleright \Delta \cdot k : !\langle S' \rangle; S \cdot k' : S'} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k : S \cdot k' : S'}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(S'); S}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k \oplus l; P \triangleright \Delta \cdot k : \oplus\{l : S\}} \qquad \frac{\forall i \in I, \Gamma \vdash P_i \triangleright \Delta \cdot k : S_i}{\Gamma \vdash k\&\{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot k : \&\{l_i : S_i\}}$$

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \quad \Gamma \vdash P_2 \triangleright \Delta_2 \quad \Delta_1 \cap \Delta_2 = \emptyset}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cdot \Delta_2} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k : S \cdot \overline{k} : \overline{S}}{\Gamma \vdash (\nu\, k) P \triangleright \Delta}$$

$$\frac{\Delta \text{ end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \frac{\Gamma \vdash P \triangleright \Delta}{\Gamma \vdash (\nu\, u) P \triangleright \Delta}$$

$$\Gamma \cdot X : \Delta \vdash X \triangleright \Delta \qquad \frac{\Gamma \cdot X : \Delta_1 \vdash P_1 \triangleright \Delta_1 \quad \Gamma \cdot X : \Delta_1 \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash \texttt{def}\ \ X = P_1\ \texttt{in}\ \ P_2 \triangleright \Delta_2}$$

Figure 2.1: Typing system for binary Session Types

The next definition captures the requirements for a well defined process. We require that dual endpoints (if present) in a linear environment Δ have dual session types. The requirement enforces the basic properties of session types.

Definition 2.1.1. Δ is *well-typed* if $s : S_1, \overline{s} : S_2 \in \Delta$ then $S_1 = \overline{S_2}$.

We use the well-typed definition to form the Subject Reduction theorem: If a process is well-typed then a reduction on that process results in a well-typed process.

Theorem 2.1.1 (Subject Reduction)**.** Let $\Gamma \vdash P \triangleright \Delta$ and Δ well-typed. If $P \twoheadrightarrow P'$ then $\Gamma \vdash P' \triangleright \Delta'$ and Δ' is well-typed.

Proof. The subject reduction theorem for session types was first proved in [YV07]. □

The subject reduction property ensures the soundness of the reduction of processes with respect to the typing system. The subject reduction typing system is used to prove various type safety results.

Multiparty Session Types: Binary session types offered the first understanding for the desired properties of a session interaction. Unfortunately binary session types could not maintain their properties when applied to communication with multiple participants. Multiparty Session types [HYC08, B$^+$08] were developed to provide a solution to this problem. Based on the idea of communication choreography, a multiparty session type describes in advance the communication protocol of a multi-participant process. Essentially, in the level of a single participant, multiparty session types introduce the ability to sequence a duality between different binary session channels. The basic idea is to construct a global communication session type between participants and project it to local session types for each participant.

We briefly provide a basic multiparty session type system based on a synchronous version of the system developed in [B+08].

$$\begin{array}{lcl} S & ::= & \texttt{bool} \mid \langle G \rangle \\ U & ::= & S \mid T \\ G & ::= & \texttt{p} \to \texttt{q} : \langle U \rangle . G \mid \texttt{p} \to \texttt{q} : \{l_i : G_i\} \mid \texttt{end} \mid t \mid \mu t . G \end{array}$$

We let $\texttt{p}, \texttt{q}, \ldots$ range over participants. Shared type S is either a value type or a shared name type. Type U denotes a shared type or a local type (i.e. session type). Global session type G is sequenced on terms that describe the sending of a value from a participant p to a participant q, a selection/branch term from a participant p to a participant q and the primitive recursor operator. Finally, global types include the recursion variable t and the inactive term end.

$$\begin{array}{lcl} T & ::= & [\texttt{q}]!\langle U \rangle ; T \mid [\texttt{q}]?(U); T \mid [\texttt{q}] \oplus \{l_i : T_i\}_{i \in I} \mid [\texttt{q}] \& \{l_i : T_i\}_{i \in I} \\ & \mid & \mu t . T \mid \texttt{end} \mid t \end{array}$$

Local types annotate the sequence of interactions from the point of view of a single participant. $[\texttt{q}]!\langle U \rangle ; T$ designates the sending of a value with type U to participant q and then continue with T. Similarly, the input local type $[\texttt{q}]?(U); T$ denotes the receiving of a value with type U and then continue as T. The select and branching local types offer a set of label selections (resp.

branching) towards participant q. The inactive session type is the standard end, while the recursion is described via the recursive variable t and the primitive recursor operator.

We define the local projection algorithm.

$$
\begin{aligned}
&\mathtt{p}' \to \mathtt{q} : \langle U \rangle . G \lceil \mathtt{p} \\
&\quad = \begin{cases} [\mathtt{q}]!\langle U \rangle ; G \lceil \mathtt{p} & \mathtt{p} = \mathtt{p}' \\ [\mathtt{p}']?(U); G \lceil \mathtt{p} & \mathtt{p} = \mathtt{q} \\ G \lceil \mathtt{p} & \text{otherwise} \end{cases} \\
&\mathtt{p}' \to \mathtt{q} : \{l_i : G_i\}_{i \in I} \lceil \mathtt{p} \\
&\quad = \begin{cases} [\mathtt{q}] \oplus \{l_i : G_i \lceil \mathtt{p}\}_{i \in I} & \mathtt{p} = \mathtt{p}' \\ [\mathtt{p}'] \& \{l_i : G_i \lceil \mathtt{p}\}_{i \in I} & \mathtt{p} = \mathtt{q} \\ G_1 \lceil \mathtt{p} & \text{if } \forall j \in I.\ G_1 \lceil \mathtt{p} = G_j \lceil \mathtt{p} \end{cases} \\
&(\mu t.G) \lceil \mathtt{p} = \begin{cases} \mu t.(G \lceil \mathtt{p}) & \mathtt{p} \in G \end{cases} \\
&t \lceil \mathtt{p} = t \quad \mathtt{end} \lceil \mathtt{p} = \mathtt{end}
\end{aligned}
$$

The projection operation of a global type G on a participant $\mathtt{p}$, denoted as $G\lceil \mathtt{p}$, returns a local type. The main idea is that the projection operator checks the prefix of G against the projection participant $\mathtt{p}$. If $\mathtt{p}$ is the sending (resp. selecting) participant then the projection results in a local sending (resp. selecting) prefix type and then proceed with projection inductively. Dually, if $\mathtt{p}$ is a receiving (resp. branching) participant the result is a local receiving (resp. branching) prefix type and then proceed with projection inductively. If the participant $\mathtt{p}$ is neither a sender nor receiver then the projection proceeds inductively. The projection algorithm stops in the $\mathtt{end}$ and t projections.

A type system for multiparty session types follows the same principles with the binary session typing system. Projection ensures the linearity and duality properties of interactions inside a multiparty session channel. For more details and different approaches on multiparty session types, see [HYC08] and [B+08].

2.2 Bisimulation Theory for the π-calculus

Program equivalence is one of the basic problems on the specification and verification of programs. The basic question that arises asks for the requirements for two programs to be equivalent. Such a question has different answers, justified by different philosophical discussions.

Process calculi contributed important results in the study of program equivalence. The structural nature of process calculi allowed for the definition of equivalence relations as mathematical objects, so way we could identify the basic and the desired properties for equivalences and study the relations between different definitions of equivalences.

A framework for the study of equivalences in process calculi arises if we represent the partial reduction of a process as a labelled graph, with the use of labelled transition semantics. A first equivalence relation is drawn out of the idea of trace equivalence, where two processes are considered equivalent if they have the same set of observed (i.e. labelled) traces in their corresponding transition graphs. The application of morphisms (i.e. functions between graphs) and especially homomorphisms (i.e. structure preserving functions) from a source graph to a target graph was the inspiring motive for developing the bisimulation relation. Homomorphisms preserve the structure of the target graph in the source graph, which is too restrictive when relating processes. In the search of a coarser relation the bisimulation relation was defined, which allows us to observe the structure of a graph only through its labelled edges, in contrast with homomorphism which also considers graph nodes as part of the graph structure (for a discussion of these notions, see [San09]).

Bisimulation for process calculus, was developed in the works of Park [Par81] and the study of equivalences on CCS [Mil89] by Milner [Mil80]. The most important property of bisimulation is its co-inductive definition. The co-inductive method defines the bisimilarity relation as the largest fix-point for a binary relation that relates processes that have co-directed graphs produced by the labelled transition system.

The core idea behind the co-inductive definition is for two processes to exhibit symmetric, matching interactions with their environment (i.e. exhibit the same observables).

A desired property for the bisimilarity relation is to be the largest equivalence relation that exhibits the observation-preserving, reduction-closed, congruence property [HY95].

In contrast to context-preserving relations, a bisimulation requires only local checks on a pair of process states, while in contrast to trace equivalence, it requires no hierarchy of checks. Furthermore, the computation of a bisimulation relation is decidible in a lower complexity class than other equivalence relations. The computational properties of co-induction make bisimilarity the finest and most successful equivalence over processes.

We proceed with the presentation of the basic bisimulation theory for the synchronous and the asynchronous π-calculus.

2.2.1 The π-calculus

The bisimulation for the π-calculus was introduced in the first work for the π-calculus in [MPW92] inspired by bisimulation works on other process calculi such as CCS [Mil89].

We briefly introduce the π-calculus bisimulation theory:

Syntax. The syntax of the π calculus follows the terms

$$P \ ::= \ \bar{x}\langle y\rangle.P \ \mid \ x(z).P \ \mid \ \mathbf{0} \ \mid \ P \mid P' \ \mid \ (\nu\, x)P \ \mid \ !P$$

A countable set of *Names* has $x, y, z, \ldots$ as its members. Process $\bar{x}\langle y\rangle.P$ denotes the capability to send name y over name x and continue with P. Name z is bounded in P by the receiving prefix $x(z).P$. When receiving occurs all the bounded occurrences of z in P are substituted with the receiving object. The inactive process $\mathbf{0}$ offers no operations. Parallel composition

places two processes in parallel. Operator $(\nu\ x)P$ restricts name x in the scope of P and the replicator $!P$ denotes the infinite use (i.e. infinite parallel composition) of process P. In process $\bar{x}\langle y\rangle.P$, x is called the subject and y the object of the send prefix.

Context and Congruence Relations.

Definition 2.2.1 (Context)**.** We define a context C as:

$$C \quad ::= \quad - \ \mid\ C \mid P \ \mid\ (\nu\ x)C \ \mid\ \bar{x}\langle y\rangle.C \ \mid\ x(z).C$$

With $C[P]$ to denote the replacement of $-$ into C by P.

A relation R is a congruence if it is closed under the context definition, i.e. $\forall C$ if $P\ R\ Q$ then $C[P]\ R\ C[Q]$.

Structural Congruence. Structural congruence for the π-calculus follows the rules for structural congruence relation in § 2.1.1, without the recursion rules and by adding:

$$\vdots$$
$$!P \equiv P \mid !P$$

where replication is structurally defined as an infinite parallel composition.

Labelled Transition System. We define a set of labels as

$$\ell \ ::= \ \bar{x}\langle y\rangle \ \mid\ \bar{x}(y) \ \mid\ x(y) \ \mid\ \tau$$

with $\bar{x}\langle y\rangle \asymp x(y)$ and $\bar{x}(y) \asymp x(y)$

The send label $\bar{x}\langle y\rangle$ denotes the send of name y on name x, while the bound send label $\bar{x}(y)$ sends a bound name y on name x. The receive label $x(y)$ denotes the reception of name y on

name x. Finally, label τ is the hidden or internal action. Send and receive actions are dual if they have the same object and subject.

We define the labelled transition system for the synchronous π-calculus:

$$\langle\text{Out}\rangle \quad \overline{x}\langle y\rangle.P \xrightarrow{\overline{x}\langle y\rangle} P \qquad \langle\text{In}\rangle \quad x(z).P \xrightarrow{x(y)} P\{y/z\}$$

$$\langle\text{ParL}\rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \langle\text{ParR}\rangle \quad \frac{Q \xrightarrow{\ell} Q' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(P) = \emptyset}{P \mid Q \xrightarrow{\ell} P \mid Q'}$$

$$\langle\text{Tau}\rangle \quad \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\ \mathtt{bn}(\ell) \cup \mathtt{bn}(\ell'))(P' \mid Q')} \qquad \langle\text{Alpha}\rangle \quad \frac{P \xrightarrow{\ell} Q \quad P \equiv_a P'}{P' \xrightarrow{\ell} Q}$$

$$\langle\text{Name}\rangle \quad \frac{P \xrightarrow{\ell} P' \quad a \notin \mathtt{fn}(\ell)}{(\nu\ a)P \xrightarrow{\ell} (\nu\ a)P'} \qquad \langle\text{Extr}\rangle \quad \frac{P \xrightarrow{\overline{b}\langle a\rangle} P'}{(\nu\ a)P \xrightarrow{\overline{b}(a)} P'}$$

We can observe the interaction of the send and receive prefixes with the environment on the corresponding send and receive labels. The send action is observed on a process $\overline{x}\langle y\rangle.P$, with name y being sent on channel x. The receive action on process $x(y).P$ requires the substitution function to substitute all bound occurrences of z to y in P. In a parallel composition a process P can act independently from its parallel Q, provided that there are no common names between the action's ℓ bound names and Q's free names. If two parallel processes can interact on dual actions $\ell \asymp \ell'$, then they can interact together on the τ action. Transitions happen with respect to alpha renaming. An action is independent from scope restriction if no restricted name is in the free names of the action. A name is *extruded* from restriction when the restricted name is send to the environment. Furthermore, the label for extrusion requires the send of a bound name.

We write $\longrightarrow$ for $\xrightarrow{\tau}$. We extend to $\Longrightarrow$ for the reflexive and transitive closure of $\longrightarrow$. We write $\overset{\ell}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\ell}\longrightarrow$ and $\overset{\hat{\ell}}{\Longrightarrow}$ for $\overset{\ell}{\Longrightarrow}$ if $\ell \neq \tau$ and $\Longrightarrow$ for $\ell = \tau$.

Bisimulation The labelled transition system is used to define different bisimilarity relations. We begin with the definition of simulation.

Definition 2.2.2 (Strong Simulation)**.** Let $\mathscr{R}$ be a binary relation over processes. $\mathscr{R}$ is called strong simulation if whenever $P\mathscr{R}Q$ then if $P \xrightarrow{\ell} P'$ then $\exists Q' \cdot Q \xrightarrow{\ell} Q'$ and $P'\mathscr{R}Q'$

If $\mathscr{R}$ and $\mathscr{R}^{-1}$ are both simulations then we say that $\mathscr{R}$ is a bisimulation.

Definition 2.2.3 (Strong Bisimulation)**.** Let $\mathscr{R}$ be a binary relation over processes. $\mathscr{R}$ is called strong bisimulation if whenever $P\mathscr{R}Q$ then

1. If $P \xrightarrow{\ell} P'$ then there exists Q' such that $Q \xrightarrow{\ell} Q'$ and $P'\mathscr{R}Q'$
2. If $Q \xrightarrow{\ell} Q'$ then there exists P' such that $P \xrightarrow{\ell} P'$ and $P'\mathscr{R}Q'$

The union of all strong bisimulation relations is called strong bisimilarity denoted as $\sim$.

A weaker bisimulation relation ignores the hidden (τ) transitions.

Definition 2.2.4 (Weak Bisimulation)**.** Let $\mathscr{R}$ be a binary relation over processes. $\mathscr{R}$ is called weak bisimulation if whenever $P\mathscr{R}Q$ then

1. If $P \xrightarrow{\ell} P'$ then there exists Q' such that $Q \stackrel{\hat{\ell}}{\Longrightarrow} Q'$ and $P'\mathscr{R}Q'$
2. If $Q \xrightarrow{\ell} Q'$ then there exists P' such that $P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ and $P'\mathscr{R}Q'$

The union of all weak bisimulation relations is called a weak bisimilarity denoted as $\approx$.

The bisimulation relation can be equivalently defined as the greatest fix-point produced by the labelled transition monotone function ($\xrightarrow{\ell}$).

Definition 2.2.5 (Stratification of Bisimulation)**.** Let $\mathscr{P}$ be the set of all processes

1. $\approx_0 = \mathscr{P} \times \mathscr{P}$

2. $P \approx_n Q$ if

   - If $P \xrightarrow{\ell} P'$ then $Q \xRightarrow{\hat{\ell}} Q'$ and $P' \approx_{n-1} Q'$.
   - If $Q \xrightarrow{\ell} Q'$ then $P \xRightarrow{\hat{\ell}} P'$ and $P' \approx_{n-1} Q'$.

3. $\approx^{\omega} = \bigcap_{n \leq 0} \approx_n$.

From the Knaster-Tarski theorem we know that $\approx^{\omega}$ is the greatest fix point on the lattice created by the transition relation on co-directed pairs of processes. Due to the image finiteness of the LTS for the synchronous π-calculus[1], we can derive that $\approx^{\omega} = \approx$, since $\approx$ is the largest bisimulation relation.

Observational Theory: Barbs are defined to study the observables of π-calculus processes.

Definition 2.2.6 (Barbs)**.** We say that we observe a barb x on P, denoted $P \downarrow_x$ if

1. $P \downarrow_{\bar{x}}$ if $P \equiv (\nu\ \tilde{w})(\bar{x}\langle y\rangle.P_1 \mid P_2), x \notin w$

2. $P \downarrow_x$ if $P \equiv (\nu\ \tilde{w})(x(y).P_1 \mid P_2), x \notin w$

We define a congruence relation on processes, that preserves barbs and is reduction closed, which is used as an equivalence relation on processes.

Definition 2.2.7 (Reduction-closed Congruence)**.** Let $\mathscr{R}$ be a binary relation. We say that $\mathscr{R}$ is a reduction-closed congruence whenever $P\ \mathscr{R}\ Q$ then:

1. $P \downarrow_x$ if and only if $Q \downarrow_x$.

[1] An intuition about an image finite relation $\mathscr{R}$ comes when for all processes P the set $\{P' \mid P \mathscr{R} P'\}$ is finite. For the definition of image finitness and a proof of the fact that the labelled transition system $\xrightarrow{\ell}$ of the synchronous π-calculus is image finite, see [SW01, § 1]

2. $P \longrightarrow^* P'$ if and only if $Q \longrightarrow^* Q'$ and $P' \mathscr{R} Q'$.

3. $\forall C, C[P] \mathscr{R} C[Q]$.

The union of all reduction-closed congruence relations is denoted with $\cong$.

It is desirable for the bisimilarity relation to exhibit congruence properties.

Lemma 2.2.1. $\approx$ is a non-input congruence.

Proof. A proof can be found in [SW01, § 2] □

Congruence in the bisimilarity relation is preserved by all contexts except the input context. This is due to the effect name substitution has on processes. The most desirable property for the bisimilarity is that it coincides with the barbed-preserving, reduction closed congruence relation [HY95]. As we can see from Lemma 2.2.1, this does not hold for synchronous weak bisimilarity.

We could however take advantage of the fact that name substitution does not respect input congruence to define a congruent synchronous weak bisimilarity relation as follows:

Definition 2.2.8. We define $P \approx_c Q$ if $P\sigma \approx Q\sigma$ for all name substitutions σ.

Theorem 2.2.1. $\approx_c = \cong$

Proof. A proof can be found in [SW01, §2] □

2.2.2 The Asynchronous π-calculus

The asynchronous π-calculus was proposed independently by Honda and Tokoro [HT91b] and by Boudol [Bou92] along with a corresponding bisimulation theory. In [ACS98] there is an extensive study for the bisimulation for the asynchronous π-calculus. We present the asynchronous π-calculus and two different labelled transition systems that give rise to corresponding bisimulation definitions.

Syntax:

$$P \;::=\; \bar{x}\langle y\rangle \;\mid\; x(z).P \;\mid\; \mathbf{0} \;\mid\; P \mid P' \;\mid\; (\nu\, a)P \;\mid\; !P$$

We present the syntax for the asynchronous π calculus (cf. [HT91b]). The asynchronous π-calculus achieves asynchrony, by restricting the continuation of the send prefix $\bar{x}\langle y\rangle$ in contrast with the syntax of the synchronous π-calculus. This restriction allows for the unordered transmission of names, since we cannot impose any order on them using a sequential operator.

Structural congruence is identical with the structural congruence for the synchronous π-calculus.

Labelled Transition System: The labelled transition system (abbrev. lts of LTS) for the asynchronous π-calculus was also introduced by Honda and Tokoro [HT91b]. Its definition implies that at any moment a process can perform an input action.

The definition of the LTS assumes the replacement of rule $\langle \text{Out}\rangle$ in the synchronous LTS with the following rule:

$$\langle \text{Out}\rangle_a \quad \bar{x}\langle y\rangle \xrightarrow{\bar{x}\langle y\rangle} \mathbf{0}$$

The second rule that allows the observation of inputs at any point of the process transition comes by replacing rule $\langle \text{In}\rangle$ in the synchronous LTS with:

$$\langle \text{In}\rangle_a \quad \mathbf{0} \xrightarrow{x(y)} \bar{x}\langle y\rangle$$

We also need to change the definition of the rule $\langle \text{Tau}\rangle$ to handle name substitution, since substitution is not observed in the $\langle \text{In}\rangle_a$ rule.

$$\langle \text{Tau}\rangle_a \quad \bar{x}y \mid x(z).Q \xrightarrow{\tau} P' \mid Q'\{y/z\}$$

Weak asynchronous bisimulation is defined the same way as with bisimulation for the synchronous π-calculus:

Definition 2.2.9 (Weak Asynchronous Bisimulation)**.** Let $\mathscr{R}$ be a binary relation over the asynchronous π-calculus processes. $\mathscr{R}$ is called weak bisimulation if whenever $P\mathscr{R}Q$ then

1. If $P \xrightarrow{\ell} P'$ then there exists Q' such that $Q \stackrel{\hat{\ell}}{\Longrightarrow} Q'$ and $P'\mathscr{R}Q'$
2. If $Q \xrightarrow{\ell} Q'$ then there exists P' such that $P \stackrel{\hat{\ell}}{\Longrightarrow} P'$ and $P'\mathscr{R}Q'$

The union of all asynchronous weak bisimulation relations is called weak bisimilarity denoted as $\approx_a$.

The asynchronous bisimulation theory and its properties can be found in [HT91b]. The asynchronous bisimulation theory and its variations are studied in [HY95]. This paper introduces some of the most significant properties for bisimulation theory in the presence of process calculus, including the soundness and completeness of the asynchronous weak bisimilarity with respect to barbed-preserving, reduction-closed congruence.

A different labelled transition system was proposed by Amadio et al. in [ACS98]. This labelled transition system disregards the fact that input actions can happen at any time imposing the input action rule:

$$\langle \text{In} \rangle \qquad x(z).P \xrightarrow{x(y)} P\{y/z\}$$

In [ACS98] different variants of the asynchronous bisimulation are discussed and compared with the bisimulation semantics from [HT91b]. The main result is the coincidence of the bisimilarity with the barbed-preserving, reduction-closed congruence.

2.2.3 Type Systems and Advanced Behavioural Theory for the π-Calculus

Type Systems: Type theory was developed as the basic meta-theory for the study of the dynamics of computational models. The core type theory for the π-calculus is suggested directly from well known and applied type systems for the λ-calculus, which gives us a basic inside for the importance of the π-calculus as a process model and as a prospective model for programming languages.

A first type system was the sorting system [Mil92], which defines a sorting classification on the names of the polyadic π-calculus and which is used for statically infering some basic properties between the subjects and the objects of the actions of the calculus.

Sorting is a typing system that equates two types (and thus the meta-information of π-calculus names) if they have the same name. The introduction of structure into sorts gave rise to a typing system able to describe data structures such as products, unions, records and variants [SW01]. The `i/o` (input/output) type system [PS96] is another basic type system for the π-calculus. In the `i/o` type systems, sorts are annotated with the input and/or the output capability and their equality is based on on the way sorts with the same `i/o`-capability are structured. The basic intuition for `i/o`-types, requires from a typing environment to control the read and write access on names, in order to disallow unintended process behaviour. Furthermore, the structured nature of `i/o`-types allows for the definition of a subtyping theory.

The linear typing system for the π-calculus [KPT99] is based on definition of the linear logic theory [Gir87], where the names of the π-calculus are treated as linear resources. Linear types are a refinement of the `i/o`-types that allow for the usage of the `i/o`-capability on a name only once. The work in [DGS12] proposes an encoding of session types into linear and variant typed π-calculus and closes the relation between linear types and session types.

Typed Bisimulation Theory: Typed bisimulation was studied in the context of `i/o`-types in [HR04]. The paper first develops a framework for may/must testing for the `i/o`-typed π-

calculus, but the basic contribution begins with the proposal of a typed label transition system that is controlled by the i/o-type environment in the presence of subtyping. The LTS is then used to define a bisimulation relation, where bisimilarity is the maximal reduction-closed congruence that preserves observation.

Bisimulation for the Higher Order π-calculus: The higher order π-calculus (HOπ) was originally introduced in [San92]. The HOπ allows for processes to be carried as messages and was inspired by the ability of the λ-calculus to carry λ-terms as parameters. A significant result in [San92] is the full abstraction theorem for an encoding of the HOπ in the first order π-calculus.

An advanced study of bisimulations for higher order languages can be found in [SKS11]. The study is concerned with various λ-calculi and the HOπ. The paper initially argues that higher order bisimulation relations are a hard subject to reason about, since the higher order values that can be observed on an action are produced by a large universe of classes. To overcome theses difficulties and limit the universe of the observed higher order values, the authors develop the environmental bisimulation that uses an environment observer to track the higher order values that were produced earlier by the tested processes.

A bisimulation theory for a higher order π-calculus with cryptographic primitives is presented in [KH11]. The main innovation on this approach is that the authors use an indexed observer set to keep track of higher order knowledge and then they propose a label transition system that observes the index of a higher value instead of the higher order value itself. The latter definition reduces a higher order labelled transition system to a first order lts and allows for a simpler bisimulation definition.

Confluence for the π-calculus: Concurent systems have in general a non-deterministic execution and as a consequence they are considered more complex than sequential systems. However it is often the case that a concurrent program will have a predictable and well formed

behaviour when it comes to the succession of program states during its execution. Such observations motivate the development of the confluence theory for process calculi. Confluent was initially studied for CCS in [Mil80, Mil89] and a confluence theory for the π-calculus was studied in [PW97]. Confluence requires that the execution of an action from a process will not preclude the execution of another action up-to the behaviour of the process, e.g the competition for accessing a name by two actions may result in precluding one of the two, (thus making the system non-confluent) and may affect the subsequent behaviour of the process. The confluence property is used for reasoning about systems, since confluent systems enjoy a number of *good* properties, such as the semantic invariant up-to internal actions and the coincidence of trace equivalence with bisimulation [Mil89].

2.3 Event Driven Programming

Event-driven programming is one of the major paradigms that utilise concurrent and communication based programming. Event-driven concurrency can be defined as a model around the notion of events. In general an event is a computation state change (i.e. an action) that can happen asynchronously and concurrently to the computation. It is furthermore characterised by detectability, in a sense that the computation itself can detect and react on an event action. More specifically events in message passing-based programming, are typically detected as the arrival of messages on asynchronous communication channels.

Interrupts. A first implemented notion of an event action was the interrupt, used as the basic mechanism to handle concurrency in a computing machine. At the operating system level, interrupts are used to coordinate the machine's resources. An interrupt is the asynchronous interruption of the instruction flow inside the central processing unit and can be issued at hardware level after the completion of different input/output operations inside a machine (called hard interrupts). Interrupts can also be issued by the code running in the CPU

itself (called soft interrupts) and are primarily used by programs to perform system calls and multi-core coordination. Upon an interrupt the program counter is stored, the CPU mode usually changes to kernel mode, the instruction flow stops and transferred into an interrupt handling code, responsible for detecting the type of the interrupt and handling its side effects.

Interrupts were developed as a fundamental and necessary hardware function for the coordination of concurrent resources and their impact was stratified in all the computation levels, from the hardware level to the operating system and the application level. Naturally interrupt characteristics have influenced the way programs are written. For example a low level interrupt on a communication device in the hardware level can be specified as a message arrival on an abstract, programming entity called channel in the application level. The characteristics of interrupts are passed into an abstract application layer entity called event. Many programming models, libraries and frameworks were and are being developed around the event entity, that give rise to a constant ongoing discussion for the trade-off of the different approaches.

Actors. The actors model was one of the first models developed using the event notion, as an expressive event-driven programming model. It was first presented as a formalism for artificial intelligence [HBS73] and was further developed as a full programming model [Cli81, Agh86]. The actor programming model requires a set of concurrent and communicating entities called actors. An actor is an object that encapsulates state, functionality and control flow. Among other computational functions an actor can create new actors, pass messages to other actors and react to the arrival of messages from other actors.

Erlang [VWW96] is one of the first and widely used programming languages that support the actor model characteristics. Erlang is a communication-based language, based on light-weight processes (actors). Communication is defined using the mailbox abstraction: each process asynchronously receives messages from other processes in its own mailbox. A mailbox structure allows pattern matching to recognise at runtime a message type.

Another widely used framework implementing the actors model is the Scala programming language [OAC+06], a purely object-oriented language that treats objects as concurrent actors. Scala was initially implemented on the Java Virtual Machine [LY99]. A runtime library Scala [HO08], unifies event-based programming with the thread-based programming, by using closure to suspend threads on blocking (receiving) operations.

Task and Stack Management. Following the terminology in [AHT+02], event-driven handling can be analysed in a two axis space, notably the task management axis and the stack management axis. Task management describes the way a task is scheduled in a processor: i) serial task management runs each task to completion, ii) pre-emptive task management allows for tasks to interleave in a processor and iii) cooperative task management where the execution from a task is passed to another task on well defined, usually blocking, points in execution. To achieve cooperative task management, a programmer organises a program into a set of blocking points, and associates each of them with an event handler. More specifically in each execution blocking point, the program proceeds with an asynchronous function call. The signal for the asynchronous function call is associated with an event handler and the caller function closure and its continuation are stored in the heap. To achieve event-driven control the caller function closure pointer, its continuation and the event are registered in an event-handling structure, able to notify for the completion of the blocking function. This technique is called manual stack management or stack ripping. A different approach, supported by some programming frameworks, is automatic stack management where the programmer can explicitly use system operations for handling asynchronous blocking calls and stack ripping.

The Concurrency Dichotomy - The Lauer Needham Duality. In 1979 Lauer and Needham [LN79] observed that there is a duality in the way concurrent programs can be expressed. The dichotomy is defined between a thread-based programming style and an event-based style. In their work they develop two empirical sets of programming primitives, with each set corresponding to one of the two models.

The thread-based model requires a multi-threaded approach, implemented with thread handling primitives such as the fork and join operations and equipped with shared memory primitives for thread communication and coordination. On the other hand the event-based model is based on a single instruction stream that consists of an event handling loop routine. Communication is defined using message passing primitives. The event loop is responsible for detecting messages (events) and proceeding with their handling according to their type.

The argument for the duality between the the models is made by expressing the primitives of one model in the terms of other. The authors in their discussion around the impact of this dual expressiveness, argue that there is no general way to decide, or better yet to justify which of the two approaches a programmer should use for a concurrent implementation, other than the nature of the underlying hardware architecture.

Events vs Threads. The purely empirical justification of the concurrency duality by Lauer and Needham, ignited a series of discussions where the researchers were (and are) trying to impose more abstract and philosophical justifications, taking into consideration different reasons, in favour of the one or the other approach. In 1996 a presentation with the title *"Why threads are a bad idea"* [Ous96], argues in favour of event-driven programming mainly by citing reasons against threaded programming. Threads suffer from deadlock problems and are difficult to synchronize. Synchronization introduces performance issues in a trade-off between complex fine-grain locks and low performance coarser synchronization techniques. Threads are hard to abstract as a single computation stream and thus a sequential debugging execution would be rather obfuscated. On the other hand the event-driven model minimizes synchronization and synchronization related problems in a simple and easy to understand, single stream program that uses callbacks to handle events.

The antithesis in Oysterhout's presentation came in 2003 in a paper called *"Why events are a bad idea"* [vBCB03], where the authors expose the virtue of a simpler and a more natural programming model for thread programming. They argue that we can overcome the event-driven

programming problems with *good* implementations and framework support. Specifically they discuss and propose methods for solving problems related to performance, synchronization, control flow stack management and scheduling.

Different programming frameworks were proposed, that implement the characteristics discussed in the threads-events controversy. A closer analysis will indicate that these different models emerged from the dialectical interaction of the two paradigms.

The explicit event library [CK05] provides a library interface used for cooperative task management and manual stack management, while it provides tools for software analysis and verification. Tame [KKK07] is a set of libraries and a source to source implementation in C++, that implements event-driven programming without explicit stack ripping. It introduces language features, similar to the synchronisation mechanisms of futures, that allow control flow to be returned from a blocked C++ function to the caller. Cappricio [vB+03] follows [vBCB03] in a thread-oriented implementation. It uses compiler transformations of user-level thread code, and replaces blocking functions with non-blocking equivalents to take into advantage the notion of cooperating thread management. A hybrid threads-event system developed for Haskell [LZ07] claims the *best of both worlds*. The programmer in the application level has the interface for thread-based programming and an exposed interface for a thread scheduler, that allows the programmer to handle threads as event-driven entities. Event Java [EJ09] integrates event correlation with object-oriented programming to provide high-level syntax for expressing complex patterns of predicated events. A type-safe event-driven session programming framework based called Session Java [HKP+10], counters the problems of traditional event-based programming with abstractions and safety guarantees based on session types. In the context of high performance, scalable web services OKWS [Kro04] describes an operating system level event-driven architecture with emphasis on security. A most influential, highly scalable architecture for web services is Staged Event-driven Architecture (SEDA) [WCB01], describes events as a pipeline of entities called stages. Every stage consists of an event queue, an event scheduler and a thread pool. Stages are responsible for processing a

blocking call. After the completion of the blocking call an event is registered in the next stage of the pipeline.

Part I

Chapter 3

Asynchronous Session Types Behavioural Theory

We developed a core theory for Asynchronous Session types, which is used as the the vehicle for the study of asynchronous semantics [HT91b] in the presence of session types [HVK98, MY09]. In this chapter we intend to present a minimal calculus, able to grasp an applied aspect of asynchronous communicating, while having as a final goal the study of the bisimulation framework for asynchronous session typed programs. Such a minimal calculus should be able to describe or extended to describe (chapters 4 and 5) asynchronous communication operators and frameworks.

To define the Asynchronous Session π calculus or ASP for short, we focus on the following characteristics for communication:

- **Buffered Communication.** Modern network transport, such as TCP, provide reliable, order-preserving delivery of messages once a connection is established. This is achieved with the use of intermediate memory buffers for storing communication data. We explicitly define queues as communication media for the exchange of messages.

- **Fine grained asynchronous session communication.** We semantically define first-in first-out (FIFO) communication queues, called endpoint configurations, as session endpoints to achieve the non-blocking property of asynchrony and the order-preserving property of session types.

- **Asynchronous session initiation.** Session types ordered message delivery contrasts with session initiation that relies on unordered shared name interactions. We use a combination of buffered communication and the asynchronous π-calculus [HT91b] semantics to describe asynchronous session initiation on shared names. Session initiation follows the principles of the asynchronous π-calculus. The asynchronous π-calculus semantics allow for an unordered session initiation inside the network and together with buffered initiation operations we get a more accurate description of a more applied communication framework.

- **A sound session type system in the presence of session subtyping.** We develop a session type system in the presence of subtyping [GH05] based on [YV07] for typing programs (i.e. closed processes with no configurations) and a novel session type for configuration endpoints (cf. [MY09]). A subject reduction theorem proves type soundness and a type safety theorem lists the cases where error-free progress can be guaranteed.

To study the bisimulation theory for ASP processes we develop a labelled type system on the ASP processes and a labelled transition system on session environments. The two labelled transition systems are combined together to give the typed transition definition, used to define the bisimulation theory for typed ASP processes. The bisimilarity relation is sound and complete with respect to a corresponding congruence relation on typed processes. In the later chapters of this dissertation we intent to reason about typed distributed systems. This gives us a motivation to develop a basic confluence theory (based on [PW97]) defined using the typed bisimilarity. The basic result extracted from confluence theory is that session interactions are

confluent.

3.1 A Core Process Model for Asynchronous Sessions

In this section we present the syntax and operational semantics for a core calculus based on the π-calculus with session primitives [HVK98, MY09]. We define the basic asynchronous communication semantics using a structure of `i/o` *message queues* which guarantee order-preserving delivery of messages inside a session. The operational semantics also allow asynchronous and unordered session initiation using a send construct with no continuation [HT91a].

3.1.1 Syntax of the Asynchronous Session π-Calculus

Processes. Figure 3.1 gives the syntax of the ASP processes. We explain session endpoint configurations containing localised `i/o`-queues and introduce asynchronous session initiation (cf. synchronous session initiation in preceding works [HVK98, YV07]). The syntax defined in the the last four terms of the BNF for (Processes) is called *run-time syntax*. Closed terms (bound and free variables are explained below) that are not structured on run-time syntax, are called *programs*.

Values $(v, v', \ldots)$ include the constants, shared channels $(a, b, c, \ldots)$ and session channel endpoints $(s, s', \ldots, \overline{s}, \overline{s}', \ldots)$. A session channel endpoint s designates one endpoint of a session, and $\overline{s}$ the opposing end of the same session. We often shorten "session channel endpoint" (i.e. the programming/runtime entity at a local configuration used to perform session actions) to just "session channel" for brevity, and we set $\overline{\overline{s}}$ to be s. Branch/select labels (simply labels) range over $l, l', \ldots$, variables range over x, y, z, and recursion variables range over X, Y, Z. Shared channel identifiers $(u, u', \ldots)$ are shared channels and variables; session identifiers

(Processes)	P,Q	$::=$	$u(x).P$	Accept
		$\mid$	$\overline{u}(x).P$	Request
		$\mid$	$k!\langle e\rangle;P$	Sending
		$\mid$	$k?(x);P$	Receiving
		$\mid$	$k \oplus l;P$	Selection
		$\mid$	$k\&\{l_i : P_i\}_{i\in I}$	Branching
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
		$\mid$	$(\nu\ a)P$	Hiding
		$\mid$	$P \mid Q$	Parallel
		$\mid$	$\mu X.P$	Recursion
		$\mid$	X	Variable
		$\mid$	$\mathbf{0}$	Inaction
		$\mid$	$a[\vec{s}]$	Shared Configuration
		$\mid$	$\overline{a}\langle s\rangle$	Asynchronous Request
		$\mid$	$(\nu\ s)P$	Session Hiding
		$\mid$	$s[\texttt{i} : \vec{h}, \texttt{o} : \vec{h'}]$	Session Configuration
(Identifiers)	u	$::=$	$a,b,c \mid x,y,z$	
	k	$::=$	$s,\overline{s} \mid x,y,z$	
	n	$::=$	$a,b,c \mid s,\overline{s}$	
(Values)	v	$::=$	$\texttt{tt},\texttt{ff} \mid a,b,c \mid s,\overline{s}$	
(Expressions)	e	$::=$	$v \mid x,y,z \mid e = e \mid e \wedge e \mid \ldots$	
(Messages)	h	$::=$	$v \mid l$	

Figure 3.1: The syntax of ASP processes.

$(k, k', \ldots)$ are session channels and variables. A session message h is either a value or a label. Expressions e are either values, variables, logical operators on Boolean expressions, or the equality operator on values. Note that the equality operator also corresponds to name matching. We write $\vec{s}$ and $\vec{h}$ for the respective vectors of session channels s and messages h, and ε for the empty vector.

The session initiation actions on shared channels are the request $\overline{u}(x).P$ and the accept $u(x).P$. On an established session channel, output $k!\langle e\rangle;P$ sends the value denoted by e through channel k, input $k?(x);P$ receives a value through k, selection $k \oplus l;P$ chooses and sends the label l through k, and branching $k\&\{l_i : P_i\}_{i\in I}$ follows the branch with the label received through k. Construct `if` e `then` P `else` Q is used for expression branching. The $(\nu\ u)P$ binder restricts a shared channel u to the scope of P. $P \mid Q$ is the standard π calculus parallel operator. Recursion is handled with operator $\mu X.P$ and process variable X using the standard λ calculus notation. The inactive process is written as $\mathbf{0}$.

The Asynchronous Session π calculus incorporates two forms of asynchronous communication, *asynchronous session initiation* [Kou09] and *asynchronous session communication* (over an established session). The former models the *unordered* transport of session *request messages* to acceptors (servers) listening on a shared channel. We use $\overline{a}\langle s\rangle$ to represent a request message in transit on shared channel a, carrying the initiation request for a (fresh) session channel s. In network communications in practice, messages are buffered for reading on arrival at the destination. This mechanism is formalised by introducing a *shared input buffer* $a[\vec{s}]$, which represents an acceptor's input buffer at a containing pending requests for sessions $\vec{s}$.

Communication in an established session is asynchronous but *order-preserving*, as in a TCP session. For this purpose, each session channel s is associated with an *endpoint configuration* (or simply, configuration) $s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h}']$, which encapsulates both input (`i`) and output (`o`) message queues. Sending a message first enqueues it at the source `o`-queue before it is eventually transferred to the destination `i`-queue, signifying the arrival of that message. For brevity, one

or more components may be omitted from a configuration when they are irrelevant, e.g. we may use $s[\mathtt{i} : \vec{h}]$ as an abbreviation of $s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h}']$ when only the $\mathtt{i}$-queue is required. The $(\nu\ s)P$ binder restricts both session channels s *and* $\overline{s}$, i.e. both endpoints of the session, to the scope of P. The process terms specified in Figure 3.1 that feature s also apply to $\overline{s}$.

The notions of , bound $\mathtt{bn}(P)$ and free names $\mathtt{fn}(P)$ and variables $\mathtt{bv}(P), \mathtt{fv}(P)$ are standard with the extension to treat $\mathtt{fn}(\overline{a}\langle s \rangle) = \{a, s\}$, $\mathtt{fn}(a[\vec{s}]) = \{a, s_1, \ldots, s_n\}$ and $\mathtt{fn}(s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h}']) = \{s\}$.

$$
\begin{array}{rcll}
P & \equiv & Q & \text{if } P \equiv_\alpha Q \\
P \mid Q & \equiv & Q \mid P & \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
P \mid \mathbf{0} & \equiv & P & \\
(\nu\ n)P \mid Q & \equiv & (\nu\ n)(P \mid Q) & \text{if } n \notin \mathtt{fn}(Q) \\
\mu X.P & \equiv & P\{\mu X.P/X\} & \\
(\nu\ n)\mathbf{0} & \equiv & \mathbf{0} & \\
(\nu\ a)a[\varepsilon] & \equiv & \mathbf{0} & \\
(\nu\ s)(s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) & \equiv & \mathbf{0} &
\end{array}
$$

Figure 3.2: Structural congruence for ASP.

We define structural congruence as the smallest congruence on processes generated by the rules in Figure 3.2. Most of the rules are standard. The first rule says that processes are equivalent up-to alpha-conversion commutativity and associativity in the next two rules. A process in parallel with the inactive process is structurally congruent to itself. A name private for the process P in the process $P \mid Q$, can also be private for the entire process if it does not occur free in Q. Recursion is defined inside structural congruence as the substitution of the recursive process on the recursive process variable in the actual process. Restriction of names in the inactive process is congruent with the inactive process. The last two rules are

for garbage collecting empty shared channel buffers and configurations, when they are not in further use.

3.1.2 Operational Semantics of the Asynchronous Session π-Calculus

The reduction relation is defined on terms with no free variables and it is denoted using the infix symbol $\longrightarrow$. The operational semantics capture the asynchronous and session nature of the ASP processes. The messages that are communicated on session channels follow a FIFO policy inside the session endpoint configuration.

In the reduction relation definition, we use the standard evaluation contexts, $E[-]$ defined as:

$$E \ ::= \ s!\langle - \rangle; P \ \mid \ \texttt{if} \ - \ \texttt{then} \ P \ \texttt{else} \ Q$$

where the $-$ can be substituted by the expression e, in $E[e]$.

Figure 3.3 lists the reduction rules. The first three rules are used for session initiation. Rule [Request1] issues a new request for a session of type S via shared channel a. A fresh (i.e. ν-bound) session with endpoints s (acceptor-side) and $\overline{s}$ (requester-side) and the initial configuration at the requester are generated, dispatching the session request message $\overline{a}\langle s \rangle$. [Request2] enqueues the request in the shared input buffer at a. [Accept] dequeues the first session request, substitutes the bound session variable with the s in the request message, and creates the acceptor-side configuration: the new session is now established between the requester and acceptor.

The next five rules are for in-session communication. As described earlier, to send a message, rule [Send] enqueues a value in the o-queue of the *local* configuration and removes the output prefix from the current active type, signifying the completion of this action. [Receive] dequeues the first value from the i-queue of the local configuration and again updates the active type accordingly. [Sel] and [Bra] similarly enqueue and dequeue a label, using the label

$$\frac{(s \notin \mathtt{fn}(P))}{\overline{a}(x).P \longrightarrow (\nu\ s)(P\{\overline{s}/x\} \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid \overline{a}\langle s\rangle)} \quad [\text{Request1}]$$

$$a[\vec{s}] \mid \overline{a}\langle s\rangle \longrightarrow a[\vec{s} \cdot s] \quad [\text{Request2}]$$

$$a(x).P \mid a[s \cdot \vec{s}] \longrightarrow P\{s/x\} \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid a[\vec{s}] \quad [\text{Accept}]$$

$$s!\langle v\rangle; P \mid s[\mathtt{o} : \vec{h}] \longrightarrow P \mid s[\mathtt{o} : \vec{h} \cdot v] \quad [\text{Send}]$$

$$s?(x); P \mid s[\mathtt{i} : v \cdot \vec{h}] \longrightarrow P\{v/x\} \mid s[\mathtt{i} : \vec{h}] \quad [\text{Receive}]$$

$$\frac{(i \in J)}{s \oplus l_i; P \mid s[\mathtt{o} : \vec{h}] \longrightarrow P \mid s[\mathtt{o} : \vec{h} \cdot l_i]} \quad [\text{Select}]$$

$$\frac{(i' \in J \subseteq I)}{s\&\{l_i : P_i\}_{i \in I} \mid s[\mathtt{i} : l_{i'} \cdot \vec{h}] \longrightarrow P_{i'} \mid s[\mathtt{i} : \vec{h}]} \quad [\text{Branch}]$$

$$s[\mathtt{o} : v \cdot \vec{h}] \mid \overline{s}[\mathtt{i} : \vec{h}'] \longrightarrow s[\mathtt{o} : \vec{h}] \mid \overline{s}[\mathtt{i} : \vec{h}' \cdot v] \quad [\text{Comm}]$$

$$\mathtt{if\ tt\ then}\ P\ \mathtt{else}\ Q \longrightarrow P \quad [\text{If-true}]$$

$$\mathtt{if\ ff\ then}\ P\ \mathtt{else}\ Q \longrightarrow Q \quad [\text{If-false}]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \quad [\text{Eval}]$$

$$\frac{P \longrightarrow P'}{(\nu\ a)P \longrightarrow (\nu\ a)P'} \quad [\text{Chan}]$$

$$\frac{P \longrightarrow P'}{(\nu\ s)P \longrightarrow (\nu\ s)P'} \quad [\text{Sess}]$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad [\text{Par}]$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} \quad [\text{Struct}]$$

Figure 3.3: Reduction rules for ASP.

to select the appropriate case in the active type. Note that these four rules manipulate only the local configurations, and output actions are always non-blocking. The actual transmission of a session message is embodied by [Comm], which removes the first message from the o-queue of the source configuration and enqueues it at the end of the i-queue at the opposing configuration.

The remaining reduction rules are standard from the π calculus reduction semantics. Rule [Eval] evaluates an expression inside an evaluation context. A reduction is not affected by the restriction of a shared (rule [Chan]) or a session name (rule [Sess]). From rule [Par] we ensure that a reduction on process is not affected if the process is composed in parallel with another process. Finally reduction is closed under structural congruence using rule [Struct]. We define $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$.

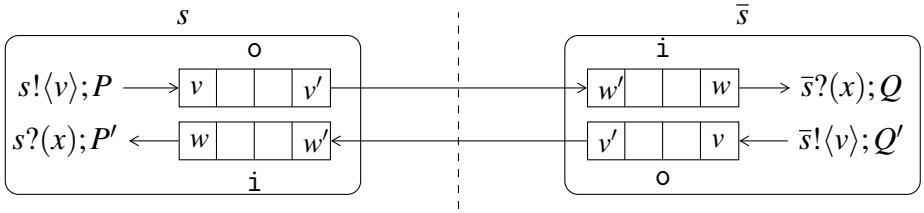

Figure 3.4: Schematic represantation of the ASP reduction semantics

Figure 3.4 shows a schematic representation of the reduction semantics for the ASP. We present each endpoint $s, \bar{s}$ with the corresponding linear processes and the input/output configurations. For example a value being sent from endpoint s to endpoint $\bar{s}$ is first sent by process $s!\langle v\rangle;P$ to the corresponding o-configuration of session s. The o-configuration follows a FIFO policy to interact with the i-configuration for the dual session $\bar{s}$. Again in the i-configuration a message follows a FIFO policy to finally interact with the receiving process $s?(x);Q$. The interaction to send a value from endpoint $\bar{s}$ to endpoint s follows a symmetric direction.

3.2 Types for Asynchronous Session Processes

This section presents a session typing discipline for ASP processes and establishes some key theoretical results: properties of subtyping (Proposition 3.2.1), subject reduction (Theorem 3.2.1), and communication safety (Theorem 4.2.2).

3.2.1 Type Syntax

The type syntax is an extension of the standard session types from [HVK98].

$$
\begin{array}{lrcl}
\text{(Shared)} & U & ::= & \texttt{bool} \mid \texttt{i}\langle S\rangle \mid \texttt{o}\langle S\rangle \\
\text{(Value)} & T & ::= & U \mid S \\
\text{(Session)} & S & ::= & !\langle T\rangle;S \mid ?(T);S \mid \oplus\{l_i : S_i\}_{i\in I} \mid \&\{l_i : S_i\}_{i\in I} \\
 & & & \mid \mu X.S \mid X \mid \texttt{end}
\end{array}
$$

The shared types U include Booleans $\texttt{bool}$, and the IO-types [PS96, HY07] $\texttt{i}\langle S\rangle$ (accept, i.e. input) and $\texttt{o}\langle S\rangle$ (request, i.e. output) for the shared channels via which sessions of type S are initiated. In the present work, IO-types (often called client/server types) are used to control locality (shared channel buffers are located only at the server side) and the associated typed transitions, playing a central role in our behavioural theory. In the session types S, the output type $!\langle T\rangle;S$ represents sending a value of type T, then continuing as S; dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i\in I}$ describes the selection of one of the labels l_i, then continuing as S_i. Branching type $\&\{l_i : S_i\}_{i\in I}$ waits with I options, behaving as type S_i if the label l_i is selected. End type $\texttt{end}$ represents session completion and is often omitted. For recursive types $\mu X.S$, we assume the type variables are guarded in the standard way. Process variable X is the standard recursive variable. We do not allow the occurence of a recursive variable as a session type object, i.e. being carried as a type on send or receive prefixes, cf. [BH13].

$$
\begin{array}{rcl}
\mathscr{F}(\mathscr{R}) & = & \{(\texttt{bool},\texttt{bool}),(\texttt{end},\texttt{end})\} \\
& \cup & \{(\texttt{i}\langle S\rangle,\texttt{i}\langle S'\rangle),(\texttt{o}\langle S\rangle,\texttt{o}\langle S'\rangle) \mid (S,S'),(S',S)\in\mathscr{R}\} \\
& \cup & \{(!\langle T_1\rangle;S_1,!\langle T_2\rangle;S_2) \mid (T_2,T_1),(S_1,S_2)\in\mathscr{R}\} \\
& \cup & \{(?(T_1);S_1,?(T_2);S_2) \mid (T_1,T_2),(S_1,S_2)\in\mathscr{R}\} \\
& \cup & \{(\oplus\{l_i:S_i\}_{i\in I},\oplus\{l_j:S'_j\}_{j\in J}) \mid I\subseteq J,\forall i\in I.(S_i,S'_i)\in\mathscr{R}\} \\
& \cup & \{(\&\{l_i:S_i\}_{i\in I},\&\{l_j:S'_j\}_{j\in J}) \mid J\subseteq I,\forall j\in J.(S_j,S'_j)\in\mathscr{R}\} \\
& \cup & \{(\mu X.S,S') \mid (S\{\mu X.S/X\},S')\in\mathscr{R}\} \\
& \cup & \{(S,\mu X.S') \mid (S,S'\{\mu X.S'/X\})\in\mathscr{R}\}
\end{array}
$$

Figure 3.5: The generating function for the session subtyping relation.

3.2.2 Session Subtyping

If P has a session channel s of type S, the ways in which P is prepared to use s are at *most* as S. For example, if S is $\&\{l_i:S_i\}_{i\in\{1,2\}}$, then P handles the cases for l_1 and l_2 but not any others; thus P can only interact with peers that select either one of these two labels. By this intuition, for a process Q with session type S' to be safely used in place of P (i.e. subsumption via $S'\leq S$), Q should be composable in the same or *more* ways (i.e. with more peers) than P, e.g. if S' is $\&\{l_i:S_i\}_{i\in\{1,2,3\}}$, then Q can interact with the same peers as P plus those that select l_3.

Formally, the subtyping relation is defined on the set of all closed and contractive types $\mathscr{T}$ as follows: for $T',T\in\mathscr{T}$, T' is a subtype of T, written $T'\leq T$, if (T',T) is in the largest fixed point of the monotone function:

$$\mathscr{F}:\mathscr{P}(\mathscr{T}\times\mathscr{T})\to\mathscr{P}(\mathscr{T}\times\mathscr{T})$$

given in Figure 3.5. Line 2 is standard: $\texttt{i}\langle S\rangle$ and $\texttt{o}\langle S\rangle$ are invariant on S since it supports both S and $\overline{S}$ (see duality below). Lines 7 and 8 give the standard rules for recursion. In Lines

3 and 4, the linear output (resp. input) is contravariant (resp. covariant) on the message type that follows [MY09]. In Line 5, a select that requires support for more labels means fewer peers can be safely composed; dually for branching in Line 6.

$$\begin{array}{llll} \overline{!\langle T\rangle;S} = ?(T);\overline{S} & & \overline{\&\{l_i:S_i\}_{i\in I}} = \oplus\{l_i:\overline{S_i}\}_{i\in I} & \\ \overline{?(T);S} = !\langle T\rangle;\overline{S} & & \overline{\oplus\{l_i:S_i\}_{i\in I}} = \&\{l_i:\overline{S_i}\}_{i\in I} & \\ \overline{X} = X & \overline{\mathtt{end}} = \mathtt{end} & \overline{\mu X.S} = \mu X.\overline{S} & \end{array}$$

Figure 3.6: Session type duality.

In Figure 3.6 we define the *duality* relation between session types. Duality follows the structure of a session type. Send and receive operations are dual, similarly for the select and branch operator. The dual of the inactive type and type variables is the identity.

We clarify the semantics of $\leq$ through *duality*.

Lemma 3.2.1. $S_1 \leq S_2$ iff $\overline{S_2} \leq \overline{S_1}$.

Proof. Let us call any relation witnessing $\leq$ (i.e. is a fixed point of the subtyping function), *a subtyping relation*. Because $\overline{\overline{S}} = S$, it suffices to show the relation $\{(\overline{S_2},\overline{S_1}) \mid S_1 \leq S_2\}$ is a subtyping relation, which is immediate by construction. □

Definition 3.2.1 (Composable Types)**.** We define the set of *composable types* of a session type S as:

$$\mathtt{comp}(S) = \{S' \mid S' \leq \overline{S}\},$$

That is, $\mathtt{comp}(S)$ is the set of types which can be dually composed with S (note S and $\overline{S}$ are composable, hence if S' is smaller than $\overline{S}$, S' should be more composable with S).

Subtyping can be completely characterised by composability.

Proposition 3.2.1 (Subtyping Properties)**.** (1) $\leq$ is a preorder; (2) $S_1 \leq S_2$ if and only if $\mathtt{comp}(S_2) \subseteq \mathtt{comp}(S_1)$.

Proof. (1) is standard, while (2) uses Lemma 3.2.1. For both, see Appendix A.1 for details. □

3.2.3 Type System for Programs

Typing judgements for programs (i.e. closed processes that do not contain run-time syntax, Section 3.1) and expressions have the form:

$$\Gamma \vdash P \triangleright \Delta \qquad \text{and} \qquad \Gamma, \Delta \vdash e : T$$

with

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma \cdot u : U \ \mid \ \Gamma \cdot X : \Delta \qquad \text{and} \qquad \Delta \ ::= \ \emptyset \ \mid \ \Delta \cdot k : S \ \mid \ \Delta \cdot a$$

The *shared environment* Γ is a mapping from variables and shared channels to constant types and shared channel types. Recursion variables are recorded to type recursive processes. The *linear environment* Δ (also called *session typing environment*) is a mapping from variables and session channels to session types. The linear environment is also used to record the shared channels . The program typing judgement is read as: program P is typed under shared environment Γ and uses channels according to linear environment Δ. The expression judgement, expression e has type T under environments Γ and Δ. We may omit Δ from the latter if it is clear from the context.

Figure 3.7 defines the typing rules for programs. The system is similar to [HKP$^+$10, B$^+$08]. Rule (SChan) types shared channels in accordance with the environment Γ. A shared input type can be used as a shared output type by rule (SChan)$'$. Rules (Bool) and (Match) assign the boolean type to boolean constants `tt`, `ff` and value matching expressions (similarly for other boolean expressions, e.g. e `and` e). Rule (Name) extracts the type of a name expression.

Rules (Req) and (Acc) check if the type of the carried session name agrees with the shared

$$\Gamma \cdot u : U \vdash u : U \quad (\text{SChan}) \qquad \Gamma \cdot u : \mathtt{i}\langle S \rangle \vdash u : \mathtt{o}\langle S \rangle \quad (\text{SChan})'$$

$$\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathtt{bool} \quad (\text{Bool})$$

$$\frac{\Gamma \vdash n : T \vee \Delta = \Delta' \cdot n : T}{\Gamma, \Delta \vdash n : T} \quad (\text{Name}) \qquad \frac{\Gamma, \Delta \vdash e_i : T_i \quad i \in \{1,2\}}{\Gamma, \Delta \vdash e_1 = e_2 : \mathtt{bool}} \quad (\text{Match})$$

$$\frac{\Gamma \vdash a : \mathtt{o}\langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : \overline{S}}{\Gamma \vdash \overline{a}(x).P \triangleright \Delta} \quad (\text{Req}) \qquad \frac{\Gamma \vdash a : \mathtt{i}\langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : S}{\Gamma \vdash a(x).P \triangleright \Delta} \quad (\text{Acc})$$

$$\frac{\Gamma \vdash v : U \quad U \neq \mathtt{i}\langle S' \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k!\langle v \rangle; P \triangleright \Delta \cdot k : !\langle U \rangle; S} \quad (\text{Send}) \qquad \frac{\Gamma \cdot x : U \vdash P \triangleright \Delta \cdot k : S \quad U \neq \mathtt{i}\langle S' \rangle}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(U); S} \quad (\text{Recv})$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k!\langle k' \rangle; P \triangleright \Delta \cdot k : !\langle S' \rangle; S \cdot k' : S'} \quad (\text{Deleg}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k : S \cdot x : S'}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(S'); S} \quad (\text{SRecv})$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S}{\Gamma \vdash k \oplus l; P \triangleright \Delta \cdot k : \oplus\{l : S\}} \quad (\text{Sel}) \qquad \frac{\forall i \in I \quad \Gamma \vdash P_i \triangleright \Delta \cdot k : S_i}{\Gamma \vdash k\&\{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot k : \&\{l_i : S_i\}_{i \in I}} \quad (\text{Bra})$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta_i \quad i \in \{1,2\} \quad \mathtt{dom}(\Delta_1) \cap \mathtt{dom}(\Delta_2) = \emptyset}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cdot \Delta_2} \quad (\text{Conc}) \qquad \frac{\Gamma, \Delta \vdash e : \mathtt{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \quad (\text{If})$$

$$\frac{\Gamma \cdot a : U \vdash P \triangleright \Delta \cdot a}{\Gamma \vdash (\nu\, a)P \triangleright \Delta} \quad (\text{CRes}) \qquad \frac{\Delta\ \mathtt{end\ only}}{\Gamma \vdash a[\varepsilon] \triangleright \Delta \cdot a} \quad (\text{EBuff})$$

$$\frac{\Gamma \cdot X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta} \quad (\text{Rec}) \qquad \Gamma \cdot X : \Delta \vdash X \triangleright \Delta \quad (\text{Var})$$

$$\frac{\Delta\ \mathtt{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \quad (\text{Inact}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright \Delta'} (\text{Subs})$$

Figure 3.7: Typing rules for programs.

environment mapping of the shared name. Furthermore the shared environment should map the shared channel to the output (resp. input) shared channel type. Rules (Send) and (Recv) require that a value being sent (resp. received) on a session channel is typed with the send session type prefix (resp. receive session type prefix) according to the shared environment Γ. Rule (Deleg) types the delegation operation of session channels. It requires the session channel being sent, to be present with the correct type in the linear environment of the conclusion judgement. Rule (SRecv) types the receiving of a delegated session channel. The delegated channel should be present in the linear environment Δ of the hypothesis judgement. Rules (Sel) and (Bra) type the selection and branching actions respectively. The selection typing judgement creates a selection session type on the label of the selection operation. Dually the branch typing judgement types a branch operation on the labels offered by the branch operator. Note that appart from the session name being typed, the linear environment should be the same in all branching processes.

The subsumption rule (Subs) is used to identify session types up-to the subtyping relation. Rule (Conc) concatenates the disjoint environment typing of parallel processes. Rule (If) checks for a boolean condition and for equality of the session types in both branches. Rule (EBuff) types records the shared name of the empty shared configuration in the linear environment. Rule (CRes) restricts a shared channel and its buffer by removing its typing from both the shared linear environments. Rule (Rec) checks a process if it has the same linear environment with the mapping of the active process variable and (Var) maps a process variable to a linear environment through the shared environment Γ. The empty process is typed with a *complete* (all channels are typed with `end`) linear environment in rule (Inact).

3.2.4 Type System for Run-time Syntax

This section extends the type system for programs (Section 3.2.3) to the full type system for run-time syntax. Our new system significantly simplifies that in [HKP+10] by adapting the

approach developed in [B+08]. First we define an additional type category T, which includes session types and *message types*:

$$\begin{array}{llll} \text{(General)} & \mathsf{T} ::= S \mid M & \text{(IMsg)} & M_{\mathtt{i}} ::= \emptyset \mid ?(T);M_{\mathtt{i}} \mid \&l;M_{\mathtt{i}} \\ \text{(Message)} & M ::= M_{\mathtt{i}} \mid M_{\mathtt{o}} & \text{(OMsg)} & M_{\mathtt{o}} ::= \emptyset \mid !\langle T\rangle;M_{\mathtt{o}} \mid \oplus l;M_{\mathtt{o}} \end{array}$$

Message types provide a type abstraction for the values stored in queues, and are used for typing endpoint configurations. A message type M is either an input $M_{\mathtt{i}}$ or an output $M_{\mathtt{o}}$ queue abstraction. Incoming messages and branch labels enqueued in an $\mathtt{i}$-queue are recorded as $?(T)$ and $\&l$ respectively. Similarly, $!\langle T\rangle$ and $\oplus l$ for outgoing messages and select labels in an $\mathtt{o}$-queue. $\emptyset$ is used to type empty queues. We then extend the linear environment Δ to include the session channels for which a configuration is found to be present as T: a configuration by itself is typed as M, where M are the enqueued message types, and the composition of a session process and its associated configuration as S. Linear environment Δ also records the presence of a session endpoint configuration s. The linear environment Δ is now given by the extended grammar:

$$\Delta ::= \emptyset \mid \Delta \cdot k : S \mid \Delta \cdot a \mid \Delta \cdot s : \mathsf{T} \mid \Delta \cdot s$$

The $*$ operator is used to type the parallel composition of run-time processes.

Definition 3.2.2 (Message Type Concatenation)**.**

$$\begin{array}{rcll} S * \emptyset & = & S & \\ S * !\langle T\rangle;M_{\mathtt{o}} & = & !\langle T\rangle;S * M_{\mathtt{o}} & \\ ?(T);S * ?(T);M_{\mathtt{i}} & = & S * M_{\mathtt{i}} & \\ S_k * \oplus l_k;M_{\mathtt{o}} & = & \oplus\{l_i : S_k * M_{\mathtt{o}}\} & (k \in I) \\ \&\{l_i : S_i\}_{i\in I} * \&l_k;M_{\mathtt{i}} & = & S_k * M_{\mathtt{i}} & (k \in I) \end{array}$$

$$\Delta_1 * \Delta_2 \quad = \quad \Delta_1 \backslash \mathtt{dom}(\Delta_2) \cup \Delta_2 \backslash \ \mathtt{dom}(\Delta_1) \cup \{s : S * M \mid s : S \in \Delta_i,\ s : M \in \Delta_j \text{ where } i, j \in \{1,2\}, i \neq j\}$$

The $*$ operator is used to reconstitute an overall session type for a session endpoint *locality*, which consists of the type of a session channel $s : S$ and the corresponding session message type $s : M$. A session s enqueues messages in the the $\mathtt{o}$-queue, before they are delivered to the opposing locality (i.e the opposing $\mathtt{i}$-queue). We consider that in this case the messages did not change their locality and still remain in the typing scope of the s channel, so we choose to define $*$ as a form of concatenation between type $M_{\mathtt{o}}$ of the $\mathtt{o}$-queue of session s and session type S of session s. On the other hand, messages that exist in the $\mathtt{i}$-queue of session s are already delivered from the opposing locality. In this case we considered $\mathtt{i}$-messages as already received by the locality and we express it in the definition of $*$ by consuming the message type $M_{\mathtt{i}}$ for session s out of the session type S of session s. We clarify the above intuitions for the $*$ operator with its description:

In the cases where we concatenate a session type S with an output message type $!\langle T\rangle;M_{\mathtt{o}}$ or $\oplus l_k;M_{\mathtt{o}}$, we concatenate the message type prefix with the session type S to get $!\langle T\rangle;S$ and $\oplus\{l_i : S_i\}_{i\in I}, k \in I$ and we continue with the concatenation inductively. Remember that an output message type $M_{\mathtt{o}}$ types an $\mathtt{o}$-configuration in the reverse order. The inductive definition on $*$ is used to reverse the concatenation order once more, resulting in a consistent session type sequence. Note that for the case of the select message type the result is non-deterministic. This follows the intuition that session types can be subsumed up-to subtyping, for example we can have:

$$[l_1] \oplus * S_1 \quad = \quad \oplus\{l_1 : S_1\}$$
$$[l_1] \oplus * S_2 \quad = \quad \oplus\{l_1 : S_1, l_2 : S_2\}$$

with the resulting type to be consistent up-to subtyping since $\oplus\{l_1 : S_1\}$ is a subtype of $\oplus\{l_1 : S_1, l_2 : S_2\}$. For the case of input message types we expect both the session type S and the

message type $M_{\mathtt{i}}$ to have matching prefixes. The $*$ operator then consumes the prefixes and proceeds inductively. Lastly, we use the $*$ operator to compose the extended linear environments. The two linear environments that are being composed should have disjoint domains. In the case where they have a common domain on a session name s, we expect that one linear environment will record a session type S for s and the other will record a message type M for s. In any other case the $*$ operator is undefined.

As an example consider linear environments:

$$\begin{aligned}\Delta_1 &= \{s_1 : S_1 \cdot s_2 :?(T_2);S_2\}\\ \Delta_2 &= \{s_1 :!\langle T_1\rangle \cdot s_2 :?(T_2)\}\end{aligned}$$

Then we can compose Δ_1 and Δ_2 to get:

$$\Delta_1 * \Delta_2 = \{s_1 :!\langle T\rangle;S_1 \cdot s_2 : S_2\}$$

Consider now a linear environment:

$$\Delta_3 = \{s_1 : S_1'\}$$

then the operation $\Delta_1 * \Delta_3$ is undefined because $s_1 : S_1, s_1 : S_1' \in \Delta_1 \cap \Delta_2$. Also note that $\Delta_2 * \Delta_3$ is defined:

$$\Delta_2 * \Delta_3 = \{s_1 :!\langle T_1\rangle;S_1' \cdot s_2 :?(T_2)\}$$

We present the typing rules for the run-time type system. Most of the typing rules are directly inherited from the program type system (Figure 3.7). Figure 3.8 lists the rules for run-time syntax. A session configuration for s is typed with the message type M. Rules (InQ) and (OutQ) respectively type the empty `i`- and `o`-queues with the empty message type and record the presence of the session queue in Δ. Rule (RcvQ) takes the typing of `i`-queue tail and pre-

$$\Gamma \vdash s[\mathtt{o} : \varepsilon] \triangleright s : \emptyset \cdot s \quad (\text{OutQ}) \qquad \Gamma \vdash s[\mathtt{i} : \varepsilon] \triangleright s : \emptyset \cdot s \quad (\text{InQ})$$

$$\frac{\Gamma \vdash s[\mathtt{o} : \vec{h}] \triangleright s : M_{\mathtt{o}} \quad \Gamma \vdash v : T}{\Gamma \vdash s[\mathtt{o} : v \cdot \vec{h}] \triangleright s : !\langle T \rangle; M_{\mathtt{o}}} \ (\text{SndQ}) \qquad \frac{\Gamma \vdash s[\mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}} \quad \Gamma \vdash v : T}{\Gamma \vdash s[\mathtt{i} : v \cdot \vec{h}] \triangleright s : ?(T); M_{\mathtt{i}}} \ (\text{RcvQ})$$

$$\frac{\Gamma \vdash s[\mathtt{o} : \vec{h}] \triangleright s : M_{\mathtt{o}}}{\Gamma \vdash s[\mathtt{o} : l \cdot \vec{h}] \triangleright s : \oplus l; M_{\mathtt{o}}} \ (\text{SelQ}) \qquad \frac{\Gamma \vdash s[\mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}}}{\Gamma \vdash s[\mathtt{i} : l \cdot \vec{h}] \triangleright s : \& l; M_{\mathtt{i}}} \ (\text{BraQ})$$

$$\frac{\Gamma \vdash s[\mathtt{o} : \vec{h}] \triangleright s' : S' \cdot s : M_{\mathtt{o}}}{\Gamma \vdash s[\mathtt{o} : \vec{h} \cdot s'] \triangleright s : !\langle S' \rangle; M_{\mathtt{o}}} \ (\text{DelQ}) \qquad \frac{\Gamma \vdash s[\mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}}}{\Gamma \vdash s[\mathtt{i} : s' \cdot \vec{h}] \triangleright s : ?(S'); M_{\mathtt{i}} \cdot s' : S'} \ (\text{SRcvQ})$$

$$\frac{\Gamma \vdash P \triangleright \Delta_1 \qquad \Gamma \vdash Q \triangleright \Delta_2}{\Gamma \vdash P \mid Q \triangleright \Delta_1 \ * \ \Delta_2} \ (\text{QConc}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot s : S \cdot \overline{s} : \overline{S} \cdot s \cdot \overline{s}}{\Gamma \vdash (\nu\, s)P \triangleright \Delta} \ (\text{SRes})$$

$$\frac{\Gamma \vdash a[\vec{h}] \triangleright \Delta}{\Gamma \vdash a[\vec{h} \cdot s] \triangleright \Delta \cdot s : \emptyset} \ (\text{Buff}) \qquad \Gamma \vdash \overline{a}\langle s \rangle \triangleright s : \emptyset \quad (\text{ReqM})$$

Figure 3.8: Extended typing rules for the ASP run-time processes.

fixes the message type for the head element. Rule (BraQ) is similar, but handles the branching by prefixing the label message type. Rules (OutQ) and (SelQ) type o-queues. Note that rules (OutQ) and (SelQ) construct the type in the reverse direction of the o-endpoint ordering. Rules (InDelQ) and (DelQ) deal with the typing of the delegated session, but are otherwise similar to (RcvQ) and (SndQ). As regards the parallel composition, rule [Conc] is replaced by rule (QConc), which uses the $*$ for combining the types of the parallel components. Rule (SRes) types the session restriction by asserting that the session endpoints have dual typing and that the corresponding session queues are present. Rule (Buff) is used (in conjunction with (EBuff) from Figure 3.7) to type non-empty shared channel endpoints by recording the enqueued session channels. Finally, (ReqM) types asynchronous session request messages with the recording of the message type for session channel s in the linear environment.

3.2.5 Subject Reduction

In this section we prove the main properties of the ASP session type system that are summarised in the subject reduction and process safety theorems.

Definition 3.2.3 (Well-configured Linear Environments)**.** We say that Δ is *well configured* if $\forall s \in \mathtt{dom}(\Delta)$, then $\Delta(s) = S$ with $\Delta(\overline{s}) = \overline{S}$

We say that a linear environment is well-configured if duality relates the types of dual session channels.

Definition 3.2.4 (Linear Environment Reduction)**.** We define:

1. $\{s : !\langle T\rangle; S \cdot \overline{s} : ?(T); S'\} \longrightarrow \{s : S \cdot \overline{s} : S'\}$
2. $\{s : \oplus\{l_i : S_i\}_{i\in I} \cdot \overline{s} : \&\{l_i : S'_i\}_{i\in I}\} \longrightarrow \{s : S_k \cdot \overline{s} : S'_k\}$ $(k \in I)$
3. $\Delta \cup \Delta'' \longrightarrow \Delta' \cup \Delta''$ if $\Delta \longrightarrow \Delta'$.

Reduction over linear environments, require the interaction between dual endpoints with dual types.

The next three lemmas are used to prove the subject reduction theorem. We prove the weakening and strengthening cases for the type environments and the standard substitution lemma.

Lemma 3.2.2 (Weakening Lemma)**.**

Let $\Gamma \vdash P \triangleright \Delta$.

1. If $X \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$.
2. If $u \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot u : U \vdash P \triangleright \Delta$.
3. If $k \notin \mathtt{dom}(\Delta)$ then $\Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}$.

Proof. The proof is done by induction on the structure of ASP process. See Appendix A.2.1 for details. □

Lemma 3.2.3 (Strengthening Lemma)**.**

1. If $X \notin \mathtt{fpv}(P)$, then $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright \Delta$.
2. If $u \notin \mathtt{fn}(P) \cup \mathtt{fv}(P)$, then $\Gamma \cdot u : U \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright \Delta$.
3. If $k \notin \mathtt{fn}(P) \cup \mathtt{fv}(P)$ then $\Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}$ implies $\Gamma \vdash P \triangleright \Delta$.

Proof. The proof is done by induction on the structure of ASP process. See Appendix A.2.1 for details. □

Lemma 3.2.4 (Substitution Lemma)**.**

1. If $\Gamma \cdot x : U, \Delta \vdash e : U'$ and $\Gamma \vdash v \triangleright U$, then $\Gamma, \Delta \vdash e\{v/x\} : U'$.
2. If $\Gamma, \Delta \cdot x : T \vdash e : U$ and s fresh, then $\Gamma, \Delta \cdot s : S \vdash e\{s/x\} : U$.
3. If $\Gamma \cdot x : U \vdash P \triangleright \Delta$ and $\Gamma \vdash v \triangleright U$, then $\Gamma \vdash P\{v/x\} \triangleright \Delta$.
4. If $\Gamma \vdash P \triangleright \Delta \cdot k : T$, then $\Gamma \vdash P\{s/k\} \triangleright \Delta \cdot s : T$.

Proof. The proof is done by induction on the structure of expressions for Parts (i) and (ii) and by induction on the structure of ASP process. See Appendix A.2.1 for details. □

The next theorem states the soundness property of the ASP typing system:

Theorem 3.2.1 (Subject Congruence and Reduction)**.**

1. If $\Gamma \vdash P \triangleright \Delta$ and $P \equiv Q$, then $\Gamma \vdash Q \triangleright \Delta$.
2. If $\Gamma \vdash P \triangleright \Delta$ with Δ well-configured and $P \longrightarrow Q$, then we have $\Gamma \vdash Q \triangleright \Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and Δ' is well-configured.

Proof. The proof for Part (i) is done with a cases analysis on the structural congruence rules. The proof for Part (ii) is done by induction on the reduction relation. For details, see Appendix A.2.2. □

We now prove communication safety.

Definition 3.2.5 (s-redex)**.** We say an *s-redex* is a parallel composition of two s-processes that has one of the following shapes:

(a) $s!\langle v\rangle;P \mid s[\mathtt{o} : \vec{h}]$ (b) $s \oplus l_i;P \mid s[\mathtt{o} : \vec{h}]$

(c) $s?(x);P \mid s[\mathtt{i} : v \cdot \vec{h}]$ (d) $s\&\{l_i : P_i\}_{i\in I} \mid s[\mathtt{i} : l_{i'} \cdot \vec{h}]$

(e) $s[\mathtt{o} : v \cdot \vec{h}] \mid \bar{s}[\mathtt{i} : \vec{h}']$

All redexes require the immediate action to correspond with the active type prefix in the local configuration.

A process P is an *error* if up-to structural congruence (following [HYC08, § 5]), P contains two s-processes which do not form an s-redex, or an expression in P contains a type error in the standard sense. As a corollary of subject reduction (Theorem 3.2.1), we obtain:

Theorem 3.2.2 (Communication and Event-Handling Safety)**.** If P is a well-typed program, then $\Gamma \vdash P \triangleright \emptyset$, and P never reduces to an error.

Proof. The proof is a direct consequence of the subject reduction theorem (Theorem 3.2.1). If we assume that a well-typed process can result in an error process this leads to contradiction using Theorem 3.2.1, because error processes are not typable. See Appendix A.2.3 for details. □

3.3 Asynchronous Session Bisimulation and its Properties

This section presents the behavioural theory for the ASP. We define the observational theory using an untyped labelled transition system on the ASP processes and a labelled transition system on the typing environment. Both labelled transition systems are combined to define a typed labelled transition system for the ASP processes. We use the typed LTS to define the asynchronous session typed bisimulation, where typed bisimilarity is the maximal typed reduction-based congruence that preserves observations. We use the bisimulation theory to study the confluence and determinacy properties of the session types. The results of this section are directly used to study properties of event-driven programming in Chapter 4 and Chapter 5.

3.3.1 Labelled Transition Semantics

We define a set of labels $(\ell, \ell', ...)$ together with their standard relations:

Definition 3.3.1 (Action Labels)**.** Let ℓ range over:

$$\ell \ ::= \ a\langle s\rangle \ \mid \ \overline{a}\langle s\rangle \ \mid \ \overline{a}(s) \ \mid \ s?\langle v\rangle \ \mid \ s!\langle v\rangle \ \mid \ s!(a) \ \mid \ s\&l \ \mid \ s\oplus l \ \mid \ \tau$$

The first three labels denote the session accept, session request and bound session request respectively. The next labels define the session actions for input, output, bound output, branching, selection respectively. The last action is the standard silent τ-action. We write $\mathtt{subj}(\ell)$ (resp. $\mathtt{obj}(\ell)$) to denote the set of free subjects (resp. object) in ℓ; and $\mathtt{fn}(\ell)$ (resp. $\mathtt{bn}(\ell)$) to denote the set of free (resp. bound) names in ℓ. Moreover $\mathtt{n}(\ell)$ defines the union $\mathtt{fn}(\ell)\cup\mathtt{bn}(\ell)$:

Actions(ℓ)	$\texttt{subj}(\ell)$	$\texttt{obj}(\ell)$	$\texttt{fn}(\ell)$	$\texttt{bn}(\ell)$
$a\langle s\rangle, \overline{a}\langle s\rangle$	$\{a\}$	$\{s\}$	$\{a,s\}$	$\emptyset$
$\overline{a}(s)$	$\{a\}$	$\{s\}$	$\{a\}$	$\{s\}$
$s!\langle v\rangle, s?\langle v\rangle$	$\{s\}$	$\{v\}$	$\{s,v\}$	$\emptyset$
$s!(a)$	$\{s\}$	$\{a\}$	$\{s\}$	$\{a\}$
$s \oplus l, s\&l$	$\{s\}$	$\emptyset$	$\{s\}$	$\emptyset$

Definition 3.3.2 (Context)**.** *A context is defined as:*

$$C ::= - \mid C \mid P \mid P \mid C \mid (\nu\, n)C \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C' \mid \mu X.C$$
$$\mid s!\langle v\rangle;C \mid s?(x);C \mid s \oplus l;C \mid s\&\{l_i : C_i\}_{i\in I} \mid \overline{a}(x).C \mid a(x).C$$

Expression $C[P]$ substitutes process P in each hole $(-)$ of the context C definition.

We define the symmetric operator $\ell \asymp \ell'$ on labels. The expression $\ell \asymp \ell'$ denotes that ℓ is a dual of ℓ' and it is defined as:

Definition 3.3.3 (Label Duality)**.**

$$a\langle s\rangle \asymp \overline{a}\langle s\rangle \quad a\langle s\rangle \asymp \overline{a}(s) \quad s?\langle v\rangle \asymp \overline{s}!\langle v\rangle \quad s?\langle a\rangle \asymp \overline{s}!(a) \quad s\&l \asymp \overline{s} \oplus l$$

Untyped Labelled Transition System. Figure 3.9 defines the untyped label transition system (LTS). Rules $\langle$Acc$\rangle$/$\langle$Req$\rangle$ are used to define the session initialisation. The accept label is observed on shared endpoints and the request label is observed on the asynchronous request processes. The next four rules $\langle$In$\rangle$/$\langle$Out$\rangle$/$\langle$Bra$\rangle$/$\langle$Sel$\rangle$ impose that an action is observable when a message moves from its local endpoint to its remote (i.e. opposing) endpoint. Note that all the the visible actions (with the exception of $\overline{a}\langle s\rangle$) are observed on (shared and session) endpoints. When the process accesses its local endpoint, the action is *invisible* from the outside, as formalised by $\langle$Local$\rangle$. The rest of the compositional rules are standard. Rule

$$\langle \text{Acc} \rangle \quad a[\vec{s}] \xrightarrow{a\langle s \rangle} a[\vec{s} \cdot s] \qquad \langle \text{Req} \rangle \quad \overline{a}\langle s \rangle \xrightarrow{\overline{a}\langle s \rangle} \mathbf{0}$$

$$\langle \text{In} \rangle \quad s[\mathtt{i} : \vec{h}] \xrightarrow{s?\langle v \rangle} s[\mathtt{i} : \vec{h} \cdot v] \qquad \langle \text{Out} \rangle \quad s[\mathtt{o} : v \cdot \vec{h}] \xrightarrow{s!\langle v \rangle} s[\mathtt{o} : \vec{h}]$$

$$\langle \text{Bra} \rangle \quad s[\mathtt{i} : \vec{h}] \xrightarrow{s\&l} s[\mathtt{i} : \vec{h} \cdot l] \qquad \langle \text{Sel} \rangle \quad s[\mathtt{o} : l \cdot \vec{h}] \xrightarrow{s \oplus l} s[\mathtt{o} : h]$$

$$\langle \text{Local} \rangle \quad \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \qquad \langle \text{Tau} \rangle \quad \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\, \mathtt{bn}(\ell, \ell'))(P' \mid Q')}$$

$$\langle \text{Par}_L \rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \langle \text{Par}_R \rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{Q \mid P \xrightarrow{\ell} Q \mid P'}$$

$$\langle \text{Res} \rangle \quad \frac{P \xrightarrow{\ell} P' \; n \notin \mathtt{fn}(\ell)}{(\nu\, n)P \xrightarrow{\ell} (\nu\, n)P'} \qquad \langle \text{OpenS} \rangle \quad \frac{P \xrightarrow{\overline{a}\langle s \rangle} P'}{(\nu\, s)P \xrightarrow{\overline{a}(s)} P'}$$

$$\langle \text{OpenN} \rangle \quad \frac{P \xrightarrow{s!\langle a \rangle} P'}{(\nu\, a)P \xrightarrow{s!(a)} P'} \qquad \langle \text{Alpha} \rangle \quad \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Figure 3.9: Labelled transition system.

$\langle$Tau$\rangle$ observes a τ transition on a parallel composition, when the parallel components exhibit dual actions. We use rules $\langle \text{Par} \rangle_L$ and $\langle \text{Par} \rangle_R$ to define that if a process can do a transition ℓ then it can do the same transition when it is composed in parallel with a process Q, provided that the bound names of ℓ are disjoint with the free names of Q. Rule $\langle$Res$\rangle$ preserves observability of an action ℓ in a process under the restriction operator, provided that the subject of ℓ remains free. Scope opening for action objects is desrcibed using rules $\langle$OpenS$\rangle$ for session actions and $\langle$OpenN$\rangle$ for shared name actions. Rule $\langle$Alpha$\rangle$ closes the transition relation under alpha-conversion.

Localisation and Typed Labelled Transition System. We introduce the notion of localisation for the ASP processes. A localised process is a typed process that is also a parallel composition, which composes all session configurations for each session name used.

Definition 3.3.4 (Localisation)**.** Let P be closed and $\Gamma \vdash P \triangleright \Delta$. Then we say $\Gamma \vdash P \triangleright \Delta$ is *localised* if:

(1) For each $s \in \texttt{dom}(\Delta)$, $s : S \cdot s \in \Delta$ and (2) If $\Gamma(a) = \texttt{i}\langle S \rangle$, then $a \in \Delta$.

A localised process owns all necessary queues as specified in its typing environment. Formally we say that for each free session name in Δ the corresponding endpoint configuration exists composed in parallel within the process. Restricted session names are implicitly checked for localisation following the fact that a localised process is typed. A typed process implies that rule [SRes] was used to check the presence of the session configuration on bound session names. We further say *P is localised* if it is so for a suitable pair of environments.

Example 3.3.1 (Simple Localisation Example)**.**

- Process $\Gamma \vdash s?(x); s!\langle x+1 \rangle; \mathbf{0} \triangleright s :?(U); !\langle U \rangle; \texttt{end}$ is not localised, since $s \in \texttt{dom}(\Delta)$ and $s \notin \Delta$.
- On the other hand, process $\Gamma \vdash s?(x); s!\langle x+1 \rangle; \mathbf{0} \mid s[\texttt{i} : \vec{h}_1, \texttt{o} : \vec{h}_2] \triangleright s :?(U); !\langle U \rangle; \texttt{end} \cdot s$ is localised, since $s \in \texttt{dom}(\Delta)$ and $s \in \Delta$.

- Similarly, process $\Gamma \vdash a(x).P \triangleright \emptyset$ is not localised, but $\Gamma \vdash a(x).P \mid a[\vec{s}] \triangleright a$ is.

By composing buffers at the appropriate channels, any typable closed process can become localised.

Proposition 3.3.1. If $\Gamma \vdash P_1 \triangleright \Delta_1$ is localised, $P_1 \longrightarrow P_2$ and $\Gamma \vdash P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_2 \triangleright \Delta_2$ is localised.

Proof. The proof uses a simple induction on the structure of the $\longrightarrow$ relation and concludes because the reduction relation preserves endpoint configurations in a process. □

We proceed with the definition of the typed LTS for typing environments on the basis of the untyped one. The basic idea is *to use the type information to control the enabling of actions* (cf. [HR04]). This is realised by introducing the definition of the *environment transition*, defined in Figure 3.10. A transition $(\Gamma, \Delta) \xrightarrow{\ell} (\Gamma', \Delta')$ means that an environment (Γ, Δ) allows an action ℓ to take place, and the resulting environment is (Γ', Δ'), constraining process transitions through the linear and shared environments. This constraint is at the heart of our typed LTS, accurately capturing interactions in the presence of sessions and local buffers.

The first rule in Figure 3.10 says that the reception of a message via a is possible only when a is input-typed (`i`-mode) and its endpoint is present ($a \in \Delta$). The second is dual, saying that an output at a is possible only when a has `o`-mode and no shared endpoint exists in the linear environment. The case is similar for a bound output action $\overline{a}(s)$. The definition for the session actions focuses on the precondition $\overline{s} \notin \Delta$, that enforces that the opposing endpoint is not present in the process for an action to be observed. This is a basic precondition, since it ensures the linearity property for session type (i.e. if we drop this precondition a session endpoint can interact with its dual endpoint and with the environment at the same time, breaking the session linearity). The two session output rules ($\ell = s!\langle v \rangle$ and $s!(a)$) are the standard value output and scope opening rule. Output actions happen on an output prefixed

$$\Gamma(a) = \mathtt{i}\langle S\rangle, a \in \Delta, s \text{ fresh} \quad \text{implies} \quad (\Gamma,\Delta) \xrightarrow{a\langle s\rangle} (\Gamma,\Delta\cdot s:\overline{S})$$

$$\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta \quad \text{implies} \quad (\Gamma,\Delta) \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta)$$

$$\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta, s \text{ fresh} \quad \text{implies} \quad (\Gamma,\Delta) \xrightarrow{\overline{a}(s)} (\Gamma,\Delta\cdot s:S)$$

$$\Gamma \vdash v : U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma,\Delta\cdot s:!\langle U\rangle;S) \xrightarrow{s!\langle v\rangle} (\Gamma,\Delta\cdot s:S)$$

$$\overline{s} \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma,\Delta\cdot s:!\langle \mathtt{o}\langle S'\rangle\rangle;S) \xrightarrow{s!(a)} (\Gamma\cdot a:\mathtt{o}\langle S'\rangle,\Delta\cdot s:S)$$

$$\Gamma \vdash v : U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma,\Delta\cdot s:?(U);S) \xrightarrow{s?\langle v\rangle} (\Gamma,\Delta\cdot s:S)$$

$$\overline{s} \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma,\Delta\cdot s:\oplus\{l_i:S_i\}_{i\in I}) \xrightarrow{s\oplus l_k} (\Gamma,\Delta\cdot s:S_k)$$

$$\overline{s} \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma,\Delta\cdot s:\&\{l_i:S_i\}_{i\in I}) \xrightarrow{s\&l_k} (\Gamma,\Delta\cdot s:S_k)$$

$$\Delta \longrightarrow \Delta' \vee \Delta = \Delta' \quad \text{implies} \quad (\Gamma,\Delta) \xrightarrow{\tau} (\Gamma,\Delta')$$

Figure 3.10: Labelled transition rules for environments.

session type and should agree with the shared environment Γ. The next rule is for value input and it is dual to the output rule. Note that in the case where the send/receive subject is a shared channel, it should be on o-mode. This is because a new accept should not be created without its endpoint in the same location. Label input and output are defined on the select and branch prefixed session types. The final rule ($\ell = \tau$) follows the reduction rules defined in § 3.2.4. We can also observe a τ action on every environment (Γ,Δ) without changing its state. The labelled transition system for environments omits the delegation case. This is justified by the fact that session input action $s?\langle s'\rangle$ may result in a non-localised process (i.e. it breaks the localisation requirement for a process).

The next definition defines a typed labelled transition system for processes.

Definition 3.3.5 (Typed Transition)**.** *Typed transition* relation is defined as:

$$\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$$

if (1) $P_1 \xrightarrow{\ell} P_2$ and (2) $(\Gamma_1, \Delta_1) \xrightarrow{\ell} (\Gamma_2, \Delta_2)$ with $\Gamma_i \vdash P_i \triangleright \Delta_i$.

We use both the untyped labelled transition system and the labelled transition system for environments to define a LTS for processes. The typed transition relation is used to study the bisimulation theory for session typed processes.

We use the notation $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, $\stackrel{\ell}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\ell}\Longrightarrow$ and $\stackrel{\widehat{\ell}}{\Longrightarrow}$ for $\Longrightarrow$ if $\ell = \tau$ and $\stackrel{\ell}{\Longrightarrow}$ otherwise. Furthermore we write $\xrightarrow{\widehat{\ell}}$ for $\longrightarrow$ if $\ell = \tau$ and $\xrightarrow{\ell}$ otherwise.

3.3.2 Bisimulation

Before we define any behavioural relation, we define the notion of a typed relation. Write $\rightleftharpoons$ for a symmetric and transitive closure of $\longrightarrow$ over linear environments.

Definition 3.3.6 (Typed Relation)**.** We say a binary relation $\mathscr{R}$ over closed, typed processes is a *typed relation* if, whenever it relates two typed processes, $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ we have $\Delta_1 \rightleftharpoons \Delta_2$.

We often leave the environments implicit, writing simply $P_1 \mathscr{R} P_2$.

We introduce the notion of typed barbs, for the observations of actions over typed processes.

Definition 3.3.7 (Typed Barbs)**.** We write

1. $\Gamma \vdash P \triangleright \Delta \downarrow a$ if $P \equiv (\nu\ \vec{n})(\overline{a}\langle s\rangle \mid R)$ with $a \notin \vec{n}$.
2. $\Gamma \vdash P \triangleright \Delta \downarrow s$ if $P \equiv (\nu\ \vec{n})(s[\mathtt{o} : h \cdot \vec{h}] \mid R)$ with $s \notin \vec{n}$ and $\overline{s} \notin \mathtt{dom}(\Delta)$.

We write $\Gamma \vdash P \triangleright \Delta \Downarrow n$ if $\exists P'.P \twoheadrightarrow P'$ and $\Gamma \vdash P' \triangleright \Delta' \downarrow n$.

We can now introduce the reduction congruence and the asynchronous bisimilarity.

Definition 3.3.8 (Reduction Congruence)**.** A typed relation $\mathscr{R}$ is *reduction congruence* if it is a congruence and satisfies the following condition: for each $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, whenever $\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2$ are localised then:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow n$ iff $\Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow n$.
2. Whenever
   - $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ holds, $P_1 \twoheadrightarrow P_1'$ implies $P_2 \twoheadrightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \rightleftharpoons \Delta_2'$.
   - The symmetric case.

The maximum reduction congruence ([HY95]), is denoted by $\cong$.

Definition 3.3.9 (Asynchronous Session Bisimulation)**.** A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, it holds:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\widehat{\ell}}{\Longrightarrow} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ with $\Delta_1' \rightleftharpoons \Delta_2'$ holds and
2. the symmetric case.

The maximum bisimulation exists which we call *bisimilarity*, denoted by $\approx$. We sometimes leave environments implicit, writing e.g. $P \approx Q$.

We extend $\approx$ to possibly non-localised closed terms by relating them when their minimal localisations are related by $\approx$ (given $\Gamma \vdash P \triangleright \Delta$, its *minimal localisation* adds empty queues to P for the input shared channels in Γ and session channels in Δ that are missing their queues). Further $\approx$ is extended to open terms in the standard way [HY95].

3.3.3 Properties of Asynchronous Session Bisimilarity

This subsection studies central properties of asynchronous session semantics.

Characterisation of reduction congruence. We first show that the bisimilarity coincides with the naturally defined reduction-closed congruence [HY95], given below.

Theorem 3.3.1 (Soundness and Completeness)**.** $\approx = \cong$.

Proof. The soundness ($\approx \subset \cong$) is by showing $\approx$ is a congruence. Since we are dealing with closed and typable terms, input and output prefix context closure is straightforward. The most difficult case is a closure under parallel composition, which requires us to check the side condition $\Delta_1' \rightleftharpoons \Delta_2'$ for each case.

The completeness direction ($\cong \subset \approx$) follows [Hen07, § 2.6] where we prove that every external action is definable by a testing process $T\langle N, succ, \ell\rangle$. The testing process uses a fresh name *succ* that allows us to detect an observable action ℓ based on the reduction closure of the reduction-closed congruence. See Appendix A.3.1 for details. □

Asynchrony, session determinacy and confluence. We study the properties of our asynchronous session bisimulations based on the notions of [PW97].

The next definition divides the labels ℓ into output actions and input actions.

Definition 3.3.10. Let us call ℓ an

1. *output action* if ℓ is one of $\overline{a}\langle s\rangle, \overline{a}(s), s!\langle v\rangle, s!(a), s \oplus \ell$.
2. *input action* if ℓ is one of $a\langle s\rangle, s?\langle v\rangle, s\&\ell$.

In the following, the first property says that we can delay an output arbitrarily, while the second says that we can always immediately perform a (well-typed) input.

Lemma 3.3.1 (Input and Output Asynchrony)**.** Suppose $\Gamma \vdash P \triangleright \Delta \overset{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

- *(input advance)* If ℓ is an input action, then $\Gamma \vdash P \triangleright \Delta \overset{\ell}{\longrightarrow\!\!\Longrightarrow} P' \triangleright \Delta'$.
- *(output delay)* If ℓ is an output action, then $\Gamma \vdash P \triangleright \Delta \Longrightarrow\!\!\overset{\ell}{\longrightarrow} P' \triangleright \Delta'$.

Proof. The proof is done by induction on the length of the silent transition. For the proof we utilise an intermediate result (see Lemma A.4.1) where we prove the permutation of a single hidden (τ) action and an input (resp. output) action. For the full proof, see Appendix A.4.1. □

The result of the above lemma starts from the fact that a single hidden action and an input (resp. output) action can be permuted in advance (resp. delay). Output and input actions are asynchronous and affect endpoint configuration terms (with the exception of action $\overline{a}\langle s \rangle$ that is observed on the asynchronous term $\overline{a}\langle s \rangle$). For example assume the transition:

$$\Gamma \vdash P_1 \mid s[\mathtt{i} : \vec{h_1}] \triangleright \Delta_1 \Longrightarrow P_2 \mid s[\mathtt{i} : \vec{h_2}] \triangleright \Delta_2 \overset{s?\langle v \rangle}{\longrightarrow} P_2 \mid s[\mathtt{i} : \vec{h_2} \cdot v] \triangleright \Delta_2 \Longrightarrow P_3 \mid s[\mathtt{i} : \vec{h_3}] \triangleright \Delta_3$$

Due to the asynchronous nature of the typed LTS, it is always safe to observe an input action before a series of silent actions. We can now observe:

$$\Gamma \vdash P_1 \mid s[\mathtt{i} : \vec{h_1}] \triangleright \Delta \overset{s?\langle v \rangle}{\longrightarrow} P_1 \mid s[\mathtt{i} : \vec{h_1} \cdot v] \triangleright \Delta \overset{s?\langle v \rangle}{\longrightarrow\!\!\Longrightarrow} P_3 \mid s[\mathtt{i} : \vec{h_3}] \triangleright \Delta_3$$

A similar example for an output action would be:

$$\Gamma \vdash P_1 \mid s[\mathtt{o} : v \cdot \vec{h_1}] \triangleright \Delta_1 \Longrightarrow P_2 \mid s[\mathtt{i} : v \cdot \vec{h_2}] \triangleright \Delta_2 \overset{s!\langle v \rangle}{\longrightarrow} P_2 \mid s[\mathtt{i} : \vec{h_2}] \triangleright \Delta_2 \Longrightarrow P_3 \mid s[\mathtt{i} : \vec{h_3}] \triangleright \Delta_3$$

An output action can be observed after a series of silent actions:

$$\Gamma \vdash P_1 \mid s[\mathtt{o} : v \cdot \vec{h_1}] \triangleright \Delta_1 \Longrightarrow \Gamma \vdash P_3 \mid s[\mathtt{o} : v \cdot \vec{h_1}] \triangleright \Delta_1 \overset{s!\langle v \rangle}{\longrightarrow} P_3 \mid s[\mathtt{i} : \vec{h_3}] \triangleright \Delta_3$$

We follow the work on the confluence for the π-calculus in [PW97], to define determinacy and confluence. Below and henceforth we often omit the environments in typed transitions. The properties of determinacy and confluence characterise all derivatives of a process.

Definition 3.3.11 (Process Derivative)**.** We say $\Gamma' \vdash Q \triangleright \Delta'$ is a *derivative* of $\Gamma \vdash P \triangleright \Delta$ if there exists $\vec{\ell}$ such that $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} \Gamma' \vdash Q \triangleright \Delta'$.

A process derivative of the typed process $\Gamma \vdash P \triangleright \Delta$ is any process that is derived by any sequence of transitions on $\Gamma \vdash P \triangleright \Delta$.

Definition 3.3.12 (Determinacy)**.** We say $\Gamma \vdash P \triangleright \Delta$ is *determinate* if for each derivative $\Gamma' \vdash Q \triangleright \Delta'$ of P and action ℓ, if $\Gamma' \vdash Q \triangleright \Delta' \stackrel{\ell}{\longrightarrow} Q' \triangleright \Delta_1$ and $\Gamma' \vdash Q \triangleright \Delta' \stackrel{\widehat{\ell}}{\Longrightarrow} Q'' \triangleright \Delta_2$ then $Q' \approx Q''$.

We define the notion of session transitions, where any transition on session channels is called a session transition. Furthermore, a process that only exhibits traces of session transitions is called session determinate:

Definition 3.3.13 (Session Determinacy)**.** Let us write $P \stackrel{\ell}{\longrightarrow}_s Q$ if $P \stackrel{\ell}{\longrightarrow} Q$ where if $\ell = \tau$ then it is generated without using [Request1], [Request2], [Accept], in Figure 3.3 (i.e. a communication is performed without session initiation actions). We extend the definition to $\stackrel{\vec{\ell}}{\Longrightarrow}_s$ and $\stackrel{\widehat{\ell}}{\Longrightarrow}_s$ etc. We say P is *session determinate* if P is typable and localised and if $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} Q \triangleright \Delta'$ then $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow}_s Q \triangleright \Delta'$. We call such Q a *session derivative* of P.

We follow the terminology from [PW97] to define the *weight of action* ℓ_1 *over action* ℓ_2, which is used to define confluence:

Definition 3.3.14 (Action Weight)**.** We define $\ell_1 \lfloor \ell_2$ as

1. $\overline{a}\langle s \rangle$ if $\ell_1 = \overline{a}(s')$ and $s' \in \mathtt{bn}(\ell_2)$.
2. $s!\langle s' \rangle$ if $\ell_1 = s!(s')$ and $s' \in \mathtt{bn}(\ell_2)$.

3. $s!\langle a\rangle$ if $\ell_1 = s!(a)$ and $a \in \mathtt{bn}(\ell_2)$

4. ℓ_1 otherwise.

We write that $\ell_1 \bowtie \ell_2$ when $\ell_1 \neq \ell_2$ and if ℓ_1, ℓ_2 are input actions then $\mathtt{subj}(\ell_1) \neq \mathtt{subj}(\ell_2)$.

The main intuition for the definition of the $\ell_1 \lfloor \ell_2$ action comes from the order in which these two actions are observed on a process. For example consider the actions:

$$\Gamma \vdash P \triangleright \Delta \xrightarrow{s_1!(a)} \Gamma \vdash P_1 \triangleright \Delta_1$$
$$\Gamma \vdash P \triangleright \Delta \xrightarrow{s_2!(a)} \Gamma \vdash P_2 \triangleright \Delta_2$$

If we were to observe on process $\Gamma \vdash P_2 \triangleright \Delta_2$ an output of the shared name a through session channel s_1 (i.e. action $s_1!\langle a\rangle$ after $s_2!(a)$) then we would observe the transition:

$$\Gamma \vdash P_2 \triangleright \Delta_2 \xrightarrow{s_1!(a)\lfloor s_2!(a)} \Gamma \vdash P_2' \triangleright \Delta_2'$$

where $s_1!(a)\lfloor s_2!(a) = s_1!\langle a\rangle$ because a was already extruded out of the scope of P. On the other hand, if we wanted to observe action $s_2!\langle a\rangle$ after $s_1!(a)$ we get the transition:

$$\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{s_2!(a)\lfloor s_1!(a)} \Gamma \vdash P_1' \triangleright \Delta_1'$$

with $s_2!(a)\lfloor s_1!(a) = s_2!\langle a\rangle$

Operator $\bowtie$ is used for the confluence definition and relates two actions that are different and moreover if they are input actions they are observed on different names.

Milner [Mil80] stated about the property of confluence that "of any two possible actions, the occurence of one will never preclude the other". This is captured by the confluence definition for the π-calculus [PW97]:

Definition 3.3.15 (Confluence)**.** We say $\Gamma \vdash P \triangleright \Delta$ is *confluent* if for each derivative Q of P and actions ℓ_1, ℓ_2 such that $\ell_1 \bowtie \ell_2$,

1. if $\Gamma \vdash Q \triangleright \Delta \xrightarrow{\ell} Q_1 \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta \xRightarrow{\ell} Q_2 \triangleright \Delta_2$, then $\Gamma \vdash Q_1 \triangleright \Delta_1 \Longrightarrow Q_1' \triangleright \Delta_1'$ and $\Gamma \vdash Q_2 \triangleright \Delta_2 \Longrightarrow Q_2' \triangleright \Delta_2'$ with $Q_1' \approx Q_2'$.
2. if $\Gamma \vdash Q \triangleright \Delta \xrightarrow{\ell_1} Q_1 \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta \xRightarrow{\ell_2} Q_2 \triangleright \Delta_2$, then $\Gamma \vdash Q_1 \triangleright \Delta_1 \xRightarrow{\widehat{\ell_2 \lfloor \ell_1}} Q_1' \triangleright \Delta_1'$ and $\Gamma \vdash Q_2 \triangleright \Delta_2 \xRightarrow{\widehat{\ell_1 \lfloor \ell_2}} Q_2' \triangleright \Delta_2'$ with $Q_1' \approx Q_2'$.

The next Lemmas are used to prove Theorem 3.3.2. The first Lemma states that session determinate processes are semantically (i.e. up-to typed bisimulation) invariant under silent actions:

Lemma 3.3.2. Let P be session determinate and $\Gamma \vdash P \triangleright \Delta \Longrightarrow Q \triangleright \Delta'$. Then $P \approx Q$.

Proof. For proof, see Appendix A.4.2 □

Lemma 3.3.3. Assume typable, localised P and actions ℓ_1, ℓ_2 such that $\mathtt{subj}(\ell_1), \mathtt{subj}(\ell_2)$ are session names and $\ell_1 \bowtie \ell_2$. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_2} P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta \xrightarrow{\ell_2 \lfloor \ell_1} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta \xrightarrow{\ell_1 \lfloor \ell_2} P' \triangleright \Delta'$

Proof. The proof considers the fact that ℓ_1 and ℓ_2 have different session subjects and are observed on session endpoint configurations. For proof, see Appendix A.4.3. □

We show that session transitions are determinate transitions.

Lemma 3.3.4. Let P be session determinate. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} P' \triangleright \Delta'$ and $\Gamma \vdash P \triangleright \Delta \xRightarrow{\widehat{\ell}} P'' \triangleright \Delta''$ then $P' \approx P''$

Proof. There are two cases:

Case: τ:

Follow Lemma 4.3.2 to get $P \approx P'$ and $P \approx P''$. The result then follows.

Case: ℓ:

Suppose that $P \xrightarrow{\ell}_s P'$ and $P \overset{\ell}{\Longrightarrow}_s P''$ implies $P \Longrightarrow_s P_1 \xrightarrow{\ell}_s P_2 \Longrightarrow_s P''$. From Lemma 4.3.2, we can conclude that $P \approx P_1$ and because of the bisimulation definition, we have $P' \approx P_2$ to complete we call upon Lemma 4.3.2 once more to get $P' \approx P''$ as required.

For proof, see Appendix A.4.4. □

We show that session transitions are confluent transitions.

Lemma 3.3.5. Let P be session determinate and $\ell_1 \bowtie \ell_2$. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \overset{\ell_2}{\Longrightarrow} P_2 \triangleright \Delta_2$, then $\Gamma \vdash P_1 \triangleright \Delta_1 \overset{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta_2 \overset{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} P'' \triangleright \Delta''$ and $P' \approx P''$.

Proof. We do a case analysis on the labels ℓ_1 and ℓ_2. The case analysis follows the pattern: If $P \xrightarrow{\ell_1}_s P_1$ and $P \Longrightarrow_s \xrightarrow{\ell_2}_s \Longrightarrow_s P_2$ then $P_1 \Longrightarrow_s \xrightarrow{\widehat{\ell_2 \lfloor \ell_1}}_s \Longrightarrow_s P_1'$ and $P_1 \Longrightarrow_s \xrightarrow{\widehat{\ell_1 \lfloor \ell_2}}_s \Longrightarrow_s P_2'$, where in each case we use Lemmas 3.3.1 and 4.3.3 to permute the order of the actions $\Longrightarrow_s, \ell_1, \ell_2, \widehat{\ell_2 \lfloor \ell_1}, \widehat{\ell_1 \lfloor \ell_2}$ to get the required result. For proof, see Appendix A.4.5. □

The above lemma states formally a basic intuition about session types, that is due to the linearity of usage of session channels, one session action cannot preclude another session action, making a session determinate process confluent. The last two lemmas are expressed by the next theorem.

Theorem 3.3.2 (Session Determinacy)**.** Let P be session determinate. Then P is determinate and confluent.

Proof. From the definition of confluence (resp. determinacy) and from the definition of P we have that each derivative Q of P is also session determinate. The proof is an immediate result of Lemma 4.3.5 (resp. Lemma 4.3.4). □

The confluence property is used to reason about the behaviour of systems. In the following definition we build a relation on determinate processes that is later shown to be a bisimulation up-to determinate transitions. We use this relation later in the thesis to reason about session determinate processes and especially event-based optimisations.

Definition 3.3.16 (Determinate Up-to expansion Relation)**.** Let $\mathscr{R}$ be a symmetric, typed relation such that if $\Gamma \vdash P \triangleright \Delta \mathscr{R} Q \triangleright \Delta$, then if

1. P, Q are determinate;
2. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \Gamma' \vdash P'' \triangleright \Delta''$ then $\Gamma \vdash Q \triangleright \Delta \stackrel{\ell}{\Longrightarrow} \Gamma' \vdash Q' \triangleright \Delta'$ and $\Gamma' \vdash P'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash P' \triangleright \Delta'$ with $\Gamma' \vdash P' \triangleright \Delta' \mathscr{R} Q' \triangleright \Delta'$;
3. the symmetric case.

Then we call $\mathscr{R}$ a *determinate up-to expansion relation*, or often simply *up-to expansion relation*.

Lemma 3.3.6. Let $\mathscr{R}$ be an up-to expansion relation. Then $\mathscr{R} \subset \approx$.

Proof. The proof is easy by showing $\Longrightarrow \mathscr{R} \Longleftarrow$ is a bisimulation. Denote this relation as $\mathscr{S}$. We can easily check that $\mathscr{S}$ is a bisimulation, using determinacy (commutativity with other actions). □

Chapter 4

Eventful Session Types Behavioural Theory

The Asynchronous Session π-calculus, is extended in this Chapter to describe the event-driven programming paradigm. To transit from asynchronous communication to an event-driven model we first need to recognise and define the notion of the event in a session type context. An event can be defined as an abstraction with three properties:

1. *Asynchrony*: An event is a computational state change (i.e an action) that happens concurrently and asynchronously with respect to the computation.
2. *Detectability*: An event action can be detected by the underlying computation.
3. *Type*: An event has a type that can be recognised and may drive the computation process.

The next step is to describe the three properties of an event in terms of the Asynchronous Session π-Calculus, so we can undestand the extension choices made for the *Eventful Session Type π-Calculus*.

ASP offers fine grained communication semantics, with the non-blocking property of asynchrony and the order-preserving property of session types. The first event property defines an event as the asynchronous *arrival* of a message in a session configuration. The second event property requires for a primitive operator, able to interact with session configurations and detect whether a message has arrived. From the last property of an event, we can see a correlation between events and sessions types. One way to understand a session type system is that a session type drives a process computation. Following this intuition, we expect an event to be correlated with a session type and more specifically an event should be correlated with the *runtime* session type of the session channel that has received a message. To complete the eventful framework we should define a type match operator for session types, able to decide about process continuation based on the inspection of the runtime session type.

In this Chapter we extend the Asynchronous Session π-Calculus to the Eventful Session π-Calculus or ESP for short. We define the event-driven extensions in the syntax and operational semantics, which now include typing notions to cope with event types. As a consequence, the typing system passes through a major extension to support event typing. Despite the extension, we use a subtyping relation to present the ASP type system as a superset of the ESP type system.

The behavioural theory undergoes minor changes in the definitions with respect to the definitions in ASP. The definition for the untyped labelled transition system and the definition of the labelled transition system for session environments for ESP are technically the same as the definitions for the ASP. Together both systems define the notion of a typed process transition that gives rise to a bisimilarity relation. The bisimilarity coincides with a corresponding reduction congruence relation. We slightly adjust the definitions for the confluence theory in the ASP to define a confluence theory for the ESP with similar results.

4.1 A Calculus for Eventful Sessions

4.1.1 Syntax of the Eventful Session π-Calculus

The Asynchronous Session π-calculus is extended with a minimal set of event-driven session programming constructs, that cooperate with the asynchronous nature of ASP to form an event-driven model of computation.

To make the transition from ASP to an eventful framework, we introduce the *message arrival predicate*, which we consider essential for the definition of an event-driven framework. The message arrival predicate is used for event detection: it interacts with a (session and shared) endpoint configuration and returns the boolean value true if the configuration is non-empty and false otherwise. The second construct we introduce is the *session typecase* [ACPP89]. The event-driven paradigm is characterised by a reactive flow of control that introduces a dynamic execution of a program. The typecase construct was first used in the λ-calculus (cf. [ACPP89]) to adjust the dynamic nature of the event-driven framework in a statically checked typing framework. We follow the same motivation to define the typecase construct for session names in the context of the π-calculus. The session typecase is typed with the session set type syntax, while the session typing system is adjusted to handle session set types and type matching. We call this extension the Eventful Session π-calculus or ESP for short.

Figure 4.1 presents the extensions of ESP syntax. We explain the extension from the ASP. The ASP syntax description can be found in § 3.1.1.

Expressions e include the *message arrival predicates*: `arrive` u checks if any session initiation request message is present (has arrived) at shared name endpoint u, `arrive` k checks if any session message is present (has arrived) in the `i`-queue of a session endpoint k, and `arrive` k h checks if the first available message of the `i`-queue, if any, is specifically h.

(Processes)	P,Q	$::=$	$u(x:S).P$	Accept
		$\mid$	$\overline{u}(x:S).P$	Request
		$\mid$	$k!\langle e\rangle;P$	Sending
		$\mid$	$k?(x);P$	Receiving
		$\mid$	$k \oplus l;P$	Selection
		$\mid$	$k\&\{l_i : P_i\}_{i\in I}$	Branching
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	Conditional
		$\mid$	$(\nu\ a)P$	Hiding
		$\mid$	$P \mid Q$	Parallel
		$\mid$	$\mathbf{0}$	Inaction
		$\mid$	$\mu X.P$	Recursion
		$\mid$	X	Variable
		$\mid$	$\texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i\in I}$	Typecase
		$\mid$	$a[\vec{s}]$	Shared Configuration
		$\mid$	$\overline{a}\langle s\rangle$	Asynchronous Request
		$\mid$	$(\nu\ s)P$	Session Hiding
		$\mid$	$s[S, \texttt{i} : \vec{h}, \texttt{o} : \vec{h'}]$	Session Configuration
(Identifiers)	u	$::=$	$a,b \mid x,y$	
	k	$::=$	$s,\overline{s} \mid x,y$	
	n	$::=$	$a,b \mid s,\overline{s}$	
(Values)	v	$::=$	$\texttt{tt},\texttt{ff} \mid a,b \mid s,\overline{s}$	
(Expressions)	e	$::=$	$v \mid x,y,z \mid e = e \mid \texttt{arrive}\ u \mid \texttt{arrive}\ k \mid \texttt{arrive}\ k\ h$	
(Messages)	h	$::=$	$v \mid l$	

Figure 4.1: The syntax of ESP processes.

To introduce a type matching primitive we first need to carry inside the process the session type of a session channel, which we call *session runtime type*. Session runtime type is carried along session initiation and session endpoints.

The session initiation actions on shared channels are the request $\overline{u}(x : S).P$ and the accept $u(x : S).P$ actions. The annotation S specifies the session runtime type that directs how the bound channel s should be used.

Session endpoints $s[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}]$ are annotated with session type S, called *session runtime type of* k, that defines an endpoint's session typed behaviour.

The $\mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$ attempts to match the session runtime type of channel k against the specified session types S_i, proceeding to the P_i for the first S_i that matches. The $\mathtt{typecase}$ acts as a binder for each of the $(x_i)_{i \in I}$ variables in the corresponding $\{P_i\}_{i \in I}$ process.

4.1.2 Structural Congruence

Structural congruence in Figure 4.2, defines a minimal congruence for the ESP syntax that uses the same defining principles as Figure 3.2. Rule $s[\mu X.S] \equiv s[S\{\mu X.S/X\}]$ is added as the eventful extension to describe session runtime recursive unfolding.

4.1.3 Operational Semantics of the Eventful Session π Calculus

Figure 4.3 gives the operational semantics for the ESP. The reference description for the operational semantics is Figure 3.3. The distinction between the ESP and the ASP operational semantics lies on the handling of the runtime syntax carried by the session endpoints. The session endpoint runtime syntax is reduced along the processes' reduction actions to maintain a consistent runtime session type of a (non-endpoint) process. Whenever an interaction

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & \\
\mu X.P & \equiv & P\{\mu X.P/X\} & \\
P & \equiv & Q & \text{if } P \equiv_\alpha Q \\
(P_1 \mid P_2) \mid P_3 & \equiv & P_1 \mid (P_2 \mid P_3) & \\
\mathbf{0} & \equiv & (\nu\, n)\mathbf{0} & \\
(\nu\, n)P \mid Q & \equiv & (\nu\, n)(P \mid Q) & \text{if } n \notin \mathtt{fn}(Q) \\
P \mid \mathbf{0} & \equiv & P & \\
\mathbf{0} & \equiv & (\nu\, a)\, a[\varepsilon] & \\
s[\mu X.S] & \equiv & s[S\{\mu X.S/X\}] & \\
\mathbf{0} & \equiv & (\nu\, s)(s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) &
\end{array}
$$

Figure 4.2: Structural congruence.

between a session channel and an endpoint configuration is performed, the runtime syntax of the session configuration is reduced accordingly. Note that for a reduction to take place, the action performed and the runtime session type should agree. For example if a send action is taking place, the corresponding runtime session type should be send prefixed. The runtime session type of a process is used for type matching in the semantics of the typecase construct. Specifically rules [Request1] and [Accept] create session endpoints with the proper runtime syntax S carried by the definition of $\overline{a}(S : s).P$ and $a(S : s).P$ prefixed processes. Rules [Send] and [Receive] reduce the send and receive session type (session type syntax for ASP are defined in §3.2.1 and session type syntax for ESP is defined in § 4.2.1) respectively. Similarly for rules [Select] and [Branch]. The [Comm] rule does not affect the state of the runtime syntax since there is no reduction of a non-endpoint process. The extension of the reduction relation includes the semantics for the `arrive` and `typecase` construct. Rules [Arriv-req], [Arrive-sess]

$$\frac{(s \notin \mathtt{fn}(P))}{\overline{a}(x:S).P \longrightarrow (\nu\ s)(P\{\overline{s}/x\} \mid \overline{s}[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid \overline{a}\langle s\rangle)} \quad \text{[Request1]}$$

$$a[\vec{s}] \mid \overline{a}\langle s\rangle \longrightarrow a[\vec{s}\cdot s] \quad \text{[Request2]}$$

$$a(x:S).P \mid a[s\cdot\vec{s}] \longrightarrow P\{s/x\} \mid s[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid a[\vec{s}] \quad \text{[Accept]}$$

$$s!\langle v\rangle;P \mid s[!\langle U\rangle;S, \mathtt{o}:\vec{h}] \longrightarrow P \mid s[S, \mathtt{o}:v\cdot\vec{h}] \quad \text{[Send]}$$

$$s?(x);P \mid s[?(U);S, \mathtt{i}:v\cdot\vec{h}] \longrightarrow P\{v/x\} \mid s[S, \mathtt{i}:\vec{h}] \quad \text{[Receive]}$$

$$\frac{(i \in J)}{s \oplus l_i;P \mid s[\oplus\{l_j:S_j\}_{j\in J}, \mathtt{o}:\vec{h}] \longrightarrow P \mid s[S_i, \mathtt{o}:\vec{h}\cdot l_i]} \quad \text{[Select]}$$

$$\frac{(i' \in J \subseteq I)}{s\&\{l_i:P_i\}_{i\in I} \mid s[\&\{l_j:S_j\}_{j\in J}, \mathtt{i}:l_{i'}\cdot\vec{h}] \longrightarrow P_{i'} \mid s[S_{i'}, \mathtt{i}:\vec{h}]} \quad \text{[Branch]}$$

$$s[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'] \longrightarrow s[\mathtt{o}:\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'\cdot v] \quad \text{[Comm]}$$

$$\frac{((|\vec{s}| \geq 1) \downarrow b)}{E[\mathtt{arrive}\ a] \mid a[\vec{s}] \longrightarrow E[b] \mid a[\vec{s}]} \quad \text{[Arrive-req]}$$

$$\frac{((|\vec{h}| \geq 1) \downarrow b)}{E[\mathtt{arrive}\ s] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[b] \mid s[\mathtt{i}:\vec{h}]} \quad \text{[Arrive-sess]}$$

$$\frac{((\vec{h} = h\cdot\vec{h}') \downarrow b)}{E[\mathtt{arrive}\ s\ h] \mid s[\mathtt{i}:\vec{h}] \longrightarrow E[b] \mid s[\mathtt{i}:\vec{h}]} \quad \text{[Arrive-msg]}$$

$$\frac{(\exists k \in I, \forall j < k \cdot S_j \not\leq S \wedge S_k \leq S)}{\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:S_i):P_i\}_{i\in I} \mid s[S] \longrightarrow P_k\{s/x_k\} \mid s[S_k]} \quad \text{[Typecase]}$$

$$\mathtt{if}\ \mathtt{tt}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P \quad \text{[If-true]}$$

$$\mathtt{if}\ \mathtt{ff}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q \quad \text{[If-false]}$$

$$e \longrightarrow e' \implies E[e] \longrightarrow E[e'] \quad \text{[Eval]}$$

$$P \longrightarrow P' \implies (\nu\ a)P \longrightarrow (\nu\ a)P' \quad \text{[Chan]}$$

$$P \longrightarrow P' \implies (\nu\ s)P \longrightarrow (\nu\ s)P' \quad \text{[Sess]}$$

$$P \longrightarrow P' \implies P \mid Q \longrightarrow P' \mid Q \quad \text{[Par]}$$

$$P \equiv P' \longrightarrow Q' \equiv Q \implies P \longrightarrow Q \quad \text{[Struct]}$$

Figure 4.3: Reduction rules for Eventful Session π-calculus.

denote that the arrive expression inside a context is reduced to the Boolean value true (`tt`) if the corresponding input endpoint is not empty and the Boolean value false (`ff`) if the corresponding input endpoint is empty. Rule [Arrive-msg] requires that expression `arrive` k h returns a true (`tt`) value if a specific value h is prefixed at input endpoint k and false otherwise (`ff`). The operation of the typecase construct requires that a session channel's runtime type S is type-checked against the session types defined in the typecase's defining body $\{S_i\}_{i\in I}$. The first match up-to subtyping S_k chooses the substitution of the corresponding bound variable x_k with session channel s on P_k as continuation.

We give examples for the use of the `arrive` and `typecase` constructs.

Example 4.1.1 (Usage of `arrive` and `typecase`)**.**

(1) Usage of `arrive`:

Define process:

$$P = \texttt{if arrive } s \texttt{ then } (s?(x); \texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2) \texttt{ else } P_3$$

and session endpoint configurations:

$$\begin{aligned} B_1 &= s[\texttt{i} : v_1 \cdot v_2] \\ B_2 &= s[\texttt{i} : v_1] \\ B_3 &= s[\texttt{i} : \varepsilon] \end{aligned}$$

Process $P \mid B_3$ yields the reduction:

$$\texttt{if arrive } s \texttt{ then } (s?(x); \texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2) \texttt{ else } P_3 \mid B_3 \longrightarrow P_3 \mid B_3$$

since the first `arrive` expression would return false on the empty B_3.

Process $P \mid B_1$ reduces as:

$$\begin{aligned} P \mid B_1 &\longrightarrow s?(x);\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2 \mid B_1 \\ &\longrightarrow (\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2)\{v_1/x\} \mid B_2 \\ &\longrightarrow P_1\{v_1/x\} \mid B_2 \end{aligned}$$

After the first reduction the process $s?(x);\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2$ consumes the first message in the `i`-queue to get another `arrive`-prefixed process. A third reduction proceeds with process $P_1 \mid B_2$.

In the third example case we have the reductions: $P \mid B_2$ returns true on the first `arrive`-inspection:

$$\begin{aligned} P \mid B_2 &\longrightarrow s?(x);\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2 \mid B_2 \\ &\longrightarrow (\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2)\{v_1/x\} \mid B_3 \\ &\longrightarrow P_2\{v_1/x\} \mid B_3 \end{aligned}$$

where the first `arrive`-inspection returns true and proceeds with the receive prefixed process $s?(x);\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2$. In the second transition the only message in B_2 is consumed and the session endpoint configuration now remains empty. The third reduction `arrive`-inspects the empty session configuration to return false and proceed with the process P_2 composed in parallel with the empty configuration B_3.

(2) Usage of `typecase`:

Let process:

$$P = \texttt{typecase } s \texttt{ of } \{(x_1 : S_1) : P_1, (x_2 : S_2) : P_2\}$$

and session endpoint configurations:

$$\begin{aligned} B_1 &= s[S_1] \\ B_2 &= s[S_2] \end{aligned}$$

In process $P \mid B_1$ the `typecase` operation matches the type S_1 in B_1, with the $(x_1 : S_1) : P_1$ case in the `typecase` body and reduces to $P_1\{s/x_1\} \mid B_1$ with the bound variable x_1 substituted by session channel s:

$$\texttt{typecase}\ s\ \texttt{of}\ \{(x_1 : S_1) : P_1, (x_2 : S_2) : P_2\} \mid s[S_1] \longrightarrow P_1\{s/x_1\} \mid B_1$$

Similarly in process $P \mid B_2$ we reduce as:

$$\texttt{typecase}\ s\ \texttt{of}\ \{(x_1 : S_1) : P_1, (x_2 : S_2) : P_2\} \mid s[S_2] \longrightarrow P_2\{s/x_2\} \mid B_2$$

4.2 Types for Eventful Session Processes

The eventful instance of the Asynchronous Session π-calculus introduces an extension to the session typing discipline. This extension depends on the impact that both of the event-driven primitives, `arrive` and `typecase` operators, have on the ASP .

The `arrive` inspection predicate can be viewed as an expression that evaluates to a Boolean type constant. The typing system extension for the `arrive` keyword, is limited in the typing of the `arrive` expression with respect to the typing environment Γ and the session type environment Δ.

To type the `typecase` construct we introduce a new construct on session types, called *session set type*. Furthermore, there is the need to distinguish the type of the actually created session channels and the type of `typecase` prefixed processes. The distinction is lifted up-to a subtyping relation for session set types to create a unified session type theory between the ASP and the ESP.

The requirements of the last paragraph predispose a complicated extension for a session typing system in the eventful context. Nevertheless the typing system developed in this section can be seen as a straightforward extension of the typing system in § 3.2.

4.2.1 Syntax

The type syntax is an extension of the typing system in § 3.2, with *session set types*. This simple extension allows us to treat type-safe event handling for an arbitrary collection of differently typed communication channels.

$$
\begin{array}{lll}
\text{(Shared)} & U & ::= \texttt{bool} \mid \texttt{i}\langle S\rangle \mid \texttt{o}\langle S\rangle \\
\text{(Value)} & T^{\,e} & ::= U \mid S \\
\text{(Session)} & S^e & ::= !\langle T\rangle;S^e \mid ?(T);S^e \mid \oplus\{l_i : S_i^e\}_{i\in I} \mid \&\{l_i : S_i^e\}_{i\in I} \mid \{S_i^e\}_{i\in I} \\
 & & \quad \mid \mu X.S^e \mid X \mid \texttt{end}
\end{array}
$$

The eventful session types S^e, is extended from the syntax in § 3.2.1 with the introduction of *session set type* $\{S_i\}_{i\in I}$, which represents a set of possible behaviours designated by the S_i^e. Session set types are used to type the `typecase` construct. The shared types U are identical to the shared types in § 3.2.1. Shared channel types $\texttt{i}\langle S\rangle, \texttt{o}\langle S\rangle$ are defined on the session types definition from ASP - session types without session set type.

The notation S^e is distinguished from notation S for ASP syntax of session types defined in § 3.2.1. Notation S is used to define the syntax and operational semantics of ESP in § 4.1 and particularly the session runtime type. S^e is used in the typing system to type the `typecase` construct.

To understand this distinction consider the semantics for creating a session endpoint. If we allow S^e in the definition of $a(x : S).P$ and the definition of session runtime typing then it will be possible to create endpoints of the type $s[\{S_i\}_{i\in I}, \texttt{i} : \varepsilon, \texttt{o} : \varepsilon]$. This session endpoint has the intuition of an arbitrary non-deterministic choice from a set of session types, which is not supported by the intuition, syntax and semantics for the ESP.

$$
\begin{array}{rcl}
\mathscr{F}(\mathscr{R}) & = & \{(\mathtt{bool},\mathtt{bool}),(\mathtt{end},\mathtt{end})\} \\
& \cup & \{(\mathtt{i}\langle S\rangle,\mathtt{i}\langle S'\rangle),(\mathtt{o}\langle S\rangle,\mathtt{o}\langle S'\rangle) \mid (S,S'),(S',S)\in\mathscr{R}\} \\
& \cup & \{(!\langle T_1\rangle;S_1,!\langle T_2\rangle;S_2) \mid (T_2,T_1),(S_1,S_2)\in\mathscr{R}\} \\
& \cup & \{(?(T_1);S_1,?(T_2);S_2) \mid (T_1,T_2),(S_1,S_2)\in\mathscr{R}\} \\
& \cup & \{(\oplus\{l_i:S_i\}_{i\in I},\oplus\{l_j:S'_j\}_{j\in J}) \mid I\subseteq J,\forall i\in I.(S_i,S'_i)\in\mathscr{R}\} \\
& \cup & \{(\&\{l_i:S_i\}_{i\in I},\&\{l_j:S'_j\}_{j\in J}) \mid J\subseteq I,\forall j\in J.(S_j,S'_j)\in\mathscr{R}\} \\
& \cup & \{(\mu X.S,S') \mid (S\{\mu X.S/X\},S')\in\mathscr{R}\} \\
& \cup & \{(S,\mu X.S') \mid (S,S'\{\mu X.S'/X\})\in\mathscr{R}\} \\
& \cup & \{(\{S_i\}_{i\in I},\{S'_j\}_{j\in J}) \mid \forall j\in J,\exists i\in I.(S_i,S'_j)\in\mathscr{R}\} \\
& \cup & \{(\{S\},S') \mid (S,S')\in\mathscr{R}\}
\end{array}
$$

Figure 4.4: The generating function for the eventful session subtyping relation.

4.2.2 Session Subtyping

Subtyping is defined with respect to the subtyping relation in § 3.2.2. The generating function in eventful types is extended to include the session set type subtyping.

As in § 3.2.2, the subtyping relation is defined on the set of all closed and contractive types $\mathscr{T}$: for $T',T\in\mathscr{T}$, T' is a subtype of T, written $T'\leq T$, if (T',T) is in the largest fixed point of the monotone function:

$$\mathscr{F}:\mathscr{P}(\mathscr{T}\times\mathscr{T})\to\mathscr{P}(\mathscr{T}\times\mathscr{T})$$

given in Figure 4.4. We describe only the extension for session set types, with reference the the description in § 3.2.2. The ordering of set types in line 9, says that if every element in the set type $\{S'_j\}_{j\in J}$ has a subtype in $\{S_i\}_{i\in I}$, then the latter is at least as composable as the former. The final clause states that singleton set types are transparent (i.e. the enclosed type can be "unwrapped") up-to subtyping.

$$\overline{!\langle T\rangle;S} = ?(T);\overline{S} \qquad \overline{\&\{l_i : S_i\}_{i\in I}} = \oplus\{l_i : \overline{S_i}\}_{i\in I}$$
$$\overline{\mu X.S} = \mu X.\overline{S} \qquad \overline{\{S_i\}_{i\in I}} = \{\overline{S_i}\}_{i\in I}$$
$$\overline{?(T);S} = !\langle T\rangle;\overline{S} \qquad \overline{\oplus\{l_i : S_i\}_{i\in I}} = \&\{l_i : \overline{S_i}\}_{i\in I} \qquad \overline{X} = X \qquad \overline{\texttt{end}} = \texttt{end}$$

Figure 4.5: Session type duality.

The duality relation in Figure 4.5 is an extension of the duality relation in Figure 3.6, to include the duality relation of session set types. Session set types are dual following the structure of session set types as expected.

The semantics of $\leq$ are clarified through *duality*.

Lemma 4.2.1. $S_1 \leq S_2$ iff $\overline{S_2} \geq \overline{S_1}$.

Proof. Let us call any relation witnessing $\leq$ (i.e. which is a fixed point of the subtyping function), *a subtyping relation*. Because $\overline{\overline{S}} = S$, it suffices to show the relation $\{(\overline{S_2}, \overline{S_1}) \mid S_1 \leq S_2\}$ is a subtyping relation, which is immediate by construction. □

Definition 4.2.1 (Composable Types)**.** We define the set of *composable types* of a session type S as:

$$\texttt{comp}(S) = \{S' \mid S' \leq \overline{S}\},$$

That is, $\texttt{comp}(S)$ is the set of types which can be composed with S (note S and $\overline{S}$ are composable, hence if S' is smaller than $\overline{S}$, S' should be more composable with S).

Subtyping can be completely characterised by composability.

Proposition 4.2.1 (Subtyping Properties)**.** (1) $\leq$ is a preorder; (2) $S_1 \leq S_2$ if and only if $\texttt{comp}(S_2) \subseteq \texttt{comp}(S_1)$.

Proof. (1) is standard, while (2) uses Lemma 4.2.1. For both, see Appendix A.1 for details. □

4.2.3 Type System for Programs

This section follows very close the definition in § 3.2.3. For this reason the explanation focuses on the extensions made.

We define typing judgements for programs and expressions.

$$\Gamma \vdash P \triangleright \Delta \qquad \text{and} \qquad \Gamma, \Delta \vdash e : T$$

with

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma \cdot u : U \ \mid \ \Gamma \cdot X : \Delta \qquad \text{and} \qquad \Delta \ ::= \ \emptyset \ \mid \ \Delta \cdot k : S^e \ \mid \ \Delta \cdot a$$

The linear environment Δ is extended from the linear environment of ASP to contain session channels s typed with the event session syntax S^e, to include session set types.

Figure 4.6 defines the typing rules for ESP programs. The description of Figure 3.7 is considered as the core reference.

Rules (AReq), (AMsg), (AVal) and (ALab) extend the typing core typing system and type the `arrive` predicates with the boolean type; (AReq) checks that u is indeed a shared channel, and (AVal) checks that the specified v corresponds to the expected message type on that session.

The reader should bear in mind that the subsumption rule uses a refined subtyping relation to handle the interleaving of core session syntax S and event session syntax S^e. Rules (Req) and (Acc) check if the shared environment maps the shared channel to the output (resp. input) shared channel type, consistent with the S (session types without session set type) annotation and the usage of the bound session variable. The initiation annotation is restricted to ensure that the active type of the session at run-time has an S shape (see [Request1] and [Accept] in Figure 4.3), so that the execution of `typecase` can resolve the type of the session to a specific case. Note that the presence of S in $\Delta \cdot x : S$ can be achieved through subsumption (Subs).

$$\Gamma \cdot u : U \vdash u : U \quad \text{(SChan)} \qquad \Gamma \cdot u : \mathtt{i}\langle S \rangle \vdash u : \mathtt{o}\langle S \rangle \quad \text{(SChan')}$$

$$\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathtt{bool} \quad \text{(Bool)}$$

$$\frac{\Gamma \vdash n : T \vee \Delta = \Delta' \cdot n : T}{\Gamma, \Delta \vdash n : T} \quad \text{(Name)} \qquad \frac{\Gamma, \Delta \vdash e_i : T_i \quad i \in \{1,2\}}{\Gamma, \Delta \vdash e_1 = e_2 : \mathtt{bool}} \quad \text{(Match)}$$

$$\frac{\Gamma, \Delta \vdash u : \mathtt{i}\langle S \rangle}{\Gamma, \Delta \vdash \mathtt{arrive}\ u : \mathtt{bool}} \quad \text{(AReq)} \qquad \frac{\exists v, \Gamma, \Delta \vdash \mathtt{arrive}\ k\ v : \mathtt{bool}}{\Gamma, \Delta \vdash \mathtt{arrive}\ k\ : \mathtt{bool}} \quad \text{(AMsg)}$$

$$\frac{\Gamma, \Delta \vdash k : ?(U); S \quad \Gamma, \Delta \vdash v : U}{\Gamma, \Delta \vdash \mathtt{arrive}\ k\ v : \mathtt{bool}} \quad \text{(AVal)} \qquad \frac{\Gamma, \Delta \vdash k : \&\{l_i : S_i\}_{i \in I} \quad j \in I}{\Gamma, \Delta \vdash \mathtt{arrive}\ k\ l_j : \mathtt{bool}} \quad \text{(ALab)}$$

$$\frac{\Gamma \vdash a : \mathtt{o}\langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : \overline{S}}{\Gamma \vdash \overline{a}(x : \overline{S}).P \triangleright \Delta} \quad \text{(Req)} \qquad \frac{\Gamma \vdash a : \mathtt{i}\langle S \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x : S}{\Gamma \vdash a(x : S).P \triangleright \Delta} \quad \text{(Acc)}$$

$$\frac{\Gamma \vdash v : U \quad U \neq \mathtt{i}\langle S' \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot k : S^e}{\Gamma \vdash k!\langle v \rangle; P \triangleright \Delta \cdot k : !\langle U \rangle; S^e} \quad \text{(Send)} \qquad \frac{\Gamma \cdot x : U \vdash P \triangleright \Delta \cdot k : S^e \quad U \neq \mathtt{i}\langle S' \rangle}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(U); S^e} \quad \text{(Recv)}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S^e}{\Gamma \vdash k!\langle k' \rangle; P \triangleright \Delta \cdot k : !\langle So'^e \rangle; S^e \cdot k' : S'^e} \quad \text{(Deleg)} \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot k : S^e \cdot x : S'^e}{\Gamma \vdash k?(x); P \triangleright \Delta \cdot k : ?(S'^e); S^e} \quad \text{(SRecv)}$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot k : S^e}{\Gamma \vdash k \oplus l; P \triangleright \Delta \cdot k : \oplus\{l : S^e\}} \quad \text{(Sel)} \qquad \frac{\forall i \in I \quad \Gamma \vdash P_i \triangleright \Delta \cdot k : S_i^e}{\Gamma \vdash k \& \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot k : \&\{l_i : S_i^e\}_{i \in I}} \quad \text{(Bra)}$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta_i \quad i \in \{1,2\} \quad \mathtt{dom}(\Delta_1) \cap \mathtt{dom}(\Delta_2) = \emptyset}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cdot \Delta_2} \quad \text{(Conc)} \qquad \frac{\Gamma, \Delta \vdash e : \mathtt{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta} \quad \text{(If)}$$

$$\frac{\Gamma \cdot a : U \vdash P \triangleright \Delta \cdot a}{\Gamma \vdash (\nu\ a)P \triangleright \Delta} \quad \text{(CRes)} \qquad \frac{\Delta\ \mathtt{end\ only}}{\Gamma \vdash a[\varepsilon] \triangleright \Delta \cdot a} \quad \text{(EBuff)}$$

$$\frac{\Gamma \cdot X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \Gamma \triangleright \mu X.P\Delta} \quad \text{(Rec)} \qquad \Gamma \cdot X : \Delta \vdash X \triangleright \Delta \quad \text{(Var)}$$

$$\frac{\Delta\ \mathtt{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \quad \text{(Inact)} \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright \Delta'} \quad \text{(Subs)}$$

$$\frac{\forall i \in I \quad \Gamma \vdash P_i \triangleright \Delta \cdot x_i : S_i^e}{\Gamma \vdash \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i^e) : P_i\}_{i \in I} \triangleright \Delta \cdot k : \{S_i^e\}_{i \in I}} \quad \text{(Typecase)}$$

Figure 4.6: Typing rules for programs.

Rule (Typecase) types the `typecase`, which intuitively says that the usage of the target session channel in each of the sub-processes is collected into a set of possible behaviours represented by the session set type. For the latter reason the rules that type session channel prefixes, (Send) and (Recv), (Deleg), (Srecv), (Sel) and (Bra) use the S^e type in their definition. Rest of the rules follow the definition in Figure 3.7.

4.2.4 Type System for Run-time Syntax

The type System for Run-time Syntax is a straightforward extension of the Run-time Syntax in § 3.2.4. The description in § 3.2.4 is used as a reference. The message T definition extends its definition to include S^e.

$$\begin{array}{llll} \text{(General)} & \mathsf{T} ::= S^e \mid M & \text{(IMsg)} & M_{\mathtt{i}} ::= \emptyset \mid ?(T);M_{\mathtt{i}} \mid \&l;M_{\mathtt{i}} \\ \text{(Message)} & M ::= M_{\mathtt{i}} \mid M_{\mathtt{o}} & \text{(OMsg)} & M_{\mathtt{o}} ::= \emptyset \mid !\langle T\rangle;M_{\mathtt{o}} \mid \oplus l;M_{\mathtt{o}} \end{array}$$

The linear environment Δ adds the notation $[S]$ next to $s : \mathsf{T}$ to differ from definition in § 3.2.4. $[S]$ notation is used for recording the session runtime syntax from session endpoints in the linear typing, as explained next. We extend the grammar for the linear environment Δ:

$$\Delta ::= \emptyset \mid \Delta \cdot k : S^e \mid \Delta \cdot a \mid \Delta \cdot s : \mathsf{T}\,[S]$$

The definition and description of the $*$ parallel concatenation operator can be found in § 3.2.4. We only adjust the $*$ operator to be consistent with the $s : \mathsf{T}[S]$ notation in the defining rule:

$$\begin{aligned} \Delta_1 * \Delta_2 \quad = \quad & \Delta_1 \backslash \mathtt{dom}(\Delta_2) \cup \Delta_2 \backslash \mathtt{dom}(\Delta_1) \cup \{s : S * M\,[S] \mid \\ & s : S \in \Delta_i,\ s : M\,[S] \in \Delta_j \text{ where } i, j \in \{1,2\}, i \neq j\} \end{aligned}$$

Runtime typing rules in Figure 4.7 are an extended version of rules in Figure 3.8. Each rule records the runtime session typing of the session endpoint being typed in its $\mathsf{T}[S]$ notation.

$$\Gamma \vdash s[S, \mathtt{o} : \varepsilon] \triangleright s : \emptyset\,[S] \quad (\text{OutQ}) \qquad \Gamma \vdash s[S, \mathtt{i} : \varepsilon] \triangleright s : \emptyset\,[S] \quad (\text{InQ})$$

$$\frac{\Gamma \vdash s[S, \mathtt{o} : \vec{h}] \triangleright s : M_{\mathtt{o}}\,[S] \quad \Gamma \vdash v : T}{\Gamma \vdash s[S, \mathtt{o} : v \cdot \vec{h}] \triangleright s : !\langle T \rangle; M_{\mathtt{o}}\,[S]} \quad (\text{SndQ})$$

$$\frac{\Gamma \vdash s[S, \mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}}\,[S] \quad \Gamma \vdash v : T}{\Gamma \vdash s[S, \mathtt{i} : v \cdot \vec{h}] \triangleright s : ?(T); M_{\mathtt{i}}\,[S]} \quad (\text{RcvQ})$$

$$\frac{\Gamma \vdash s[S, \mathtt{o} : \vec{h}] \triangleright s : M_{\mathtt{o}}\,[S]}{\Gamma \vdash s[S, \mathtt{o} : l \cdot \vec{h}] \triangleright s : \oplus l; M_{\mathtt{o}}\,[S]} \quad (\text{SelQ}) \qquad \frac{\Gamma \vdash s[S, \mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}}\,[S]}{\Gamma \vdash s[S, \mathtt{i} : l \cdot \vec{h}] \triangleright s : \& l; M_{\mathtt{i}}\,[S]} \quad (\text{BraQ})$$

$$\frac{\Gamma \vdash s[S, \mathtt{o} : \vec{h}] \triangleright s' : S' \cdot s : M_{\mathtt{o}}\,[S]}{\Gamma \vdash s[S, \mathtt{o} : \vec{h} \cdot s'] \triangleright s : !\langle S' \rangle; M_{\mathtt{o}}\,[S]} \quad (\text{DelQ})$$

$$\frac{\Gamma \vdash s[S, \mathtt{i} : \vec{h}] \triangleright s : M_{\mathtt{i}}\,[S]}{\Gamma \vdash s[S, \mathtt{i} : s' \cdot \vec{h}] \triangleright s : ?(S'); M_{\mathtt{i}}\,[S] \cdot s' : S'} \quad (\text{SRcvQ})$$

$$\frac{\Gamma \vdash P \triangleright \Delta_1 \qquad \Gamma \vdash Q \triangleright \Delta_2}{\Gamma \vdash P \mid Q \triangleright \Delta_1 * \Delta_2} \quad (\text{QConc}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot s : S^e\,[S_1] \cdot \overline{s} : \overline{S^e}\,[S_2]}{\Gamma \vdash (\nu\, s)P \triangleright \Delta} \quad (\text{SRes})$$

$$\frac{\Gamma \vdash a[\vec{h}] \triangleright \Delta}{\Gamma \vdash a[\vec{h} \cdot s] \triangleright \Delta \cdot s : \emptyset\,[S]} \quad (\text{Buff}) \qquad \Gamma \vdash \overline{a}\langle s \rangle \triangleright s : \emptyset\,[S] \quad (\text{ReqM})$$

Figure 4.7: Extended typing rules for the ESP run-time processes.

Furthermore rule (SRes) is consisted with the S^e notation.

A notable fact is the distinction between process typing and runtime session typing in rule (SRes) and through the runtime session typing system. A process typing $s : S^e$ defines the type of a session channel taking information from processes and endpoints. Session runtime syntax $[S]$ defines the session type information from a non-endpoint process. This runtime information is used by the typing system to perform type match and choose a safe and sound process to handle a session event.

4.2.5 Subject Reduction

In this section we show that the ESP extension maintains the typing properties of the ASP.

Following Definition 3.2.3 we define the well-configured linear environment using the information for the session endpoint runtime type in the linear environment:

Definition 4.2.2 (Well-configured Linear Environments)**.** We say that Δ is *well configured* if whenever $\forall s \in \mathtt{dom}(\Delta)$, then either $\Delta(s) = S^e$ with $\Delta(\overline{s}) = \overline{S^e}$, or $\Delta(s) = S^e\,[S_1]$ with $\Delta(\overline{s}) = \overline{S^e}\,[S_2]$.

The linear environment reduction for the ESP corresponds to the linear environment for the ASP in § 3.2.5. Note that linear environment reduction is now expressed up-to session set subtyping.

Lemmas for:

- Weakening – Lemma 3.2.2
- Strengthening – Lemma 3.2.3
- Substitution – Lemma 3.2.4

continue to hold for the ESP type system.

We proceed with the theorems for the soundness and safety of the typing system.

Theorem 4.2.1 (Subject Congruence and Reduction)**.**

1. If $\Gamma \vdash P \triangleright \Delta$ and $P \equiv Q$, then $\Gamma \vdash Q \triangleright \Delta$.
2. If $\Gamma \vdash P \triangleright \Delta$ with Δ well-configured and $P \longrightarrow Q$, then we have $\Gamma \vdash Q \triangleright \Delta'$ such that $\Delta \longrightarrow^* \Delta'$ and Δ' is well-configured.

Proof. For proof, see Appendix A.2. □

We now prove communication safety. We extend the definition of an s-redex in Definition 3.2.5 to include `arrive` and `typecase` s-redexes:

Definition 4.2.3 (s-redex)**.** We say an *s-redex* is a parallel composition of two s-processes that has one of the following shapes:

(a) $s!\langle v\rangle;P \mid s[!\langle T\rangle;S]$ (b) $s \oplus l_i;P \mid s[\oplus\{l_j : S_j\}_{j\in J}]$ with $i \in J$

(c) $s?(x);P \mid s[?(T);S, \mathtt{i} : v \cdot \vec{h}]$ (d) $s\&\{l_i : P_i\}_{i\in I} \mid s[\&\{l_j : S_j\}_{j\in J}, \mathtt{i} : l_{i'} \cdot \vec{h}]$ with $i' \in J \subseteq I$

(e) $s[\mathtt{o} : v \cdot \vec{h}] \mid \overline{s}[\mathtt{i} : \vec{h}']$

(f) $E[\mathtt{arrive}\ s\ v] \mid s[?(U);S, \mathtt{i} : \vec{h}]$ with v of type U, and $\vec{h} = \varepsilon$ or $\vec{h} = v' \cdot \vec{h}'$, v' of type U

(g) $E[\mathtt{arrive}\ s\ l_i] \mid s[\&\{l_j : S_j\}_{j\in J}, \mathtt{i} : \vec{h}]$ with $i \in J$, and $\vec{h} = \varepsilon$ or $\vec{h} = l_{i'} \cdot \vec{h}'$, $l_{i'} \in J$

(h) $\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i : S_i) : P_i\} \mid s[S]$ with $\exists i \in I.\ S_i \leq S$

All redexes require the immediate action to correspond with the active type prefix in the local configuration. Cases (f–h) are for the new primitives for asynchronous event handling.

A process P is an *error* if up-to structural congruence (following [HYC08, § 5]), P contains two s-processes which do not form an s-redex, or an expression in P contains a type error in the standard sense. As a corollary of subject reduction (Theorem 4.2.1), we obtain:

Theorem 4.2.2 (Communication and Event-Handling Safety)**.** If P is a well-typed program, then $\Gamma \vdash P \triangleright \emptyset$, and P never reduces to an error.

Proof. See Appendix A.2 for details. □

4.3 Eventful Session Bisimulation and its Properties

Following Section 3.3 we define the behavioural theory for Eventful Session π-calculus. The definitions for the behavioral theory coincide with the definitions in Section 3.3. For this purpose we list the definitions and results with the appropriate comments whenever needed.

4.3.1 Labelled Transition Semantics

The labelled transition system is defined on the labels defined in Definition 3.3.1, together with the definition for free and bound label names (§ 3.3.1) and Definition 3.1 for the label duality. For contexts we use the definitions for contexts in the ASP (Definition 3.3.2).

Untyped Labelled Transition System. Figure 4.8 gives the untyped label transition system (LTS). Note that in contrast with the LTS in Figure 3.9, session endpoints are defined with the runtime session type. Furthermore, observable transition does not result in the transition of the runtime session type in queues. Finally, the transition rules for the `arrive` and the `typecase` constructs are subsumed by rule $\langle$Local$\rangle$.

Localisation and Typed Labelled Transition System. We define the localisation property for ESP processes, based on the localisation Definition 4.3.1 for the ASP. In this definition we take a slightly different approach, for checking endpoint configuration presence, based on the fact that a linear session environment records the session runtime syntax.

Definition 4.3.1 (Localisation)**.** Let P be closed and $\Gamma \vdash P \triangleright \Delta$. Then we say $\Gamma \vdash P \triangleright \Delta$ is *localised* if:

(1) For each $s \in \mathtt{dom}(\Delta)$, $s : S[S'] \in \Delta$ and (2) If $\Gamma(a) = \mathtt{i}\langle S \rangle$, then $a \in \Delta$.

We exploit the fact that the presence of a session type runtime in the linear environment, denotes the presence of a session endpoint configuration. We impose that a session endpoint

$$\langle \text{Acc} \rangle \quad a[\vec{s}] \xrightarrow{a\langle s \rangle} a[\vec{s} \cdot s] \qquad \langle \text{Req} \rangle \quad \overline{a}\langle s \rangle \xrightarrow{\overline{a}\langle s \rangle} \mathbf{0}$$

$$\langle \text{In} \rangle \quad s[S, \mathtt{i} : \vec{h}] \xrightarrow{s?\langle v \rangle} s[S, \mathtt{i} : \vec{h} \cdot v] \qquad \langle \text{Out} \rangle \quad s[S, \mathtt{o} : v \cdot \vec{h}] \xrightarrow{s!\langle v \rangle} s[S, \mathtt{o} : \vec{h}]$$

$$\langle \text{Bra} \rangle \quad s[S, \mathtt{i} : \vec{h}] \xrightarrow{s\&l} s[S, \mathtt{i} : \vec{h} \cdot l] \qquad \langle \text{Sel} \rangle \quad s[S, \mathtt{o} : l \cdot \vec{h}] \xrightarrow{s \oplus l} s[S, \mathtt{o} : h]$$

$$\langle \text{Local} \rangle \quad \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \qquad \langle \text{Tau} \rangle \quad \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\, \mathtt{bn}(\ell, \ell'))(P' \mid Q')}$$

$$\langle \text{Par}_L \rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \langle \text{Par}_R \rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{Q \mid P \xrightarrow{\ell} Q \mid P'}$$

$$\langle \text{Res} \rangle \quad \frac{P \xrightarrow{\ell} P' \quad n \notin \mathtt{fn}(\ell)}{(\nu\, n)P \xrightarrow{\ell} (\nu\, n)P'} \qquad \langle \text{OpenS} \rangle \quad \frac{P \xrightarrow{\overline{a}\langle s \rangle} P'}{(\nu\, s)P \xrightarrow{\overline{a}(s)} P'}$$

$$\langle \text{OpenN} \rangle \quad \frac{P \xrightarrow{s!\langle a \rangle} P'}{(\nu\, a)P \xrightarrow{s!(a)} P'} \qquad \langle \text{Alpha} \rangle \quad \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Figure 4.8: Labelled transition system.

configuration should be present for every free session name. Bound session names are implicitly checked by the fact that a localised process is typable, i.e. typing rule [SRes] was used, to check localisation of bound session names.

The labelled transition system for environment is essentially the labelled transition system define in Figure 3.10 and described in § 3.3. Note that the LTS for ESP environment requires to carry the session runtime type in its definition, but Figure 3.10 is still valid for its definition, since the session runtime is not changed in the case of observable environment actions.

We use the typed transition definition for the ASP in Definition 3.3.5, to define the typed transition for the ESP.

As the ASP definition we extend the typed transition to: $\Longrightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$, $\overset{\ell}{\Longrightarrow}$ for the composition $\Longrightarrow\xrightarrow{\ell}\Longrightarrow$ and $\overset{\hat{\ell}}{\Longrightarrow}$ for $\Longrightarrow$ if $\ell = \tau$ and $\overset{\ell}{\Longrightarrow}$ otherwise. Furthermore we write $\xrightarrow{\hat{\ell}}$ for $\longrightarrow$ if $\ell = \tau$ and $\xrightarrow{\ell}$ otherwise.

4.3.2 Bisimulation

The symmetric and transitive closure of $\longrightarrow$ over linear environment is denoted as in Definition 3.3.2 using the symbol $\rightleftharpoons$. We assume that the typed relation Definition 3.3.6 holds for the ESP processes. We introduce typed barbs for the ESP using Definition 3.3.7 from the ASP calculus.

We explicitly define the Reduction Congruence relation, which is essentially the same with the reduction congruence Definition 3.3.8 for the ASP.

Definition 4.3.2 (Reduction Congruence)**.** A typed relation $\mathscr{R}$ is *reduction congruence* if it is a congruence and satisfies the following condition: for each $\Gamma \vdash P_1 \triangleright \Delta_1 \; \mathscr{R} \; P_2 \triangleright \Delta_2$ whenever $\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2$ are localised then:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow n$ iff $\Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow n$.

2. Whenever

   - $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ holds, $P_1 \twoheadrightarrow P_1'$ implies $P_2 \twoheadrightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \rightleftharpoons \Delta_2'$.
   - The symmetric case.

The maximum reduction congruence [HY95], is denoted by $\cong$.

We explicitly define ESP Asynchronous Session Bisimulation, identical with bisimulation (Definition 3.3.9) for the ASP.

Definition 4.3.3 (Asynchronous Session Bisimulation)**.** A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, it holds:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\hat{\ell}}{\Longrightarrow} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathscr{R} P_2' \triangleright \Delta_2'$ with $\Delta_1' \rightleftharpoons \Delta_2'$ holds and
2. the symmetric case.

The maximum bisimulation exists which we call *bisimilarity*, denoted by $\approx$. We sometimes leave environments implicit, writing e.g. $P \approx Q$.

We extend $\approx$ to possibly non-localised closed terms by relating them when their minimal localisations are related by $\approx$ (given $\Gamma \vdash P \triangleright \Delta$, its *minimal localisation* adds empty queues to P for the input shared channels in Γ and session channels in Δ that are missing their queues). Further $\approx$ is extended to open terms in the standard way [HY95].

4.3.3 Properties of Asynchronous Session Bisimilarity

This subsection studies central properties of eventful session semantics. The results for the Asynchronous Session π-calculus in Section 3.3.3 continue to hold for the Eventful Session π-calculus.

Characterisation of reduction congruence. Bisimilarity coincides with the naturally defined reduction-closed congruence [HY95].

Theorem 4.3.1 (Soundness and Completeness)**.** $\approx = \cong$.

Proof. To proof is done following the structure of the proof for Theorem 3.3.1 and extending to the cases for the construct of `arrive` and `typecase`. Note that the presence of session runtime typing does not affect the proof method used. Appendix A.3.1 gives the details. □

Asynchrony, Session Determinacy and Confluence. We study the properties of our asynchronous session bisimulations based on the notions of [PW97]. The results from § 3.3.3 for the ASP are preserved in the ESP extension. Definitions for output/input actions, determinacy and confluence are in § 3.3.3.

Lemma 4.3.1 (Input and Output Asynchrony)**.** Suppose $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

- *(input advance)* If ℓ is an input action, then $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\longrightarrow\!\!\Longrightarrow} P' \triangleright \Delta'$.
- *(output delay)* If ℓ is an output action, then $\Gamma \vdash P \triangleright \Delta \Longrightarrow\!\!\stackrel{\ell}{\longrightarrow} P' \triangleright \Delta'$.

Proof. For proof, see Appendix A.4.1. □

The notion of session determinacy for ASP (§ 3.3.13) is now defined as:

Definition 4.3.4 (Session Determinacy)**.** Let us write $P \xrightarrow{\ell}_s Q$ if $P \xrightarrow{\ell} Q$ where if $\ell = \tau$ then it is generated without using [Request1], [Request2], [Accept], [Arrive-req], [Arrive-sses] nor [Arrive-msg] in Figure 4.3 (i.e. a communication is performed without arrival predicates or session initiation actions). We extend the definition to $\overset{\vec{\ell}}{\Longrightarrow}_s$ and $\overset{\hat{\ell}}{\Longrightarrow}_s$ etc. We say P is *session determinate* if P is typable and localised and if $\Gamma \vdash P \triangleright \Delta \overset{\vec{\ell}}{\Longrightarrow} Q \triangleright \Delta'$ then $\Gamma \vdash P \triangleright \Delta \overset{\vec{\ell}}{\Longrightarrow}_s Q \triangleright \Delta'$. We call such Q a *session derivative* of P.

In the eventful extension of ASP, the arrive predicate reduction breaks the confluence and determinacy properties, while the typecase reduction preserves them. For a discussion and an example demonstrating the lack of confluence see Example 3 in § 5.1 of this thesis. The above definition for session determinacy transfers this fact to the lemmas that follow. The proof cases of the following lemmas are distinguished from the proves in § 3.3.3 with the add of the `typecase` case. For a discussion and intuitions around the next results, see § 3.3.3.

Lemma 4.3.2. Let P be session determinate and $\Gamma \vdash P \triangleright \Delta \Longrightarrow Q \triangleright \Delta'$. Then $P \approx Q$.

Proof. For proof, see Appendix A.4.2 □

Lemma 4.3.3. Assume typable, localised P and actions ℓ_1, ℓ_2 such that $\mathtt{subj}(\ell_1), \mathtt{subj}(\ell_2)$ are session names and $\ell_1 \bowtie \ell_2$. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_2} P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta \xrightarrow{\ell_2 \lfloor \ell_1} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta \xrightarrow{\ell_1 \lfloor \ell_2} P' \triangleright \Delta'$

Proof. For proof, see Appendix A.4.3. □

Lemma 4.3.4. Let P be session determinate. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} P' \triangleright \Delta'$ and $\Gamma \vdash P \triangleright \Delta \overset{\hat{\ell}}{\Longrightarrow} P'' \triangleright \Delta''$ then $P' \approx P''$

Proof. For proof, see Appendix A.4.4. □

Lemma 4.3.5. Let P be session determinate and $\ell_1 \bowtie \ell_2$. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \overset{\ell_2}{\Longrightarrow} P_2 \triangleright \Delta_2$, then $\Gamma \vdash P_1 \triangleright \Delta_1 \overset{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta_2 \overset{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} P'' \triangleright \Delta''$ and $P' \approx P''$

Proof. For proof, see Appendix A.4.5. □

Theorem 4.3.2 (Session Determinacy)**.** Let P be session determinate. Then P is determinate and confluent.

Proof. From the definition of confluence (resp. determinacy) and from the definition of P we have that each derivative Q of P is also session determinate. The proof is an immediate result of Lemma 4.3.5 (resp. Lemma 4.3.4). □

The following relation is used to prove the event-based optimisation.

Definition 4.3.5 (Determinate Up-to expansion Relation)**.** Let $\mathscr{R}$ be a symmetric, typed relation such that if $\Gamma \vdash P \triangleright \Delta \mathscr{R} Q \triangleright \Delta$, then if

1. P, Q are determinate;
2. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \Gamma' \vdash P'' \triangleright \Delta''$ then $\Gamma \vdash Q \triangleright \Delta \stackrel{\ell}{\Longrightarrow} \Gamma' \vdash Q' \triangleright \Delta'$ and $\Gamma' \vdash P'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash P' \triangleright \Delta'$ with $\Gamma' \vdash P' \triangleright \Delta' \mathscr{R} Q' \triangleright \Delta'$;
3. the symmetric case.

Then we call $\mathscr{R}$ a *determinate up-to expansion relation*, or often simply *up-to expansion relation*.

Lemma 4.3.6. Let $\mathscr{R}$ be an up-to expansion relation. Then $\mathscr{R} \subset \approx$.

Proof. The proof is easy by showing $\Longrightarrow \mathscr{R} \Longleftarrow$ is a bisimulation. Denote this relation as $\mathscr{S}$. We can easily check that $\mathscr{S}$ is a bisimulation, using determinacy (commutativity with other actions). □

Chapter 5

Applications of the Eventful Behavioural Theory

In this Chapter we demonstrate the applicability of the Eventful Session π-calculus behavioural theory. In the first section we give core examples that give the first insights for the order-preserving and non-blocking properties of the ESP. We also demonstrate the basic properties of the `arrive`-predicate. In the second section of this Chapter, we show the differentiation of our bisimulation equivalence with other well known bisimulation equivalences for other well known π-calculi. We proceed with the ESP encoding of the *selector*, which is a basic event handling operator. The selector is used to construct an event-loop, where we use the bisimulation and confluence theory to study the properties of the event-loop. The last section uses the event-loop to study a transform from a threaded program to an event-driven program. The transformation is based on the work by Lauer and Needham [LN79], where they study the duality of the threaded and event-based approaches. Our transformation is proven to be typed and semantic preserving.

5.1 Properties of the ESP Behavioural Theory

This section discusses the basic properties of the behavioural theory developed in § 4.3. We use examples to examine the nature of non-blocking and order-preserving properties of the asynchronous bisimulation. A process is characterised as non-blocking if the process prefix does not block the execution of the process. In other words we can observe an action on the process other than the action expected to obesrve on the prefix of the process. The order-preserving property requires that messages are received in the same order they are being sent. This is a main property that should hold for communication on session names. We show by example the non-confluence property of the `arrive` operation, i.e. the `arrive` construct breaks the confluence on session transitions. The last example gives an equivalence on a recursive process that uses input asynchrony gives a very basic intuition about the structure of event-driven programs.

In this Section, let: $R_i = s_i[\mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}'_i]$.

1. Input and output permutation. Two actions at different session names are permutable up to $\approx$, if they are both in input or both in output mode:

$$s_1?(x);s_2?(y);P \mid R_1 \mid R_2 \quad \approx \quad s_2?(y);s_1?(x);P \mid R_1 \mid R_2$$
$$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid R_1 \mid R_2 \quad \approx \quad s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid R_1 \mid R_2$$

Permutability shows that actions on different session names are non-blocking and asynchronous. We expect a natural permutation on different session channels in the presence of asynchrony. The fact that communication is fine-grained with the existence of both an input and an output buffer on session endpoint configurations, allows for permutation of both input and output actions.

Note that an input and an output action on different sessions cannot generally be permuted:

$$s_1?(x);s_2!\langle v\rangle;P \mid R_1 \mid R_2 \not\approx s_2!\langle v\rangle;s_1?(x);P \mid R_1 \mid R_2$$

2. Input and output ordering. In contrast to actions on different session names, two actions on the same session name cannot be permuted:

$$\begin{aligned} s_1?(x);s_1?(y);P \mid R_1 &\not\approx s_1?(y);s_1?(x);P \mid R_1 \\ s_1!\langle v\rangle;s_1!\langle w\rangle;P \mid R_1 &\not\approx s_1!\langle w\rangle;s_1!\langle v\rangle;P \mid R_1 \end{aligned}$$

Non-permutability on the same session name shows the order-preserving property inside a session. This result is expected since it is part of session types principle to enforce an action sequence inside a session. Following this conclusion, it also holds that:

$$s_1?(x);s_1!\langle v\rangle;P \mid R_1 \not\approx s_1!\langle v\rangle;s_1?(x);P \mid R_1$$

3. Arrival predicates. Let $P_1 \not\approx P_2$. If the syntax of ESP does not include the `arrive` predicate then:

$$\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v] \approx \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid s[\texttt{i} : v] \mid \overline{s}[\texttt{o} : \varepsilon]$$

In the presence of `arrive` s, the bisimulation does not hold anymore.

$$\begin{aligned} &\texttt{if arrive}\ s\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v] \\ &\not\approx \\ &\texttt{if arrive}\ s\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid s[\texttt{i} : v] \mid \overline{s}[\texttt{o} : \varepsilon] \end{aligned}$$

This is because

$$\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2 \mid s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v] \longrightarrow P_2$$

but

$$\texttt{if arrive } s \texttt{ then } P_1 \texttt{ else } P_2 \mid s[\texttt{i} : v] \mid \overline{s}[\texttt{o} : \varepsilon] \not\longrightarrow P_2$$

The above result is important when designing and reasoning about systems that handle process control by inspecting the arrival of messages.

As a direct consequence is the lack of confluence in the presence of the `arrive` predicate if P_1, P_2 are confluent. To show confluence in a structure similar to the above, it is required to show that P_1, P_2 are confluent and $P_1 \approx P_2$. Note that

$$\texttt{if arrive } s \texttt{ then } P \texttt{ else } P \mid s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v] \approx P \mid s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v]$$

Again this is expected as in this case the `if /else` construct is considered in some sense redundant by the bisimularity relation.

4. Arrive inspection ordering. A typical event-driven programming scenario requires the sequential `arrive`-inspection of messages inside a loop. In this example we demonstrate the simplest such module as a recursive sequence of `arrive` inspections over session configurations.

$$\begin{aligned} P_1 &= \texttt{if arrive } s_1 \texttt{ then } s_1?(x); P_2 \texttt{ else if arrive } s_2 \texttt{ then } s_2?(x); P_1 \texttt{ else } P_1 \\ P_2 &= \texttt{if arrive } s_2 \texttt{ then } s_2?(x); P_1 \texttt{ else if arrive } s_1 \texttt{ then } s_1?(x); P_2 \texttt{ else } P_2 \end{aligned}$$

Both processes recurse on the inspection of sessions s_1 and s_2. We can show by using an *up-to expansion relation* (Lemma 4.3.6) that $P_1 \mid R_1 \mid R_2 \approx P_2 \mid R_1 \mid R_2$. This result is used in Section 5.3 to verify properties of the selector constructs.

5.2 Comparisons with Asynchronous and Synchronous π-calculi

There is a behavioural differentiation between ESP and other well-known π-calculi. We compare the non-blocking and order-preserving properties and the semantic effect of the `arrive`-predicate, between the most studied synchronous and asynchronous bisimulations for the π-calculus.

In this section we clarify the relationship between:

1. The asynchronous bisimulation $\approx_a$ for the session-typed asynchronous π-calculus without queues, defined based on the semantics proposed in [HT91a] (Honda and Tokoro introduce a labelled transition system where we can always observe an input action).

2. The synchronous bisimulation $\approx_s$ for the session-typed synchronous π-calculus without queues (cf. [THK94, HVK98]).

3. The asynchronous bisimulation $\approx_2$ for the asynchronous session π-calculus with input-queue endpoint configuration. The bisimilarity relation $\approx_2$ is based on the systems presented in [GV10, CDCY07, MY09] for the asynchronous session types that are defined using two endpoint configurations for each session channel, without distinction between input and output entries in queues. We briefly introduce below these semantics, which we call *non-local* since the output directly puts the output message in the input queue.

$$\langle \mathrm{Acc}_A \rangle \quad a[\vec{s}] \xrightarrow{a\langle s \rangle} a[\vec{s} \cdot s] \qquad \langle \mathrm{Req}_A \rangle \quad \overline{a}\langle s \rangle \xrightarrow{\overline{a}\langle s \rangle} \mathbf{0}$$

$$\langle \mathrm{In}_A \rangle \quad s[\vec{h}] \xrightarrow{s?\langle v \rangle} s[\vec{h} \cdot v] \qquad \langle \mathrm{Out}_A \rangle \quad s!\langle v \rangle; P \xrightarrow{s!\langle v \rangle} P$$

$$\langle \mathrm{Bra}_A \rangle \quad s[\vec{h}] \xrightarrow{s\&l} s[\vec{h} \cdot l] \qquad \langle \mathrm{Sel}_A \rangle \quad s \oplus l; P \xrightarrow{s \oplus l} P$$

$$\langle \mathrm{Local}_A \rangle \quad \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \qquad \langle \mathrm{Par}_A \rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$\langle \mathrm{Tau}_A \rangle \quad \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\, \mathtt{bn}(\ell, \ell'))(P' \mid Q')} \qquad \langle \mathrm{Res}_A \rangle \quad \frac{P \xrightarrow{\ell} P' \quad n \notin \mathtt{fn}(\ell)}{(\nu\, n)P \xrightarrow{\ell} (\nu\, n)P'}$$

$$\langle \mathrm{OpenS}_A \rangle \quad \frac{P \xrightarrow{\overline{a}\langle s \rangle} P'}{(\nu\, a)P \xrightarrow{\overline{a}(s)} P'} \qquad \langle \mathrm{OpenN}_A \rangle \frac{P \xrightarrow{\overline{s}\langle a \rangle} P'}{(\nu\, a)P \xrightarrow{\overline{s}(a)} P'}$$

$$\langle \mathrm{Alpha}_A \rangle \quad \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Figure 5.1: Labelled Transition for Session Type System with Two Buffer Endpoint Without IO

4. The asynchronous session π-calculus with two end-point IO-queues $\approx$, i.e. the one developed in § 4.3.

The proof for the behavioural semantics comparison can be found in Appendix B.1.1.

In Figure 5.1 we define the labelled transition relation for the non-local semantics. We define the transition relation for the non-local semantics by replacing the output and selection rules in Figure 3.9 with rules:

$$\langle \mathrm{Out} \rangle \quad s!\langle v \rangle; P \xrightarrow{s!\langle v \rangle} P \qquad \langle \mathrm{Sel} \rangle \quad s \oplus l; P \xrightarrow{s \oplus l} P$$

The observation of the output actions ($s!\langle v \rangle$, $s \oplus l$) happens on the transition of the output prefix for processes. On the dual side, the observation of the input actions happens on session endpoint configurations, which are called input queues.

We use the non-local, untyped labelled transition system together with the environment transition relation (Figure 3.10) to define the non-local, typed transition relation.

Definition 5.2.1 (Non-local Typed Transition)**.** We write $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \Gamma' \vdash P' \triangleright \Delta'$ if

1. $P \xrightarrow{\ell} P'$
2. $(\Gamma, \Delta) \xrightarrow{\ell} (\Gamma', \Delta')$

The bisimulation relation is then defined with respect to the non-local, typed transition relation.

Definition 5.2.2 (Non-local Asynchronous Bisimulation)**.** Let $\Gamma \vdash P_1 \triangleright \Delta_1 \mathrel{\mathscr{R}} \Gamma \vdash P_2 \triangleright \Delta_2$. $\mathscr{R}$ is a non-local asynchronous bisimulation if whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathrel{\mathscr{R}} \Gamma \vdash P_2 \triangleright \Delta_2$ then

1. If $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} P_1' \triangleright \Delta_1'$ then $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} P_2' \triangleright \Delta_2'$ and $\Gamma \vdash P_1' \triangleright \Delta_1' \mathrel{\mathscr{R}} \Gamma \vdash P_2' \triangleright \Delta_2'$
2. The symmetric case.

The largest bisimulation, denoted $\approx_2$, is called non-local asynchronous bisimilarity.

Figure 5.2 summarises the distinguishing examples. The first table compares the *non-blocking* property of the input/output actions on different channels. We say that a process is input (resp. output) non-blocking, if the permutation of input (resp. output) actions in the process maintains behaviour (i.e. the latter process is bisimilar with the former). Non-blocking input holds for the asynchronous π-calculus, the non-local session semantics and, as expected, for the ESP behavioural semantics. Non-blocking output holds only for the asynchronous π-calculus and the ESP bisimulation theory. Non-blocking output does not hold for non-local semantics due to the lack of output queue in the non-local session endpoint configuration.

In the second table, we explore the Input/Output Order-Preserving preserving property, which ensures that messages on the same channel will be received/delivered in the order they were

sent. In contrast to the non-blocking property, we require a non-bisimilar permutation of input (resp. output) actions to clarify the input (resp. output) order-preserving property. The order-preserving property does not hold for the input and the output case of the asynchronous π-calculus. All other calculi respect order-preserving in both cases.

The table in Figure 5.3 explains whether the cases in Lemma 4.3.1 (1) (input advance) or (2) (output delay) are satisfied or not. If not, we place a counterexample. Input advance and output delay cannot happen in the synchronous setting. It is interesting to observe that due to the absence of the output queue in the non-local semantics, the output advance property does not hold.

5.2.1 Synchronous and Asynchronous π-calculi in the presence of `arrive`

Another technical interest is the effects of the `arrive`-predicate on four calculi under study. In the cases of the synchronous and the asynchronous π-calculus we cannot define an `arrive`-predicate, since their definition does not include local buffers for message passing. We define two `arrive`-inspected calculi with local buffers that simulate the blocking and the order-preserving properties for the synchronous and the asynchronous π-calculi, respectively.

For the synchronous π-calculus, we require to have the blocking and the order-preserving properties for both input and output and for the asynchronous π-calculus we require to have the non-blocking and the non-order preserving properties for both input and output. In the context of the synchronous π-calculus, we cannot define the `arrive`-operator without a compromise of the non-blocking input property, due to the asynchronous nature of the `arrive`-operator on the input queue of an endpoint.

We represent an asynchronous version of the synchronous π-calculus with the definition of a buffer with size one. We clarify these intuitions with the syntax and the label transition semantics for the Synchronous-like π-calculus with `arrive`.

	Non-Blocking Input	Non-Blocking Output
$\approx_a$	$s_1?(x);s_2?(y);P \approx_a s_2?(y);s_1?(x);P$	$s_1!\langle v\rangle \mid s_2!\langle w\rangle \mid P \approx_a s_2!\langle w\rangle \mid s_1!\langle v\rangle \mid P$
$\approx_s$	$s_1?(x);s_2?(y);P \not\approx_s s_2?(y);s_1?(x);P$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \not\approx_s s_2!\langle w\rangle;s_1!\langle v\rangle;P$
$\approx_2$	$s_1?(x);s_2?(y);P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \approx_2$ $s_2?(y);s_1?(x);P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \not\approx_2$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$
$\approx$	$s_1?(x);s_2?(y);P \mid B_1 \mid B_2 \approx$ $s_2?(y);s_1?(x);P \mid B_1 \mid B_2$ $B_i = s_i[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid B_1 \mid B_1 \approx$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid B_1 \mid B_2$ $B_i = s_i[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

	Input Order-Preserving	Output Order-Preserving
$\approx_a$	$s?(x);s?(y);P \approx_a s?(y);s?(x);P$	$s!\langle v\rangle \mid s!\langle w\rangle \mid P \approx_a s!\langle w\rangle \mid s!\langle v\rangle \mid P$
$\approx_s$	$s?(x);s?(y);P \not\approx_s s?(y);s?(x);P$	$s!\langle v\rangle;s!\langle w\rangle;P \not\approx_s s!\langle w\rangle;s!\langle v\rangle;P$
$\approx_2$	$s?(x);s?(y);P \mid s[\varepsilon] \not\approx_2$ $s?(y);s?(x);P \mid s[\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid s[\varepsilon] \not\approx_2$ $s!\langle w\rangle;s!\langle v\rangle;P \mid s[\varepsilon]$
$\approx$	$s?(x);s?(y);P \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx$ $s?(x);s?(y);P \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx$ $s!\langle w\rangle;s!\langle v\rangle;P \mid s[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

Figure 5.2: Comparisons between bisimulations in the asynchronous and the synchronous π-calculi.

	Lemma 3.3.1 (1)	Lemma 3.3.1 (2)
$\approx_a$	yes	yes
$\approx_s$	$(\nu\ s)(s!\langle v\rangle;s'?(x);\mathbf{0} \mid s?(x);\mathbf{0})$	$(\nu\ s)(s!\langle v\rangle;s'!\langle v'\rangle;\mathbf{0} \mid s'?(x);\mathbf{0})$
$\approx_2$	yes	$s!\langle v\rangle;s'?(x);\mathbf{0} \mid s'[v']$
$\approx\!\!\!\!\approx_a$	yes	yes

Figure 5.3: Comparison between the synchronous and the asynchronous π-calculi for Lemma 4.3.1

Syntax of the Synchronous-like π-Calculus with `arrive`:

$$P \quad ::= \quad \mathbf{0} \mid a[\varepsilon] \mid a(x).P \mid \overline{a}\langle v\rangle.P \mid P \mid P$$
$$\mid \quad (\nu\ a)P \mid\ !P \mid \texttt{if arrive } a \texttt{ then } P \texttt{ else } P$$

The syntax for the synchronous-like π-calculus with `arrive` extends the standard synchronous π-calculus syntax with a name configuration buffer $a[\varepsilon]$ and the `arrive` expression.

Label Transition Semantics of the Synchronous-like π-Calculus with `arrive`:

$$\overline{a}(v).P \xrightarrow{\overline{a}\langle v\rangle} P \qquad a(x).P \mid a[\varepsilon] \xrightarrow{a\langle v\rangle} a(x).P \mid a[v]$$

$$a(x).P \mid a[v] \xrightarrow{\tau} P\{v/x\} \qquad \frac{P \xrightarrow{\ell} P',\ \mathtt{fn}(\ell) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$\frac{P \xrightarrow{\ell} P',\ Q \xrightarrow{\ell'} Q',\ \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\ \mathtt{bn}(\ell,\ell'))(P' \mid Q')} \qquad \frac{P \xrightarrow{\ell} P'\ n \notin \mathtt{fn}(\ell)}{(\mathtt{new}\ n)P \xrightarrow{\ell} (\mathtt{new}\ n)P'}$$

$$\frac{P \xrightarrow{\overline{a}\langle v\rangle} P'}{(\mathtt{new}\ a)P \xrightarrow{\overline{a}(v)} P'} \qquad \frac{P \equiv_\alpha P'\ P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

$$\mathtt{if\ arrive}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid a[\varepsilon] \xrightarrow{\tau} Q \mid a[\varepsilon]$$

$$\mathtt{if\ arrive}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid a[v] \xrightarrow{\tau} P \mid a[v]$$

The label transition semantics restrict the size of the name configuration to be at most one. This is defined in the rule:

$$a(x).P \mid a[\varepsilon] \xrightarrow{a\langle v\rangle} a(x).P \mid a[v]$$

where we further require the name a to exists as a subject in an input prefix, in order to restrict to a behaviour closer to the order-preserving property of the synchronous π-calculus (i.e. we only observe an input action $a\langle v\rangle$ if there exists an input prefix to consume the input message v). The LTS is completed with the definition of the semantics for the process:

$$\mathtt{if\ arrive}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q$$

The rest of the rules are standard π-calculus label transition rules.

To demonstrate the compromise made from the definition of the synchronous π-calculus to achieve the definition of the synchronous-like π-calculus, consider the input action in both

calculi:

$$a(x).P \xrightarrow{a?\langle v\rangle} P\{v/x\}$$
$$a(x).P \mid a[\varepsilon] \xrightarrow{a?\langle v\rangle}\longrightarrow P\{v/x\} \mid a[\varepsilon]$$

The first process demonstrates an input action on an input prefixed process in the classic synchronous π-calculus. The second process uses the semantics for the synchronous-like π-calculus, where we observe an asynchronous input action. First an input action puts a message v in the communication buffer and then a silent τ action receives and substitutes message v in the receiveing process. Our interest focuses on the fact that, between the two transitions we can apply an `arrive` -inspection that will return the boolean value true `tt`.

We move on to the asynchronous π-calculus with the `arrive` operator, which is easier to define using endpoint configurations. The main idea here is to have queues that use a random buffer policy for message exchange:

Syntax of the Asynchronous π-Calculus with `arrive`:

$$\begin{aligned} P \quad ::= \quad & \mathbf{0} \mid a[\varepsilon] \mid a(x).P \mid \overline{a}\langle v\rangle \mid P \mid P \\ \mid \quad & (\nu\, a)P \mid !P \mid \texttt{if arrive}\ a\ \texttt{then}\ P\ \texttt{else}\ P \end{aligned}$$

The syntax for the asynchronous π-calculus with `arrive`, shares the same syntax with the synchronous-like π-calculus system, with the exception of the send prefix that is defined with no continuation.

Label Transition Semantics for the Asynchronous π-Calculus with `arrive`:

$$\overline{a}\langle v\rangle \xrightarrow{\overline{a}\langle v\rangle} \mathbf{0} \qquad a[\vec{h}] \xrightarrow{a\langle h\rangle} a[\vec{h}\cdot h]$$

$$a?(x);P \mid a[\vec{h_1}\cdot h_i\cdot\vec{h_2}] \xrightarrow{\tau} P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}] \qquad \frac{P \xrightarrow{\ell} P' \ \mathtt{fn}(\ell)\cap\mathtt{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$\frac{P \xrightarrow{\ell} P' \ Q \xrightarrow{\ell'} Q' \ \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\ \mathtt{bn}(\ell,\ell'))(P' \mid Q')} \qquad \frac{P \xrightarrow{\overline{a}\langle v\rangle} P'}{(\mathtt{new}\ v)P \xrightarrow{\overline{a}(v)} P'}$$

$$\frac{P \xrightarrow{\ell} P' \ n \notin \mathtt{fn}(\ell)}{(\mathtt{new}\ n)P \xrightarrow{\ell} (\mathtt{new}\ n)P'} \qquad \frac{P \equiv_\alpha P' \ P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

$$\mathtt{if\ arrive}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid a[\varepsilon] \xrightarrow{\tau} Q \mid a[\varepsilon]$$

$$\mathtt{if\ arrive}\ a\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid a[h\cdot\vec{h}] \xrightarrow{\tau} P \mid a[\vec{h}]$$

The label transition semantics allow an infinite size name configuration buffer. The definition for rule:

$$a?(x);P \mid a[\vec{h_1}\cdot h_i\cdot\vec{h_2}] \xrightarrow{\tau} P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}]$$

uses a random policy to select a message to receive from the queue, which disallows the order preserving property in the system but keeps the non-blocking property as required by the asynchronous π-calculus. The rest of the label transition rules are standard π-calculus rules.

We define the `arrive`-inspected non-local semantics with the addition of the `arrive` predicate in the syntax of non-local semantics and the addition of `arrive` transition semantics in the label transition system for non-local semantics.

Figure 5.4 summarises the results between processes `arrive`-prefixed processes and conditional branch prefixed processes that do not use the `arrive`. We number the four calculi as: (1) the asynchronous π-calculus with `arrive`; (2) the synchronous-like π-calculus with `arrive`; (3) the local-semantics with `arrive`; and (4) the eventful session π-calculus. It is interesting to see that all of the calculi (1–4) separate the semantics between the two cases

	With `arrive`	Without `arrive`
(1)	$\texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle$ $\not\approx \texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[v]$	$\texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle$ $\approx \texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[v]$
(2)	$\texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle;\mathbf{0}$ $\not\approx \texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[v]$	$\texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle;\mathbf{0}$ $\approx \texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[v]$
(3)	$\texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle;\mathbf{0}$ $\not\approx \texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid s[v]$	$\texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[\varepsilon] \mid \overline{s}\langle v\rangle;\mathbf{0}$ $\approx \texttt{if } e \texttt{ then } P \texttt{ else } Q \mid s[v]$
(4)	$\texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid B_1$ $\not\approx \texttt{if arrive } s \texttt{ then } P \texttt{ else } Q \mid B_2$ $B_1 = s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v]$ $B_2 = s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v]$	$\texttt{if } e \texttt{ then } P \texttt{ else } Q \mid B_1$ $\approx \texttt{if } e \texttt{ then } P \texttt{ else } Q \mid B_2$ $B_1 = s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v]$ $B_2 = s[\texttt{i} : \varepsilon] \mid \overline{s}[\texttt{o} : v]$

Figure 5.4: Arrived message detection behaviour in asynchronous and synchronous calculi.

(i.e. with and without the `arrive` predicate). We can see that the IO queues provide non-blocking inputs and outputs, while preserving the input/output ordering, which distinguishes the present framework from other known semantics.

As a conclusion, we observe that the present semantic framework is closer to the asynchronous bisimulation (1) $\approx_a$ augmented with order-preserving nature per session. Its key properties arise from local, buffered session semantics and typing. We have also seen the semantic significance of the `arrive` predicates, which enables processes to observe the effects of fine-grained synchronisations.

5.3 Representing High-level Event Constructs in ESP

A study of the event-driven paradigm identifies the different *selector* constructs, among the known high-level programming facilities (including those realised by libraries) used for event-based programming. The functionality of the selector construct offers a powerful event primitive which is the key for programming many event-based applications and other high-level event-based programming libraries. In the context of operating systems the selector construct is reffered to as polling, which is the operation for waiting on the status of a set of input/output devices until one of them is ready. A selector operation is introduced in the Java NIO package[1] [Lea03, SMI11] and it is used for building high performance concurrent applications. In brief, the selector component implements a mechanism that inspects a set of communication (input) channels for the arrival of messages. If a message is present, it is dispatched together with its channel for processing.

The functionality of the selector is used to build a core event-driven programming routine, called the event-loop. An event-loop waits on a select operation for the dispatching of ready to be processed events. When an event gets dispatched, its type is identified and the computation

[1] Java NIO stands for for Java new input/output, which is a package that performs I/O operations based on intermediate buffers and asynchrony.

proceeds with its processing accordingly. After the event-processing the event-loop routine recurses until the next event is ready.

This section gives the high-level semantics for a typesafe selector, shows its type-safe and semantic preserving encoding into ESP , and studies the behavioural properties of the selector based on the encoding, which will be directly used for the application in § 5.5.

5.3.1 A Basic Event Loop

Before we proceed with the semantic definition of the selector construct, we demonstrate the operation of the selector with the use of a simple example of event-driven session programming. The example comes from the context of web-services, where we define a multithreaded server and then its equivalent event-based server to constrast and understand event-driven concurrency. The event-driven server demonstrates the use of the selector construct as a basic function for the event loop routine.

Consider the following multithreaded session server.

$$\mu X.a(x:S).(x?(y_1);x?(y_2);x!\langle v'\rangle;\mathbf{0} \mid X) \;\mid\; \Pi_{i=1}^{n}\overline{a}(x:\overline{S}).x!\langle v_1\rangle;x!\langle v_2\rangle;x?(z);\mathbf{0}$$

The server process (on the left), listening (accepting sessions) on shared channel a, can unfold the required number of parallel "threads" (processes) to handle n client processes (on the right) concurrently. The annotation S, which declares the communication protocol (i.e. session type) that the server follows may be, e.g., $?(U_1);?(U_1);!\langle U_2\rangle;\emptyset$, which declares the reception of a sequence of two messages of type U_1 before sending a value v' of type U_2 in reply.

We give an event-driven server with the same capability to handle concurrent clients according to type S, but without needing to fork (create) a new thread for client. The event-driven server comprises a *single* thread, implementing the event loop routine, independent of the number

of clients. We make use of a few high-level macros to focus on the key concepts. The central notion is the *event selector*, henceforth referred to as just *selector* for short. In this example a selector offers two main functions. One is to store (*register*) session channels, which the selector then monitors for event occurrences, i.e. message arrivals. The other is to retrieve (*select*) a stored session channel at which a message has arrived and is ready for reading.

$$
\begin{array}{l}
\texttt{new sel } r \texttt{ in register } s_1 \texttt{ to } r \texttt{ in } \ldots \texttt{register } s_n \texttt{ to } r \texttt{ in} \\
\mu X.\texttt{select } x \texttt{ from } r \texttt{ in} \\
\quad \texttt{if } x = a \texttt{ then} \\
\quad\quad a(x:S).x?(y_1);x?(y_2);x!\langle v'\rangle;X \\
\quad \texttt{else} \\
\quad\quad \texttt{typecase } x \texttt{ of } \{ \\
\quad\quad\quad (x_1 :?(U_1);?(U_1);!\langle U_2\rangle;\texttt{end}) : x_1?(y_1);\texttt{register } x_1 \texttt{ to } r \texttt{ in } X, \\
\quad\quad\quad (x_2 :?(U_1);!\langle U_2\rangle;\texttt{end}) : \quad\quad x_2?(y_2);x_2!\langle v'\rangle;X \\
\quad\quad \}
\end{array}
$$

The process first creates a new selector `sel` with name r. It then registers sessions $s_1, \ldots, s_n$ to the selector, as with n client connections of the multithreaded examples. In this example for brevity and for understanding we define a static event loop by registering a fixed number of already established sessions in contrast with the multithreaded example which dynamicaly accepts new sessions during the computation. A full definition of a dynamic event loop will be discussed later in this section. After session registering, the control flow enters the main *event loop*. In each iteration, the server will select a session s_i that is enabled for reading (waiting until one satisfies this condition), remove it from the internal storage of the selector r and substitute s_i for x. The `typecase` tests the selected s_i against the specified session type cases. If it is a newly established session, s_i will have the type specified by the first case: the server will proceed by receiving the first U_1 message, then re-registering s_i back to the selector to await the arrival of the second message. Otherwise, s_i will correspond to the second case:

the server will receive the second U_1 message and send the U_2; this session is now completed and the server proceeds to the next iteration. In this way, session types are used to determine not only the type of the expected event, but also the point in the protocol at which the event is occurring, ensuring that the event is handled correctly. The key characteristic of the event-driven server is that, by only selecting sessions with arrived messages, the event loop can safely and efficiently interleave the handling of multiple, concurrent clients in a single thread because delayed message arrival in any one session does not block the execution of any other session.

5.3.2 Selector semantics

The core functionality of the selector, can be defined using three operations: *create* a new selector, *register* a channel with the selector, and *select* (i.e. retrieve from the selector) a channel on which a message has arrived. The syntax for the extended ESP, denoted ESP$^+$, is summarised as:

$$\begin{array}{lll} P & ::= & \vdots \\ & \mid & \texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \mid \texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \mid \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle\vec{s}\rangle \end{array}$$

A selector is represented as the process $r\langle\vec{s}\rangle$, where r is the name of the selector and $\vec{s}$ the registered channels in the selector queue. Construct $\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P$ is used to create a new selector on the bound name r. Selector r is used to register channels with type S. Note that S can be a session set type, allowing sessions with different types to be registered with the selector. The operation $\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P$ registers a session s in selector r and then continues with process P. The selection of a session channel with a non-empty i-configuration is done via process $\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P$, where variable x exists bounded in

process P. The operation of the selector is defined as a reduction relation below:

$$\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \longrightarrow (\texttt{new}\ r)(P \mid r\langle\varepsilon\rangle)$$

$$\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \mid r\langle\vec{s}\rangle \longrightarrow P \mid r\langle\vec{s}\cdot s\rangle$$

$$\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle s\cdot\vec{s}\rangle \mid s[\texttt{i} : \vec{h}] \longrightarrow P\{s/x\} \mid r\langle\vec{s}\rangle \mid s[\texttt{i} : \vec{h}] \qquad (\vec{h} \neq \varepsilon)$$

$$\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle s\cdot\vec{s}\rangle \mid s[\texttt{i} : \varepsilon] \longrightarrow \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle\vec{s}\cdot s\rangle \mid s[\texttt{i} : \varepsilon]$$

We introduce the structural rule $(\texttt{new}\ r)r\langle\varepsilon\rangle \cong \mathbf{0}$ for garbage collection.

Operator $\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P$, binds r in P and creates a new selector $r\langle\varepsilon\rangle$, named r, with session interaction type S. $\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P$ registers the session channel with r, adding s to the queue $\vec{s}$. The selector $\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P$ checks whether a message is available (i.e. an event has occurred) on the first session in the queue, s (note that x binds P). If so, it executes $P\{s/x\}$; otherwise, s is re-enqueued and the next session is tested.

5.3.3 From ESP $^+$ to ESP

The selector semantics of ESP $^+$ can be easily encoded in ESP by combining the *message arrival predicate* and recursions. We define the mapping from ESP $^+$ to ESP .

$$\begin{aligned}
[\![\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P]\!] &\stackrel{\texttt{def}}{=} (\nu\ b)(\overline{b}(x_{\overline{r}}).b(x_r).[\![P]\!] \mid b[\varepsilon]) \\
[\![\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P]\!] &\stackrel{\texttt{def}}{=} \overline{r}!\langle s\rangle;[\![P]\!] \\
[\![r\langle\vec{s}\cdot\vec{s}'\rangle]\!] &\stackrel{\texttt{def}}{=} \overline{r}[\texttt{o} : \vec{s}'] \mid r[\texttt{i} : \vec{s}] \\
[\![\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P]\!] &\stackrel{\texttt{def}}{=} \mu Select.r?(x);\texttt{if arrive}\ x\ \texttt{then}\ [\![P]\!]\ \texttt{else}\ \overline{r}!\langle x\rangle;Select
\end{aligned}$$

The mapping for other constructs is homomorphic. A selector is created using asynchronous session initiation, where the two configuration endpoints of an establish session encode the selector's queue configuration. The register operation is syntactic sugar for the delegation of a session to the dual session endpoint. The use of `arrive` is the key to avoiding blocked inputs in the select operation, allowing the selector to proceed asynchronously while handling any

available messages in the inspected session queues. The operations on the collection queue (via r and $\overline{r}$) exchange session channels, hence session delegation [HVK98] is essential.

Using the above selector encoding, the basic static event loop in Example 5.3.1, is encoded in ESP as:

$$
\begin{array}{l}
(\nu\ a)(\overline{a}(x_{\overline{r}}).a(x_r).x_{\overline{r}}!\langle s_1\rangle;\ldots x_{\overline{r}}!\langle s_n\rangle; \\
\quad \mu\, \mathit{Select}.x_r?(y);\ \texttt{if arrive}\ y\ \texttt{then} \\
\qquad\qquad \texttt{typecase}\ y\ \texttt{of}\ \{ \\
\qquad\qquad\quad (y_1 :?(U_1);?(U_1);?(U_2);\texttt{end}) : y_1?(y_1);x_{\overline{r}}!\langle y_1\rangle;\mathit{Select} \\
\qquad\qquad\quad (y_2 :?(U_1);!\langle U_2\rangle;\texttt{end}) : y_2?(y_2);y_2!\langle v'\rangle;\mathit{Select} \\
\qquad\qquad \} \\
\qquad\quad \texttt{else}\ x_{\overline{r}}!\langle y\rangle;\mathit{Select}) \mid a[\varepsilon])
\end{array}
$$

5.3.4 Typing Event Selectors

Typing selectors. Typing rules for the extended ESP selector construct naturally follow from the ESP -typing of the selector encoding. The type for a *user* of the selector is written $\overline{\texttt{sel}}\langle S\rangle$, and for the selector itself $\texttt{sel}\langle S\rangle$. For simplicity, we assume these types do not occur as part of other types. The linear environment Δ is extended with two additional type assignments, $r : \overline{\texttt{sel}}\langle S\rangle$ and $r : \texttt{sel}\langle S\rangle$, the latter only used for runtime typing for selector queues. The program typing rules for the selector operations are:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle}{\Gamma \vdash \texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \triangleright \Delta}\ [\text{Selector}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \quad S' \leq S}{\Gamma \vdash \texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \cdot s : S'}\ [\text{Reg}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \cdot x : S}{\Gamma \vdash \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle}\ [\text{Select}]$$

We define a mapping $[\![\Delta]\!]$ from the typing syntax of the selector to the ESP typing system,

where $[\![r : \overline{\texttt{sel}}\langle S\rangle]\!]$ is mapped as $r : S_r \cdot \bar{r} : \overline{S_r}$, when $S_r = \mu X.?(S);X$. All other mappings are homomorphic. The mapping for a selector construct is, as expected, a recursive session type, since a selector process is recursive. We write ESP $^+$ for the extension of ESP with selectors:

Proposition 5.3.1 (Soundness of Selector Typing Rules)**.**

1. (Type Preservation) $\Gamma \vdash P \triangleright \Delta$ in ESP $^+$ if and only if $\Gamma \vdash [\![P]\!] \triangleright [\![\Delta]\!]$.
2. (Soundness) $P \equiv P'$ implies $[\![P]\!] \equiv [\![P']\!]$; and $P \longrightarrow P'$ implies $[\![P]\!] \longrightarrow^* [\![P']\!]$.
3. (Safety) A typable process in ESP $^+$ never reduces to an error.

Proof. (1) is proved by typing the mapping from ESP $^+$ to ESP . A full proof can be found in Appendix B.2.1. (2) is straightforward. (3) is a corollary from (1, 2) and Theorems 4.2.1 and 4.2.2. □

The selector encoding demonstrates how the fine-grained typing rules of ESP can suggest and justify sound typing rules for high-level event handling constructs through ESP encodings.

5.4 Behavioural Properties of the Selector

This section investigates the basic properties for the event-loop, under the hypothesis that event handling processes are sequential and determinate. We can observe that if we arbitrarily permute the entries (session names) inside a selector queue, its behaviour remains the same with respect to the asynchronous session bisimilarity, $\approx$.

In the following definitions, we let $B_i = s_i[\texttt{i} : h_i, \texttt{o} : h_i']$. We also extend the process syntax to $R;Q$ where R is a sequential series of actions, used as a prefix. The *context definition* now allows $C[R]$ where R is replaced at the *hole* $(-)$ of the context.

Definition 5.4.1. Let

$$P_{Sel} = \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ \texttt{typecase}\ x\ \texttt{of}\ \{(x_i : S_i) : C[R_i]\}_{1 \leq i \leq m}$$

where

1. $C = -;\texttt{register}\ x\ \texttt{to}\ r\ \texttt{in}\ \mathit{Select}$, where Select is the recursive variable of the `select` construct (see the encoding of the selector in § 5.3.3)
2. $R_i\{s/x_i\}$ is a blocking prefixed, sequential series of actions and
3. $C[R_i\{s/x_i\}]$ is session determinate.

Then we define

$$\mathsf{Sel}_i^n = P_{Sel} \mid r\langle s_i, \ldots, s_k, s_{k+1}, \ldots, s_1, s_n, \ldots, s_{i-1}\rangle$$

and

$$\mathsf{PermSel}_i^n = P_{Sel} \mid r\langle s_i, \ldots, s_{k+1}, s_k, \ldots, s_1, s_n, \ldots, s_{i-1}\rangle$$

i.e. we permute two arbitrary entries in the selector queue in Sel_i^n to get a $\mathsf{PermSel}_i^n$.

We also write $\prod_i P_{1 \leq i \leq m}$ for $P_1 \mid P_2 \mid \cdots \mid P_m$.

Lemma 5.4.1. $\mathsf{Sel}_k^n \mid \prod_{1 \leq i \leq n} B_i \approx \mathsf{PermSel}_k^n \mid \prod_{1 \leq i \leq n} B_i$

Proof. We build a proof based on a similar idea explained in Example 4 in Section 5.1. For full details of the proof, see Appendix B.2.2. □

Next we extend the selector to the *dynamic selector*, where we allow the dynamic addition of session channels in the selector queue through a shared channel in order to capture the full functionality of the event-loop. The extension requires from the selector to periodically `arrive`-check a shared channel queue endpoint for new incoming session connections. To

achieve this, we extend, without loss of generality, the selector queue $r\langle v_1 \dots v_n \rangle$ to register tuples of the form $\vec{v}$, writting $r\langle \vec{v_1} \dots \vec{v_n} \rangle$. Then we register in the selecto queue tuples of values with either the form (s, shd) or the form (sd, a), both typed as $(S, \mathtt{io}\langle S' \rangle)$ (i.e. the first value is a session channel and the second value is a shared name). Values shd and sd represent a dummy shared channel and a dummy session channel, respectively. The dynamic selection, uses name matching to check whether the shared channel in the tuple is not a dummy (i.e. the shared channel in the tuple is the shared channel listening for new connections), and at the same time `arrive`-checks the shared channel for new sessions, otherwise it means that the tuple contains a session channel s (or sd) and proceeds as the static selector to `arrive`-check the s (or sd). For a session channel, we assume the corresponding endpoint of sd has empty type `end` and is always empty $\mathsf{sd}[\mathtt{i} : \varepsilon]$. Hence the expression `arrive` sd will be automatically evaluated to false (`ff`) in the case we `arrive`-check session sd. If the `arrive`-checks for the shared and the session name in the tuple fail the tuple is re-registered in the queue of the selector. We formally define the encoding from ESP to ESP $^+$ for the dynamic selector:

Definition 5.4.2 (Dynamic Selector Encoding)**.** We extend the register queue $r\langle v_1 \dots v_n \rangle$ to store tuples of the form $\vec{v}$, writting $r\langle \vec{v_1} \dots \vec{v_n} \rangle$. A dynamic selector is encoded as:

$$\begin{aligned}[[\mathtt{select}\ (x_s, x_a)\ \mathtt{from}\ r\ \mathtt{in}\ P]] \stackrel{\mathtt{def}}{=}\ & \mu Select.r?((x_s, x_a)); \mathtt{if}\ \mathtt{arrive}\ x_a\ \mathtt{and}\ x_a \neq \mathsf{shd} \\ & \mathtt{then}\ [[P]]\ \mathtt{else}\ \mathtt{if}\ \mathtt{arrive}\ x_s\ \mathtt{then}\ [[P]]\ \mathtt{else}\ \bar{r}!\langle (x_s, x_a) \rangle; Select\end{aligned}$$

It is straightforward to extend $[[P]]$ for other constructs and prove the same soundness properties as Proposition 5.3.1.

Definition 5.4.3 (Dynamic Selector)**.** We define

$$\begin{aligned}P_{DSel} \quad = \quad & \mathtt{select}\ (x_s, x_a)\ \mathtt{from}\ r\ \mathtt{in}\ \mathtt{if}\ x_a = a\ \mathtt{then}\ x_a(y).\mathtt{register}\ (y, \mathsf{shd})\ \mathtt{to}\ r\ \mathtt{in} \\ & \mathtt{register}\ (x_s, x_a)\ \mathtt{to}\ r\ \mathtt{in}\ X\ \mathtt{else}\ \mathtt{typecase}\ x_s\ \mathtt{of}\ \{(x_i : S_i) : C[R_i]\}_{1 \leq i \leq m}\end{aligned}$$

where

1. $C = -;\texttt{register}\ (x_s, x_a)\ \texttt{to}\ r\ \texttt{in}\ \mathit{Select}$, where Select is the recursive variable of the `select` construct (see the encoding of the dynamic selector in § 5.4.2)

2. $R_i\{s/x_i\}$ is a blocked prefixed and sequential series of actions.

3. $C[R_i]$ is session determinate.

Then we define

$$\mathsf{DSel}_i^n = P_{DSel} \mid r\langle v_i, \ldots, v_k, v_{k+1}, \ldots, v_1, v_n, \ldots, v_{i-1}\rangle$$

and

$$\mathsf{PermDSel}_i^n = P_{DSel} \mid r\langle v_i, \ldots, v_{k+1}, v_k, \ldots, v_1, v_n, \ldots, v_{i-1}\rangle$$

Lemma 5.4.2. $\mathsf{DSel}_k^n \mid \prod_{1\leq i\leq n} B_i \approx \mathsf{PermDSel}_k^n \mid \prod_{1\leq i\leq n} B_i$.

Proof. For the full proof, see Appendix B.2.2. □

Due to the fact that bisimulation is an equivalence relation, we can use Lemma 5.4.2 (and Lemma 5.4.1) to arbitrarily apply a sequence of permutations in the channels in a selector queue and maintain the process behaviour under the hypothesis of sequentiality and determinacy.

The permutation of confluent selectors is a very important result for reasoning and verifying event-loop applications, and is essential to understand the reactive nature of the event-driven programming paradigm (see the next section).

5.5 Lauer-Needham Transform

In an early work [LN79], Lauer and Needham observed that a concurrent program may be written equivalently either in a thread-based programming style or in an event-based style.

A thread-based programming style is defined on thread creation and shared memory primitives, in contrast to the event-based style that requires message passing and a single-threaded event loop that processes messages sequentially. Many studies follow the Lauer-Needham framework and use the selector primitive (cf. [BDM98, Lea03, SMI11]) for the event-based style to compare the two programming styles, often focusing on performance of server architectures (see § 2.3 and [HKP+10, § 6] for recent studies on event programming). These implementations implicitly or explicitly assume a *transformation* from a program written in the thread-based style, especially those which generate a new thread for each service request (as in some of the thread-based web servers), to its *equivalent* event-based program, which treats concurrency by using a single threaded event-loop (as in event-based web servers). However neither the precise semantic effects of such a transformation nor the exact meaning of the associated "equivalence" have ever been clarified.

In this section we study the semantic effects of such a transformation using the asynchronous session bisimulation. We follow [LN79] to introduce a formal mapping from a thread-based process to an event-loop process. We assume a multithreaded server process whose code creates fresh threads at each service invocation (session accept). The key idea to transform the multithreaded server to an event-driven server with the same behaviour, is to decompose its whole code into distinct smaller code segments, each handling the part of the original code starting from a blocking action. Such a blocking action is represented as reception of a message (input or branching). A single global event-loop uses the selector construct to inspect a set of session configuration buffers for message arrivals. We stipulate a class of processes which we consider for our translation. We set

$$*a(x).P = \mu X.a(x).(P|X)$$

to abbreviate an *input replication*.

5.5.1 Multithreaded Server Process

Definition 5.5.1 (Server)**.** A *simple server at a* is a closed process

$$*a(x).P$$

with a typing of form

$$a : \mathtt{i}\langle S\rangle, b_1 : \mathtt{o}\langle S_1\rangle, .., b_n : \mathtt{o}\langle S_n\rangle$$

where P is sequential (i.e. contains no parallel composition $|$), non-recursive and is determinate under any localisation. A simple server is often considered with its localisation with an empty queue $a[\varepsilon]$.

A simple server spawns an unbounded number of threads as it receives session requests repeatedly. Each thread may initiate other sessions with outside, and its interactions may involve delegations and name passing. Furthermore it is semantically ensured by determinacy that a server does *not* involve accesses to non-trivial mutable local state by threads. A practical example is a web-server interacting independently with a dynamic set of clients.

5.5.2 The Transform

In this subsection we define a transform from a simple server to an event-driven server. We begin by defining the programming constructs and operations used for defining the transform.

Preliminaries: We assume ESP extended with the following notions:

1. We model π-calculus communication on shared channels as the creation of session channels that implement a send/receive interaction.

$$\begin{aligned}[[\overline{o}\langle v\rangle.P]] &= \overline{o}(s:\mathtt{o}\langle ?(U);\mathtt{end}\rangle).s!\langle v\rangle;P\\ [[o(x).P]] &= o(x:\mathtt{i}\langle ?(U);\mathtt{end}\rangle).s?(x);P\end{aligned}$$

2. Polyadic Inputs/Outputs on session channels. Polydicity on session channels is straightforward since session channels have a linear usage, i.e. there are only two session endpoints for each channel.

   Mapping from ESP to polyadic ESP :

$$\begin{aligned}[[s!\langle v_1,\ldots,v_n\rangle;P]] &= s!\langle v_1\rangle;\ldots s!\langle v_n\rangle;[[P]]\\ [[s?(x_1,\ldots,x_n);P]] &= s?(x_1);\ldots s?(x_n);[[P]]\end{aligned}$$

3. Each client/session accepted is maintained in the server through its context. A context is expressed as a data closure structure. We define the syntax sugar to create and manipulate such a structure.

   Let y be a meta-value range over a list mapping between labels and values:

$$y \ ::= \ (l_i : v_i)_{i\in I}$$

   We define operations on meta-value y. Each operator is translated into an ESP $^+$ value:

$$\begin{aligned}\mathtt{val}((l_i:v_i)_{i\in I}) &= (v_i)_{i\in I} \qquad & \mathtt{new\ env}\ (l_i:v_i)_{i\in I}\ \mathtt{in}\ P &= (\nu\ v_1,v_2,\ldots)P\\ [[l_k]]_{(l_i:v_i)_{i\in I}} &= v_k & (l_i:v_i)_{i\in I}\{l_k\mapsto v\} &= (v_i)_{i\in I}\{v/v_k\}\end{aligned}$$

   Operations on y are ESP terms and are used for a list manipulation structure. $\mathtt{val}(y)$ returns the list of values $(v_i)_{i\in I}$ from the list mapping y. $\mathtt{new\ env}\ (l_i:v_i)_{i\in I}\ \mathtt{in}\ P$ creates a list of fresh names $(v_i)_{i\in I}$ restricted in P. Expression $[[l_k]]_y$ represents the value v_k of

y mapped by l_k. The $y\{l_k \mapsto v\}$ returns the list of values $(v_i)_{i\in I}$ from y with value v_k substituted by value v.

4. We say that a process P is *blocking* if it is input prefixed:

$$P \;::=\; a(x).P' \;\mid\; s?(x);P' \;\mid\; s\&\{l_i : P_i\}$$

We define the infix process operator $\preceq$ as a preorder relation over:

$$\begin{array}{lllllllll} P & \preceq & \overline{a}(x).P & \quad P & \preceq & a(x).P & \quad P & \preceq & s!\langle v\rangle;P \\ P & \preceq & s?(x);P & \quad P & \preceq & s\oplus l;P & \quad P_i & \preceq & s\&\{l_i : P_i\}_{i\in I} \end{array}$$

The blocking sub-terms of P are defined as

$$\mathsf{subterms}(P) = \{P_i \;\mid\; P_i \text{ blocking}, P_i \preceq P\}$$

with $i \leq j$ if $P_i \preceq P_j$

The Transform: We are now ready to define a transform from a simple server to an event-driven server in the terms of Lauer and Needham [LN79].

Definition 5.5.2 (Lauer-Needham Transform)**.** Let

$$*a(w : S).P$$

be a simple server. Then the mapping

$$LN[\![*a(w : S).P]\!]$$

is inductively defined by the rules in Figure 5.5.

$$
\begin{array}{rcl}
LN[\![*a(w:S).P]\!]^y & \stackrel{\text{def}}{=} & (\nu\ o,q,\vec{c})(\mathsf{Loop}\langle o,q\rangle^y \mid \overline{o} \mid q\langle(\mathtt{sd},a,\emptyset,c_0)\rangle \mid \\
& & \qquad \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle)^y \\
& & \\
& & \text{where } o,\ q \text{ and } \vec{c} = c_1..c_n \text{ are fresh and pairwise distinct.} \\
& & \\
\mathsf{Loop}\langle o,q\rangle^y & \stackrel{\text{def}}{=} & *o.\mathtt{select}\ (x_s,x_a,\tilde{x},z)\ \mathtt{from}\ q\ \mathtt{in}\ \mathtt{if}\ x_a = a\ \mathtt{then} \\
& & \mathtt{new}\ \mathtt{env}\ y\ \mathtt{in}\ \overline{z}\langle x_s,\mathtt{val}(y)\rangle.\mathbf{0} \\
& & \mathtt{else}\ \mathtt{typecase}\ x_s\ \mathtt{of}\ \{ \\
& & \qquad x_i : S_i :\ \overline{z}\langle x_s,\tilde{x}\rangle.\mathbf{0} \\
& & \}_{i\in I} \\
\mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle & \stackrel{\text{def}}{=} & \mathscr{B}[\![a(w:S).P]\!]^y \mid \prod_{i\in I}\mathscr{B}[\![P_i]\!]^y \\
& & \\
& & \text{where } \{P_i\}_{i\in I} = \mathsf{subterms}(P) \\
& & \text{and } \Gamma \vdash P_i \triangleright \Delta \cdot w : S_i \\
& & \\
\mathscr{B}[\![*a(w:S).P]\!]^y & \stackrel{\text{def}}{=} & *c_0(x_s,\tilde{x}).a(w':S). \\
& & \mathtt{register}\ (x_s,a,\emptyset,c_0)\ \mathtt{to}\ q\ \mathtt{in}\ [\![P,y\{w\mapsto w'\}]\!]^y \\
\mathscr{B}[\![x^{(i)}?(z);Q]\!]^y & \stackrel{\text{def}}{=} & *c_i(x',\tilde{x}).x'?(z');[\![Q,y\{z\mapsto z'\}\{w\mapsto x'\}]\!]^y \\
\mathscr{B}[\![x^{(i)}\&\{l_j:Q_j\}_{j\in J}]\!]^y & \stackrel{\text{def}}{=} & *c_i(x',\tilde{x}).x'\&\{l_j : [\![Q_j,y\{w\mapsto x'\}]\!]\}^y_{j\in I} \\
& & \\
[\![x!\langle e\rangle;Q,\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & [\![x]\!]_y!\langle[\![e]\!]_y\rangle;[\![Q,\mathtt{val}(y)]\!]^y \\
[\![x!\langle k\rangle;Q,\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & [\![x]\!]_y!\langle[\![k]\!]_y\rangle;[\![Q,\mathtt{val}(y)]\!]^y \\
[\![x\oplus l_j;Q,\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & [\![x]\!]_y\oplus l_j;[\![Q,\mathtt{val}(y)]\!]^y \\
[\![\overline{b}(z:S);Q,\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & \overline{b}(z':S).[\![Q,y\{z\mapsto z'\}]\!]^y \\
[\![Q,\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & \mathtt{register}\ ([\![w]\!]_y,\mathtt{shd},\mathtt{val}(y),c_{i+1})\ \mathtt{to}\ q\ \mathtt{in}\ \overline{o}.\mathbf{0}\ \ (Q \text{ is blocking}) \\
[\![\mathbf{0},\tilde{x}]\!]^y & \stackrel{\text{def}}{=} & \overline{o}.\mathbf{0}
\end{array}
$$

Figure 5.5: Translation Function for Lauer-Needham Transform

The transformation in Figure 5.5 uses the techniques of *cooperative task management* and *manual stack management* (cf. [AHT+02]). The simple server process is divided to its blocking subterms implementing a notion of code ripping. The event-loop process selects the next ready event and dispatches it to the corresponding blocking process subterm for processing. A tuple structure is a closure used to maintain the state of an event and its continuation between a blocking subterms. The storage of an event continuation implies the use of the *continuation passing style* - CPS[2]. We implemented CPS with the definition of replicated processes guarded by an input on the shared channels $o, c_1, \ldots, c_n$, which are passed as parameters and are used to invoke and pass the state closure to the next part of the execution.

The main map $LN[\![*a(w:S).P]\!]$ consists of:

1. A shared channel $\overline{o}$ at the output position is used to initiate the server.

2. A selector queue $q\langle(\mathsf{sd}, a, \emptyset, c_0)\rangle$ named q with the initial element $(\mathsf{sd}, a, \emptyset, c_0)$. The selector is used to register a tuple structure that consists of: i) a shared channel that is arrive-inspected for events, ii) a session channel that is arrive-inspected for events, iii) a tuple structure that maintains the closure state for a session execution, iv) the continuation channel c_i for the execution. The initial element consists of the shared channel a that accepts new session connections for the server.

3. An dynamic *event loop* $\mathsf{Loop}\langle o, q\rangle$ which denotes an loop invoked at shared channel o. The event loop uses the selector structure q to select the next ready event. A `typecase` construct decides how a selected event should be processed.

4. A collection of *code blocks* $\mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle$, each defined using an auxiliary map $\mathscr{B}[\![R]\!]$ and $[\![Q, \tilde{x}]\!]$. Each parallel process in the $\mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle$ process is derived out of the $\mathsf{subterms}(P)$ definition. Each blocking subterm R of P is mapped via the

[2]The continuation passing style is used in functional programming and requires that a function will get as an argument its continuation. After the termination of the function, the function is responsible for invoking the continuation code.

$\mathscr{B}[\![R]\!]$ mapping to create a a replicated, guarded and terminating used to process an event until the next blocking point. Codeblock processes are anotated with the corresponding i index from the $\mathsf{subterms}(P)$ definition. This is used to decide the index of the guard c_i and the continuation c_{i+1}. The guard on shared channels c_i is used to pass control to codeblock processes.

5. We use the operators for context manipulation to maintain the state of an event. Upon a session channel accept, in the $\mathsf{Loop}\langle o,q\rangle$ process, we create a new closure with the `new env` y `in` notation. Operator $[\![x]\!]_y$ is used in the $\mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle$ process and is responsible for getting a value from the closure structure. Operator $y\{l \mapsto v\}$ is used for updating the event structure and operator $\texttt{val}(y)$ is used when passing the closure structure around the execution.

The execution of $LN[\![*a(w:S).P]\!]$ starts with the input on the guard of the $\mathsf{Loop}\langle o,q\rangle$ process. The $\mathsf{Loop}\langle o,q\rangle$ fetches a channel from the selector queue at which a message has arrived via the select operator. What the select returns is a structure containing a shared name, a session name, a closure and a channel used for continuation. First the shared name is matched against the shared name a to check whether there is a new session to accept. If the match succeeds the a new event closure is created and the event-loop will pass control to Codeblock process $\mathscr{B}[\![a(w:S).P]\!]$ via the shared channel c_0. Once invoked, the initial code block, $\mathscr{B}[\![a(w:S).P]\!]$, receives a fresh session channel through the endpoint of a, saves it in the environment closure, and moves to $[\![P,\tilde{x}]\!]$. The code $[\![P,\tilde{x}]\!]$ carries out "instructions" from P, using the environment denoted by y to interpret variables. After completing all the consecutive *non-blocking actions* (invocations, outputs, selections, conditionals and recursions) starting from the initial input, the code will reach a blocking prefix or $\mathbf{0}$. If the former is the case, it registers the blocking session channel, the associated continuation and the current environment in the selector queue q. Then the control flow returns to the event-loop via the output on $\overline{o}$.

If the shared name match, in the $\mathsf{Loop}\langle o,q\rangle$ process fails (the shared name is a dummy name),

it means that the selector has returned a session name. The session name is then typechecked by a `typecase` operator and invokes the corresponding continuation code block, passing the session channel and the corresponing state environment via the continuation channel c_i. The code block, which has the shape $\mathscr{B}[\![P_i]\!]$ for a blocking sub-term P_i of P, now receives the message via the passed session channel, saves it in the passed environment, and continues with the remaining behaviour until it reaches a blocking action, in the same way as illustrated for the initial code block. The combination of a typecase and a session channel passing above enables the protection of session type abstraction, ensuring type and communication safety.

Example 5.5.1 (Lauer-Needham Transform)**.** As an example of a server, consider:

$$P = *a(x).x?(w);x!\langle w+1\rangle;x?(z);x?(w+z);\mathbf{0} \mid a[\varepsilon]$$

This process has the session type $?(\mathtt{nat});!\langle\mathtt{nat}\rangle;?(\mathtt{nat});!\langle\mathtt{nat}\rangle;\mathtt{end}$ at a which can be read: *a process should first expect to receive a message of type* `nat`*, then send a* `nat`*, then to receive again a* `nat`*, and finish by sending a result.* We extract the blocking subterms from this process as follows.

Blocking Process	Type at Blocking Prefix
$a(x).x?(w);x!\langle w+1\rangle;x?(z);x!\langle w+z\rangle;\mathbf{0}$	$\mathtt{i}\langle?(\mathtt{nat});!(\mathtt{nat});?(\mathtt{nat});!(\mathtt{nat})\rangle$
$x?(w);x!\langle w+1\rangle;x?(z);x!\langle w+z\rangle;\mathbf{0}$	$?(\mathtt{nat});!(\mathtt{nat});?(\mathtt{nat});!(\mathtt{nat})$
$x?(z);x!\langle w+z\rangle;\mathbf{0}$	$?(\mathtt{nat});!(\mathtt{nat})$

These blocking processes are translated into *code blocks*, denoted CodeBlocks, given as:

$$*c_0(x_s,\tilde{x}).a(x').\mathtt{register}\ q\ \mathtt{to}\ (x_s,a,\emptyset,c_0)\ \mathtt{in}\ \mathtt{register}\ q\ \mathtt{to}\ (x',\mathtt{shd},y\{x\mapsto x'\},c_1)\ \mathtt{in}\ ;\overline{o}\ \mid$$
$$*c_1(x,\tilde{x}).x?(w');x!\langle w'+1\rangle;\mathtt{register}\ q\ \mathtt{to}\ (x,\mathtt{shd},y\{w\mapsto w'\},c_2)\ \mathtt{in}\ \overline{o}\ \mid$$
$$*c_2(x,y).x?(z');x!\langle w+z'\rangle;\overline{o}$$

which is used for processing each message. Above, the operation `register` stores the blocking session channel, the associated continuation c_i and the current environment y in a selector queue sel.

Finally, using these code blocks, the main event-loop denoted Loop, is given as:

$$
\begin{array}{lll}
\texttt{Loop} & = & *o.\texttt{select}\ (x_s, x_a, y, \tilde{x})\ \texttt{from}\ q\ \texttt{in}\ \texttt{if}\ x_a = a\ \texttt{then}\ \texttt{new}\ \texttt{env}\ y\ \texttt{in}\ \bar{z}\langle x_s, \tilde{x}\rangle.\ \texttt{else} \\
& & \texttt{typecase}\ x_s\ \texttt{of}\ \{ \\
& & \quad x_1 : ?(\texttt{nat}); !\langle\texttt{nat}\rangle; ?(\texttt{nat}); !\langle\texttt{nat}\rangle; \texttt{end} : \quad \bar{z}\langle x_s, \tilde{x}\rangle. \\
& & \quad x_2 : ?(\texttt{nat}); !\langle\texttt{nat}\rangle; \texttt{end} : \quad \bar{z}\langle x_s, \tilde{x}\rangle. \\
& & \}
\end{array}
$$

Above `select from` q `in` selects a message from the selector queue *sel*, and treats it in P. The `new` construct creates a new environment y. The `typecase` construct then branches into different processes depending on the session of the received message, and dispatches the task to each code block.

A server is currently stateless, because the construction of the server does not allow an internal shared state to be accessed by the spawned threads[3]. Hence we have:

Lemma 5.5.1. Let $*a(x).P \mid a[\varepsilon]$ be a simple server. Then $*a(x).P \mid a[\varepsilon]$ is confluent.

Proof. The proof is straightforward by using the confluence properties for the ESP, studied in § 4.3.3 (see Appendix B.3). □

Lemma 5.5.2. Let $*a(x).P \mid a[\varepsilon]$ be a simple server. Then $LN[\![*a(x).P \mid a[\varepsilon]]\!]$ is confluent.

Proof. For proof see, Appendix B.3. □

Theorem 5.5.1 (Semantic Preservation)**.** Let $*a(x).P \mid a[\varepsilon]$ be a simple server. Then we have $*a(x).P \mid a[\varepsilon] \approx LN[\![*a(x).P \mid a[\varepsilon]]\!]$.

Proof. The proof of the above theorem constructs a determinate up-to expansion relation, cf. Definition 4.3.5 and Lemma 3.3.6. The up-to expansion relation contains each process pair that has all the parallel processes on a blocking prefix for the threaded server and starts

[3] The transform can be extended to the situation where threads share state, though its behavioural justification takes a different form.

from the Loop process for the thread-free process, with arbitrary localisations. We show the conditions needed Definition 4.3.5 by using Lemmas 5.5.1, 5.5.2, as well as Lemma 5.4.2. We conclude the proof using Lemma 3.3.6. For details of the proof, see Appendix B.3. □

Part II

Chapter 6

Multiparty Session Types Behavioural Theory

Multiparty session types [HYC08, B$^+$08] were developed to overcome the limitations presented by the binary definition of session types. Influenced by the idea of communication choreography, multiparty session types offer a type-safe framework for a multi-participant communication scenario.

Choreography in communication requires knowledge in advance of the communication scenario by all computing processes. This was expressed through a structure called the global session type. Global session types describe the communication choreography among the different participants. A local projection procedure projects the global type to individual local session types for each participant. Local session types can be considered as a more refined structure of binary session types. The implementation of each participant should conform with its local projection to guarantee type safety and progress properties for the entire distributed program.

In this Chapter we are interested in studying the behavioural theory for multiparty session types following the principles set out in Chapter 3. We first develop a core synchronous mul-

tiparty session π-calculus and then extend it, in a straightforward way to three asynchronous multiparty session π-calculi. What distinguishes the three asynchronous calculi is the locality of the intermediate buffers that are used to achieve asynchronous communication. A local output buffer locally stores values until they are sent on an endpoint, while a local input buffer locally stores values that were received on an endpoint. A third version considers the existence of both output and input buffers for session endpoints (similar with the calculi presented in Chapter 3 and Chapter 4). The semantic difference in input and output localities result in a different behaviour for each of the four multiparty session π-calculi. All four calculi are strongly related with the asynchronous multiparty framework developed in [B+08]. We then develop a bisimulation framework based on different typed labelled transition systems.

The typed labelled transition semantics take into account the session type information from typed processes. A main innovation in this chapter is the study of the impact of the global session type on the label transition system in contrast to a labelled transition system that is defined using only the local session type information.

All the bisimulation definitions developed in this chapter are sound and complete with respect to their corresponding reduction-closed congruence relation.

6.1 Intuition for the Multiparty Behavioural Theory

We use the means of example to demonstrate the basic insights for the development of a multiparty behavioural theory. Figure 6.1 describes two multiparty scenarios for a client that is concurrently served by two servers on the same multiparty session. We assume the syntax and the semantics for synchronous multiparty session types, which we believe are intuitive (for a formal definition, see § 6.2).

In scenario (a) of Figure 6.1 process Client_3 communicates with the single threaded process Server_1 and the multithreaded process Server_2 via multiparty session s_1. Process Server_1 is

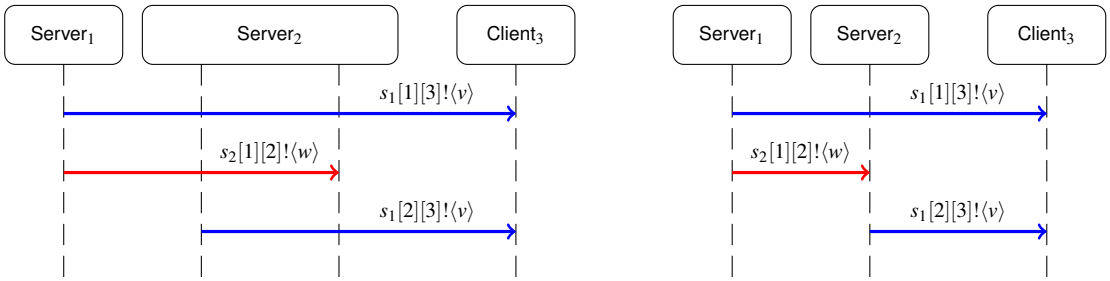

Figure 6.1: Resource Managment Example: (a) before optimisation; (b) after optimisation

responsible for sending value v to process Client_3 (expressed with label $s[1][3]!\langle v\rangle$) and similarly the first thread of process Server_2 is responsible for sending a value v to process Client_3 (label $s[2][3]!\langle v\rangle$). Furthermore process Server_1 is communicating with the second thread of process Server_2 via the intra-service session s_2, to exchange value w (label $s[1][2]!\langle w\rangle$). The process for the three participants can be expressed in the synchronous multiparty session types as:

$$S_1 = s_1[1][3]!\langle v\rangle; s_2[1][2]!\langle w\rangle; \mathbf{0} \qquad S_2 = s_1[2][3]!\langle v\rangle; \mathbf{0} \mid s_2[2][1]?(x); \mathbf{0}$$
$$C_3 = s_1[3][1]?(x); s_1[3][2]?(y); \mathbf{0}$$

The multiparty protocols for channels $s_1 : G_1$ and $s_2 : G_2$ are defined as:

$$G_1 = 1 \to 3 : \langle V\rangle.2 \to 3 : \langle V\rangle.\mathtt{end} \qquad G_2 = 1 \to 2 : \langle W\rangle.\mathtt{end}$$

where in channel s_1 participant 1 sends a value with type V to participant 3 and then participant 2 sends a value with the same type again to participant 3 and the session terminates, while channel s_2 expects the send of a value with type W from participant 1 to participant 2 before it terminates.

In the second scenario (b) we want to optimise Server_2 and avoid the overhead for thread creation. We also want to maintain the global protocol specification $s_1 : G_1, s_2 : G_2$ for the entire scenario. The scenario for Server_2 now expects a message from Server_1 (label $s_2[1][2]!\langle w\rangle$) and then sends value v to the Client_3 (label $s_2[2][3]!\langle v\rangle$). The new process for Server_2 now

becomes:

$$S_2' = s_2[2][1]?(x); s_1[2][3]!\langle v\rangle; \mathbf{0}$$

Note that S_2' respects $s_1 : G_1$ and $s_2 : G_2$. Clearly, the behaviour of processes $S_1 \mid S_2$ and $S_1 \mid S_2'$ on the actions they exhibit with respect to an arbitrary observer is not the same, since the former can act as $S_1 \mid S_2 \xrightarrow{s[2][3]!\langle v\rangle}$ while the latter cannot. Nevertheless, because of the restriction on global protocols $s_1 : G_1$ and $s_2 : G_2$ the former process can replace the latter without any problems in the communication with the Client_3 process.

This last observation drives our insight to define a pair of bisimilarity relations for the study of multiparty session types. The first bisimilarity relation is based on the information from the local session type of a process (does not relate $S_1 \mid S_2$ and $S_1 \mid S_2'$) and the second relation takes into account the information from the global session type assignment on session channels (relates $S_1 \mid S_2$ and $S_1 \mid S_2'$) . Furthermore, we are interested in studying the relation between the two bisimilarities.

6.2 Synchronous Multiparty Session π-Calculus as a Core Calculus

This section defines a synchronous version of the multiparty session π-calculus (or synchronous MSP). The syntax follows [B+08] except we eliminate queues for asynchronous communication. The synchronous version of the Multiparty Session π-calculus will be used as the reference theory to be extended to a set of different asynchronous MSP calculi.

6.2.1 Syntax and Operational Semantics

Figure 6.2 defines the syntax for the synchronous MSP. Shared names are denoted as $a, b, \ldots$, session names as $s, s', \ldots$, variables as $x, y, \ldots$ and constants as $\mathtt{tt}, \mathtt{ff}, \ldots$. We call $\mathtt{p}, \mathtt{q}, \ldots$ the

(Processes)	P	::=	$\overline{u}[\mathtt{p}](x).P$	Request
		|	$u[\mathtt{p}](x).P$	Accept
		|	$c[\mathtt{p}]!\langle e\rangle;P$	Sending
		|	$c[\mathtt{p}]?(x);P$	Receiving
		|	$c[\mathtt{p}]\oplus l;P$	Selection
		|	$c[\mathtt{p}]\&\{l_i : P_i\}_{i\in I}$	Branching
		|	$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$	Conditional
		|	$P \mid Q$	Parallel
		|	$\mathbf{0}$	Inaction
		|	$(\nu\ n)P$	Hiding
		|	$\mu X.P$	Recursion
		|	X	Variable
(Identifiers)	u	::=	$x \mid a$	
(Name)	n	::=	$s \mid a$	
(Expression)	e	::=	$v \mid x \mid e\ \mathtt{and}\ e'$	
		|	$e = e' \mid \ldots$	
(Session)	c	::=	$s[\mathtt{p}] \mid x$	
(Value)	v	::=	$a \mid \mathtt{tt} \mid \mathtt{ff} \mid s[\mathtt{p}]$	

Figure 6.2: Syntax for synchronous multiparty session calculus

participants and let them to range over natural numbers. We write $s[\mathtt{p}]$ to denote a participant $\mathtt{p}$ on session s and we call it a session role. Symbol u ranges over shared names or variables and c ranges over session roles or variables. Values $v, v', \ldots$ range over shared names, constants, or roles. Expressions $e, e', \ldots$ are either values, logical operations on expressions or name matching operations.

For the primitives for session initiation, $\overline{u}[\mathtt{p}](x).P$ initiates a new session through an identifier u (which represents a shared interaction point) with the other multiple participants, each of shape $u[\mathtt{p}](x)..Q_{\mathtt{q}}$ where $1 \leq \mathtt{q} \leq \mathtt{p}-1$. The (bound) variable x is the channel used to do the communications. Session communications (communications that take place inside an established session) are performed using the next two pairs of processes: the sending and

$$
\begin{array}{rcll}
P & \equiv_\alpha & P & \\
P & \equiv & P \mid \mathbf{0} & \\
P \mid Q & \equiv & Q \mid P & \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
\mu X.P & \equiv & P\{\mu X.P/X\} & \\
(\nu\ n)(\nu\ n')P & \equiv & (\nu\ n')(\nu\ n)P & \\
(\nu\ n)\mathbf{0} & \equiv & \mathbf{0} & \\
(\nu\ n)(P) \mid Q & \equiv & (\nu\ n)(P \mid Q) & n \notin \mathtt{fn}(Q)
\end{array}
$$

Figure 6.3: Structural Congruence for Synchronous Multiparty Session Calculus

receiving of a value and the selection and branching (where the former chooses one of the branches offered by the latter). The input/output operations specify the sender and the receiver, respectively. Hence $c[\mathtt{p}]!\langle e\rangle;P$ sends a value to participant $\mathtt{p}$; accordingly, $c[\mathtt{p}]?(x);P$ denotes the intention of receiving a value from the participant $\mathtt{p}$. The same holds for selection/branching. We call $s[\mathtt{p}]$ a *channel with role*: it represents the channel of the participant $\mathtt{p}$ in the session s. Process $\mathbf{0}$ is the inactive process. The rest of the processes are standard π-calculus processes. Process $P \mid Q$ is the parallel composition of processes P and Q. Process $(\nu\ n)P$ restricts name n in the scope of P. Term X is the process variable used in the standard μ-recursive expression $\mu X.P$. We say that a process is closed it does not contain free variables. We denote $\mathtt{fn}(P)$/$\mathtt{bn}(P)$ and $\mathtt{fv}(P)$/$\mathtt{bv}(P)$ for a set of free/bound names and free/bound variables, respectively.

Structural Congruence

The structural congruence rules are defined in Figure 6.3. Processes are structurally congruent up-to alpha renaming. A parallel composition of terms P and $\mathbf{0}$ has no structural effect on process P. Associativity and commutativity hold for parallel composition. Recursive unfolding is defined up-to structural congruence. Restriction order is irrelevant and restricting a name in term $\mathbf{0}$ is congruent with $\mathbf{0}$. Finally a restricted name n for a process P can still be

$$a[1](x).P_1 \mid \ldots \mid \overline{a}[n](x).P_n \longrightarrow (\nu\, s)(P_1\{s[1]/x\} \mid \ldots \mid P_n\{s[n]/x\}) \quad \text{[Link]}$$

$$s[\mathtt{p}][\mathtt{q}]!\langle e\rangle;P \mid s[\mathtt{q}][\mathtt{p}]?(x);Q \longrightarrow P \mid Q\{v/x\} \quad (e \downarrow v) \quad \text{[Comm]}$$

$$s[\mathtt{p}][\mathtt{q}] \oplus l_k;P \mid s[\mathtt{q}][\mathtt{p}]\&\{l_i : P_i\}_{i\in I} \longrightarrow P \mid P_k \quad k \in I \quad \text{[Label]}$$

$$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P \quad (e \downarrow \mathtt{tt}) \quad \text{[If-True]}$$

$$\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q \quad (e \downarrow \mathtt{ff}) \quad \text{[If-False]}$$

$$\frac{P \longrightarrow P'}{(\nu\, n)P \longrightarrow (\nu\, n)P'} \quad \text{[Res]}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \text{[Par]}$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} \quad \text{[Str]}$$

Figure 6.4: Operational semantics for synchronous multiparty session calculus

restricted in a parallel process of P and Q provided that n does not occur free in Q.

Operational semantics

Operational semantics of the calculus are defined in Figure 6.4. Rule [Link] defines synchronous session initiation. All session roles must be present to synchronously reduce each role $\mathtt{p}$ on a fresh session name $s[\mathtt{p}]$. Rule [Comm] is for sending a value to the corresponding receiving process where $e \downarrow v$ means expression e evaluates to value v. The interaction between selection and branching is defined via rule [Label], where a select prefix on role $s[\mathtt{p}]$ selects branch P_k offered by role $s[\mathtt{q}]$ via label l_k. The rest of the operational rules are standard π-calculus operational rules. Rules [If-True] and [If-False], describe the reduction semantics for the standard control construct $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$, where process P is selected if e evaluates to true ($\mathtt{tt}$) and process Q otherwise. Rule [Res] states that the restriction of a name n in a process P does not affect internal reductions of P. Similarly rule [Par] states that the parallel composition of a process P does not affect its internal reductions. Rule [Str] closes the reduction relation over structural congruence $\equiv$. We write $\twoheadrightarrow$ for $(\longrightarrow \cup \equiv)^*$.

Exchange	U	$::=$	$S \mid T$	
Sort	S	$::=$	$\texttt{bool} \mid \langle G \rangle$	
Global	G	$::=$	$\texttt{p} \to \texttt{q} : \langle U \rangle.G'$	exchange
		$\mid$	$\texttt{p} \to \texttt{q} : \{l_i : G_i\}_{i \in I}$	branching
		$\mid$	$\mu t.G$	recursion
		$\mid$	t	variable
		$\mid$	$\texttt{end}$	end

Figure 6.5: Global types

6.2.2 Session Types for Synchronous Multiparty Session π-calculus

In this subsection we define the basic global type session type syntax. We define global types, which we project to local types via a projection algorithm. The properties for local types are clarified through the local type projection algorithm and the duality relation.

Global types, ranged over by $G, G', \ldots$ describe the whole conversation scenario of a multiparty session as a type signature. Their grammar is given in Figure 6.5. The global type $\texttt{p} \to \texttt{q} : \langle U \rangle.G'$ says that participant $\texttt{p}$ sends a message of type U to participant $\texttt{q}$ and then interactions described in G' take place. Type $\texttt{p} \to \texttt{q} : \{l_i : G_i\}_{i \in I}$ says participant $\texttt{p}$ sends one of the labels l_i to $\texttt{q}$. If l_j is sent with $j \in I$, interactions described in G_j take place. In both cases we assume for well-formedness that $\texttt{p} \neq \texttt{q}$. Type $\mu t.G$ is a recursive type, assuming type variables $(t, t', \ldots)$ are guarded in the standard way, i.e. type variables only appear under some prefix. We take an *equi-recursive* view of recursive types, not distinguishing between $\mu.G$ and its unfolding $G\{\mu t.G/t\}$ [Pie02, § 21.8]. Type $\texttt{end}$ represents the termination of the session. *Exchange types* $U, U', \ldots$ consist of *sorts* types $S, S', \ldots$ for values (either base types or global types), and *local session* types $T, T', \ldots$ for channels (defined in the next paragraph). We assume that G in the grammar of sorts is closed, i.e. without free type variables.

Local	T	$::=$	$[\mathtt{p}]!\langle U\rangle;T$	send
		$\mid$	$[\mathtt{p}]?(U);T$	receive
		$\mid$	$[\mathtt{p}]\oplus\{l_i : T_i\}_{i\in I}$	selection
		$\mid$	$[\mathtt{p}]\&\{l_i : T_i\}_{i\in I}$	branching
		$\mid$	$\mu t.T$	recursion
		$\mid$	t	variable
		$\mid$	$\mathtt{end}$	end

Figure 6.6: Local types

Local types are defined in Figure 6.6 and correspond to the communication actions, representing sessions from the view-points of single participants. We often call local types *session types*. The *send type* $[\mathtt{p}]!\langle U\rangle;T$ expresses the sending of a value of type U to participant $\mathtt{p}$, followed by the communications of T. The *selection type* $[\mathtt{p}]\oplus\{l_i : T_i\}_{i\in I}$ represents the transmission of a label l_i chosen in the set $\{l_i \mid i\in I\}$ to participant $\mathtt{p}$ followed by the communications described by T_i. The *receive type* $[\mathtt{p}]?(U);T$ is dual to the send type, and expresses the input of a value value with type U from participant $\mathtt{p}$ and then continues as T. Similarly the *branching type* $[\mathtt{p}]\&\{l_i : T_i\}$ offers the set of labels $\{l_i \mid i\in I\}$ for a selection between the types $T_i \mid i\in I$ respectively. The inactive type is represented with the $\mathtt{end}$ term. The recursive variable t is used for the standard μ-recursive type $\mu t.T$.

We proceed with the definition of global and local participants.

Definition 6.2.1 (Global and Local Participants)**.**

- We define $\mathtt{partic}(G)$ as:

$$\mathtt{partic}(\mathtt{end}) = \emptyset \qquad \mathtt{partic}(t) = \emptyset \qquad \mathtt{partic}(\mu t.G) = \mathtt{partic}(G)$$

$$\begin{aligned} \mathtt{partic}(\mathtt{p}\to\mathtt{q} : \langle U\rangle.G) &= \{\mathtt{p},\mathtt{q}\}\cup\mathtt{partic}(G) \\ \mathtt{partic}(\mathtt{p}\to\mathtt{q} : \{l_i : G_i\}_{i\in I}) &= \{\mathtt{p},\mathtt{q}\}\cup\{\mathtt{partic}(G_i) \mid i\in I\} \end{aligned}$$

- We define $\mathtt{partic}(T)$ on local types as:

$$\mathtt{partic}(\mathtt{end}) = \emptyset \qquad \mathtt{partic}(t) = \emptyset \qquad \mathtt{partic}(\mu t.T) = \mathtt{partic}(T)$$

$$\begin{aligned}
\mathtt{partic}([\mathtt{p}]!\langle U\rangle;T) &= \{\mathtt{p}\} \cup \mathtt{partic}(T) \\
\mathtt{partic}([\mathtt{p}]?(U);T) &= \{\mathtt{p}\} \cup \mathtt{partic}(T) \\
\mathtt{partic}([\mathtt{p}] \oplus \{l_i : T_i\}_{i\in I}) &= \{\mathtt{p}\} \cup \mathtt{partic}(T) \\
\mathtt{partic}([\mathtt{p}] \& \{l_i : T_i\}_{i\in I}) &= \{\mathtt{p}\} \cup \mathtt{partic}(T)
\end{aligned}$$

Global participants sets for global types $\mathtt{partic}(G)$ and local types $\mathtt{partic}(T)$ are inductively defined on the syntax of global types and local types respectively and include all participants in a type.

Global and Local Projection

The relation between global and local types is formalised by the standard projection function [HYC08]. We proceed with the definition of global types projection:

Definition 6.2.2 (Global Projection)**.** The projection of a global type G onto a participant $\mathtt{p}$ is defined by induction on G:

$$
\begin{aligned}
\mathtt{p}' \to \mathtt{q} : \langle U \rangle . G\lceil \mathtt{p} &= \begin{cases} [\mathtt{q}]!\langle U \rangle ; G\lceil \mathtt{p} & \mathtt{p} = \mathtt{p}' \\ [\mathtt{p}']?(U); G\lceil \mathtt{p} & \mathtt{p} = \mathtt{q} \\ G\lceil \mathtt{p} & \text{otherwise} \end{cases} \\
\mathtt{p}' \to \mathtt{q} : \{l_i : G_i\}_{i \in I}\lceil \mathtt{p} &= \begin{cases} [\mathtt{q}] \oplus \{l_i : G_i\lceil \mathtt{p}\}_{i \in I} & \mathtt{p} = \mathtt{p}' \\ [\mathtt{p}'] \& \{l_i : G_i\lceil \mathtt{p}\}_{i \in I} & \mathtt{p} = \mathtt{q} \\ G_1\lceil \mathtt{p} & \text{if } \forall j \in I.\ G_1\lceil \mathtt{p} = G_j\lceil \mathtt{p} \end{cases} \\
(\mu t.G)\lceil \mathtt{p} &= \begin{cases} \mu t.(G\lceil \mathtt{p}) & \mathtt{p} \in G \\ \mathtt{end} & \text{otherwise} \end{cases} \\
t\lceil \mathtt{p} &= t \\
\mathtt{end}\lceil \mathtt{p} &= \mathtt{end}
\end{aligned}
$$

Inactive `end` and recursive variable t types are projected to their respective local types. We project a global type $\mathtt{p}' \to \mathtt{q} : \langle U \rangle . G$ to participant $\mathtt{p}$ as a sending local type if $\mathtt{p} = \mathtt{p}'$ and as a receiving local type if $\mathtt{p} = \mathtt{q}$. In any case the continuation of the projection is $G\lceil \mathtt{p}$. For $\mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i \in I}$ global type the projection is the select local type for $\mathtt{p} = \mathtt{p}'$ and the branch local type $\mathtt{p} = \mathtt{q}$. Otherwise we use the projection of one of the $\{G_i \mid i \in I\}$ global types (all types G_i should have the same projection with respect to $\mathtt{p}$). Recursion $\mu t.G$ is projected onto a local type using local recursion and the projection of the global type G with respect to $\mathtt{p}$.

Definition 6.2.3 (Projection Set)**.** The *projection set* of $s : G$ is defined as

$$\mathtt{proj}(s : G) = \{s[\mathtt{p}] : G\lceil \mathtt{p} \mid \mathtt{p} \in \mathtt{partic}(G)\}$$

We define the following projection from a local type T to produce binary session types needed for defining coherency and well-formedness properties later.

Definition 6.2.4 (Local projection)**.** The projection of a local type T onto a participant $\mathtt{p}$ is defined by induction on T:

$$
\begin{array}{rcl}
[\mathtt{p}]!\langle U\rangle;T\lceil\mathtt{q} & = & \begin{cases} !\langle U\rangle;T\lceil\mathtt{q} & \mathtt{q}=\mathtt{p} \\ T\lceil\mathtt{q} & \text{otherwise} \end{cases} \\[2ex]
[\mathtt{p}]?(U);T\lceil\mathtt{q} & = & \begin{cases} ?(U);T\lceil\mathtt{q} & \mathtt{q}=\mathtt{p} \\ T\lceil\mathtt{q} & \text{otherwise} \end{cases} \\[2ex]
[\mathtt{p}]\oplus\{l_i:T_i\}_{i\in I}\lceil\mathtt{q} & = & \begin{cases} \oplus\{l_i:T_i\lceil\mathtt{q}\}_{i\in I} & \mathtt{q}=\mathtt{p} \\ T_1\lceil\mathtt{q} & \text{if } \forall i\in I.T_i\lceil\mathtt{q}=T_1\lceil\mathtt{q} \end{cases} \\[2ex]
[\mathtt{p}]\&\{l_i:T_i\}_{i\in I}\lceil\mathtt{q} & = & \begin{cases} \&\{l_i:T_i\lceil\mathtt{q}\}_{i\in I} & \mathtt{q}=\mathtt{p} \\ T_1\lceil\mathtt{q} & \text{if } \forall i\in I.T_i\lceil\mathtt{q}=T_1\lceil\mathtt{q} \end{cases} \\[2ex]
(\mu t.T)\lceil\mathtt{q} & = & \mu t.(T\lceil\mathtt{q}) \\
t\lceil\mathtt{q} & = & t \\
\mathtt{end}\lceil\mathtt{q} & = & \mathtt{end}
\end{array}
$$

Inactive local type and the recursive variables are always projected to their corresponding binary session types syntax. The recursion operator $\mu t.T$ is projected onto the corresponding binary session types syntax. The types $[\mathtt{p}]!\langle U\rangle;T, [\mathtt{p}]?(U);T$ are projected with respect to $\mathtt{q}$ to binary session type send and binary session type receive respectively, and continue with the projection of T on q if $\mathtt{p}=\mathtt{q}$. If $\mathtt{p}\neq\mathtt{q}$ local projection continues with the projection of T. There is a similar argument for $[\mathtt{p}]\oplus\{l_i:T_i\}_{i\in I}, [\mathtt{p}]\&\{l_i:T_i\}_{i\in I}$, where in the case of $\mathtt{p}=\mathtt{q}$ the projection follows binary session types. In the case where $\mathtt{p}\neq\mathtt{q}$ we project one of the continuations in $\{T_i\}_{i\in I}$ since we expect all the projections to be the same.

The duality over the binary session types is defined in Figure 6.7. Duality is inductively defined on the structure of binary session types with $\mathtt{end}$ and t to be homomorphic. The send

$$\begin{array}{rclcrcl}
\overline{!\langle U\rangle;T} & = & ?(U);\overline{T} & \qquad & \overline{?(U);T} & = & !\langle U\rangle;\overline{T} \\
\overline{\oplus\{l_i : T_i\}_{i\in I}} & = & \&\{l_i : \overline{T_i}\}_{i\in I} & & \overline{\&\{l_i : T_i\}_{i\in I}} & = & \oplus\{l_i : \overline{T_i}\}_{i\in I} \\
\mathtt{end} & = & \overline{\mathtt{end}} & \qquad t = \bar{t} \qquad & \overline{\mu t.T} & = & \mu t.\overline{T}
\end{array}$$

Figure 6.7: Multiparty Session Duality

prefix is dual to the receive session types. Similarly the select type is dual to the branch type. Recursion is defined inductively on the structure of the recursive type.

Lemma 6.2.1. *If* $\mathtt{p},\mathtt{q} \in \mathtt{partic}(G)$ *then* $(G\lceil\mathtt{p})\lceil\mathtt{q} = \overline{(G\lceil\mathtt{q})\lceil\mathtt{p}}$.

Proof. The proof is an induction on the syntax structure of G. See Appendix C.1.1 for details. □

6.2.3 Typing System and its Properties

The typing system is expressed though typing judgements for expressions and processes, that have the shapes:

$$\Gamma \vdash e : S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

where Γ is the standard environment which associates variables to sort types, shared names to global types and process variables to session environments; and Δ is the session environment which associates channels to session types. Formally we define:

$$\Gamma \ ::= \ \emptyset \ \mid \ \Gamma\cdot u : S \ \mid \ \Gamma\cdot X : \Delta \quad \text{and} \quad \Delta \ ::= \ \emptyset \ \mid \ \Delta\cdot s[\mathtt{p}] : T$$

assuming we can write $\Gamma\cdot u : S$ if $u \notin \mathtt{dom}(\Gamma)$. We extend this to a concatenation for typing environments as $\Delta\cdot\Delta' = \Delta\cup\Delta'$.

Coherency is an important property for the soundness of session environments. We define coherency of session environments as follows:

Definition 6.2.5 (Coherency)**.** Typing Δ is *coherent with respect to session* s, written notation $\mathtt{co}(\Delta(s))$, if

$$\forall s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}} \in \Delta, \mathtt{p} \neq \mathtt{q} \text{ then } T_{\mathtt{p}} \lceil \mathtt{q} = \overline{T_{\mathtt{q}} \lceil \mathtt{p}}$$

A typing Δ is *coherent*, written $\mathtt{co}(\Delta)$, if it is coherent with respect to all s in its domain. We say that the typing judgement $\Gamma \vdash P \triangleright \Delta$ is *coherent* if $\mathtt{co}(\Delta)$.

Typing System

We proceed with the definition of the typing system. The typing rules are essentially identical to the communication typing system for programs in [B+08] (since we do not require session queues). Figure 6.8 defines the typing system. Rule [Name] types a shared name or shared variable to $\langle G \rangle$. Boolean $\mathtt{tt}, \mathtt{ff}$ are typed with the $\mathtt{bool}$ type via rule [Bool]. Logical expressions are also typed with the $\mathtt{bool}$ type via rule [And], etc. Rules [MReq] and [MAcc] check that the local type of a session role agrees with the global type of the initiating shared name. Rules [Send] and [Recv] prefix the local type with send and receive local types respectively, after checking the type environment for the sending value type (receiving variable type resp.). Delegation is typed under rules [Deleg] and [Srecv] where we check type consistency of the delegating/receiving session role. Rules [Sel] and [Bra] type select and branch processes respectively. A select process uses the select local type. A branching process checks that all continuing process have consistent typing environments. [Conc] types a parallel composition of processes by checking the disjointness of their typing environments. Conditional is typed with [If], where we check the expression e to be of $\mathtt{bool}$ type and the branching processes to have the same typing environment. Rule [Nres] defines the typing for shared name restriction. Rule [Sres] uses the coherency property to restrict a session name. Rule [Var] assigns a session environment to process variable X with respect to the shared environment Γ and rule [Rec]

$$\Gamma \cdot u : S \vdash u : S \quad [\text{Name}] \qquad \Gamma \vdash \texttt{tt}, \texttt{ff} : \texttt{bool} \quad [\text{Bool}]$$

$$\frac{\Gamma \vdash e_i : \texttt{bool}}{\Gamma \vdash e_1 \ \texttt{and} \ e_2 : \texttt{bool}} \quad [\text{And}]$$

$$\frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x[\mathtt{p}] : G\lceil \mathtt{p} \quad \max(\texttt{partic}(G)) = \mathtt{p}}{\Gamma \vdash \overline{a}[\mathtt{p}](x).P \triangleright \Delta} \quad [\text{MReq}] \qquad \frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta \cdot x[\mathtt{p}] : G\lceil \mathtt{p}}{\Gamma \vdash a[\mathtt{p}](x).P \triangleright \Delta} \quad [\text{MAcc}]$$

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash P \triangleright \Delta \cdot c : T}{\Gamma \vdash c[\mathtt{q}]!\langle e \rangle; P \triangleright \Delta \cdot c : [\mathtt{q}]!\langle S \rangle; T} \quad [\text{Send}] \qquad \frac{\Gamma \cdot x : S \vdash P \triangleright \Delta \cdot c : T}{\Gamma \vdash c[\mathtt{q}]?(x); P \triangleright \Delta \cdot c : [\mathtt{q}]?(S); T} \quad [\text{Recv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot c : T}{\Gamma \vdash c[\mathtt{q}]!\langle c' \rangle; P \triangleright \Delta \cdot c : [\mathtt{q}]!\langle T' \rangle; T \cdot c' : T'} \quad [\text{Deleg}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot c : T \cdot x : T'}{\Gamma \vdash c[\mathtt{q}]?(x); P \triangleright \Delta \cdot c : [\mathtt{q}]?(T'); T} \quad [\text{SRecv}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot c : T}{\Gamma \vdash c[\mathtt{q}] \oplus l_i; P \triangleright \Delta \cdot c : [\mathtt{q}] \oplus \{l_i : T_i\}_{i \in I}} \quad [\text{Sel}]$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta \cdot c : T_i \quad \forall\, i \in I}{\Gamma \vdash c[\mathtt{q}] \& \{l_i : P_i\}_{i \in I} \triangleright \Delta \cdot c : [\mathtt{q}] \& \{l_i : T_i\}_{i \in I}} \quad [\text{Bra}]$$

$$\frac{\Gamma \vdash P_i \triangleright \Delta_i \quad i \in \{1,2\} \quad \Delta_1 \cap \Delta_2 = \emptyset}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cdot \Delta_2} \quad [\text{Conc}] \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash P_i \triangleright \Delta \quad i \in \{1,2\}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \triangleright \Delta} \quad [\text{If}]$$

$$\frac{\Delta \ \texttt{end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \quad [\text{Inact}] \qquad \frac{\Gamma \cdot a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu\ a)P \triangleright \Delta} \quad [\text{NRes}]$$

$$\frac{\texttt{co}(\{s[1] : T_1 \ldots s[n] : T_n\}) \quad \Gamma \vdash P \triangleright \Delta \cdot s[1] : T_1 \ldots s[n] : T_n}{\Gamma \vdash (\nu\ s)P \triangleright \Delta} \quad [\text{SRes}] \qquad \Gamma \cdot X : \Delta \vdash X \triangleright \Delta \quad [\text{Var}]$$

$$\frac{\Gamma \cdot X : \Delta \vdash P \triangleright \Delta}{\Gamma \vdash \mu X.P \triangleright \Delta} \quad [\text{Rec}]$$

Figure 6.8: Typing System for Synchronous Multiparty Session Calculus

expects the recursive process to have the same type as the recursive variable inside. Finally the inactive process **0** is typed with the complete typing environment, where every session role is mapped to the inactive local type `end`.

6.2.4 Type soundness

Next we define the reduction semantics for local types. Since session environments represent the forthcoming communications, session environments can change when processes are reduced. This can be formalised as in [HYC08, B+08] by introducing the notion of reduction of session environments, whose rules are:

Definition 6.2.6 (Session Environment Reduction)**.**

1. $\{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle;T \cdot s[\mathtt{q}] : [\mathtt{p}]?(U);T'\} \longrightarrow \{s[\mathtt{p}] : T \cdot s[\mathtt{q}] : T'\}$.
2. $\{s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_i\}_{i\in I} \cdot s[\mathtt{q}] : [\mathtt{p}]\&\{l_j : T'_j\}_{j\in J}\} \longrightarrow \{s[\mathtt{p}] : T_k \cdot s[\mathtt{q}] : T'_k\}\ I \subseteq J, k \in I$.
3. $\Delta \cup \Delta' \longrightarrow \Delta \cup \Delta''$ if $\Delta' \longrightarrow \Delta''$.

Session types are reduced upon an interaction with their dual counterpart. Note that $\Delta \longrightarrow^* \Delta'$ is non-deterministic (i.e. not always confluent) by the second rule.

The following theorem is proved in [B+08].

Theorem 6.2.1 (Subject reduction)**.** Let $\Gamma \vdash P \triangleright \Delta$ be coherent and $P \twoheadrightarrow P'$ then $\Delta \twoheadrightarrow \Delta'$ and $\Gamma \vdash P' \triangleright \Delta'$ is coherent.

Proof. The proof is a standard subject reduction proof for session types, where we use induction on the length of $\twoheadrightarrow$. See Appendix C.2.1. □

6.2.5 Labelled Transition System

We present the labelled transition semantics for the synchronous MSP. Along with the untyped LTS we present a labelled transition system on environments (Γ, Δ), following the behaviour of local session types. The environment LTS is used to control the transition behaviour of typed processes, with respect to local session typing. We begin with the definition of the action labels for the LTS.

Labels: We use the following labels, range over $\ell, \ell', ...$:

Definition 6.2.7 (Labels)**.**

$$\begin{array}{rcl} \ell & ::= & \overline{a}[A](s) \mid a[A](s) \mid s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \mid s[\mathtt{p}][\mathtt{q}]!(a) \\ & \mid & s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}']) \mid s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \mid s[\mathtt{p}][\mathtt{q}] \oplus l \mid s[\mathtt{p}][\mathtt{q}]\&l \mid \tau \end{array}$$

A participant set A is a set of multiparty session participants. Labels $\overline{a}[A](s)$ and $a[A](s)$ define the accept and request, respectively, of a fresh session s by roles in set A. Actions on session channels are denoted with labels $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s[\mathtt{p}][\mathtt{q}]?\langle v\rangle$ for output and input of value v from $\mathtt{p}$ to $\mathtt{q}$ on session s. Bound output values can be shared channels or session roles (delegation). Labels $s[\mathtt{p}][\mathtt{q}] \oplus l$ and $s[\mathtt{p}][\mathtt{q}]\&l$ define the selection and branching respectively. Label τ is the standard hidden transition.

Dual label definition is used to define the parallel rule in the labelled transition system:

Definition 6.2.8 (Dual Labels)**.**

$$s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \asymp s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \qquad s[\mathtt{p}][\mathtt{q}]!(v) \asymp s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \qquad s[\mathtt{p}][\mathtt{q}] \oplus l \asymp s[\mathtt{q}][\mathtt{p}]\&l$$

Dual labels are input and output (resp. selection and branching) on the same session channel and on complementary roles. For example, in $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s[\mathtt{q}][\mathtt{p}]?\langle v\rangle$, role $\mathtt{p}$ sends to $\mathtt{q}$ and

role q receives from p. Another important definition for the session initiation is the notion of the complete role set.

Definition 6.2.9. *We say the role set A is* complete with respect to n *if* $n = \mathtt{max}(A)$ *and* $A = \{1, 2, \ldots, n\}$.

The complete participant set means that all global protocol participants are present in the set. For example, $\{1,3,4\}$ is not complete, but $\{1,2,3,4\}$ is.

We use $\mathtt{fn}(\ell)$ and $\mathtt{bn}(\ell)$ to denote a set of free and bound names in ℓ and set $\mathtt{n}(\ell) = \mathtt{bn}(\ell) \cup \mathtt{fn}(\ell)$.

Untyped Labelled Transition System

Figure 6.9 gives the untyped labelled transition system. Rules ⟨Req⟩ and ⟨Acc⟩ define the accept and request actions for a fresh session s on participants set $\{\mathtt{p}\}$. Rules ⟨Send⟩ and ⟨Rcv⟩ give the send and receive respectively for value v from role p to role q in session s. Similarly rules ⟨Sel⟩ and ⟨Bra⟩ define selecting and branching labels.

The last three rules are for collecting and synchronising the multiparty participants together. Rule ⟨AccPar⟩ accumulates the accept participants and records them into role set A. Rule ⟨ReqPar⟩ accumulates the accept participants and the request participant into role set A. Note that the request action role set always includes the maximum role number among the participants. Finally, rule ⟨TauS⟩ checks that a role set is complete, thus a new session can be created under the τ-action (synchronisation). The rest of the rules are standard π-calculus LTS rules. Rule ⟨Tau⟩ performs a τ action on a parallel composition if the parallel components exhibit symmetric actions. If any bounded names are observed, then they are restricted in the entire parallel composition. Rule ⟨Par⟩ implies that an action ℓ observed on a process P, can be also observed on process $P \mid Q$, provided that the bound names of ℓ do not occur free in Q. Rules ⟨OpenN⟩ and ⟨OpenS⟩ describe scope opening for shared and session names respectively.

$$\langle\text{Req}\rangle \quad \overline{a}[\mathtt{p}](x).P \xrightarrow{\overline{a}[\{\mathtt{p}\}](s)} P\{s[\mathtt{p}]/x\} \qquad \langle\text{Acc}\rangle \quad a[\mathtt{p}](x).P \xrightarrow{a[\{\mathtt{p}\}](s)} P\{s[\mathtt{p}]/x\}$$

$$\langle\text{Send}\rangle \quad s[\mathtt{p}][\mathtt{q}]!\langle e\rangle;P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} P \quad (e\downarrow v) \qquad \langle\text{Rcv}\rangle \quad s[\mathtt{p}][\mathtt{q}]?(x);P \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v\rangle} P\{v/x\}$$

$$\langle\text{Sel}\rangle \quad s[\mathtt{p}][\mathtt{q}]\oplus l;P \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} P \qquad \langle\text{Bra}\rangle \quad s[\mathtt{p}][\mathtt{q}]\&\{l_i:P_i\}_{i\in I} \xrightarrow{s[\mathtt{p}][\mathtt{q}]\&l_k} P_k$$

$$\langle\text{Tau}\rangle \quad \frac{P \xrightarrow{\ell} P' \quad Q \xrightarrow{\ell'} Q' \quad \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\ \mathtt{bn}(\ell)\cap\mathtt{bn}(\ell'))(P' \mid Q')} \qquad \langle\text{Par}\rangle \quad \frac{P \xrightarrow{\ell} P' \quad \mathtt{bn}(\ell)\cap\mathtt{fn}(Q)=\emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q}$$

$$\langle\text{Res}\rangle \ \frac{P \xrightarrow{\ell} P' \quad n \notin \mathtt{fn}(\ell)}{(\nu\ n)P \xrightarrow{\ell} (\nu\ n)P'} \qquad \langle\text{OpenS}\rangle \ \frac{P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}']\rangle} P'}{(\nu\ s')P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} P'} \qquad \langle\text{OpenN}\rangle \ \frac{P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle a\rangle} P'}{(\nu\ a)P \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} P'}$$

$$\langle\text{Alpha}\rangle \quad \frac{P \equiv_\alpha P' \quad P' \xrightarrow{\ell} Q'}{P \xrightarrow{\ell} Q} \qquad \langle\text{AcPar}\rangle \quad \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{a[A'](s)} P_2' \quad A\cap A'=\emptyset}{P_1 \mid P_2 \xrightarrow{a[A\cup A'](s)} P_1' \mid P_2'}$$

$$\langle\text{ReqPar}\rangle \quad \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{\overline{a}[A'](s)} P_2' \quad A\cap A'=\emptyset,\ A\cup A' \text{ not complete w.r.t } \max(A')}{P_1 \mid P_2 \xrightarrow{\overline{a}[A\cup A'](s)} P_1' \mid P_2'}$$

$$\langle\text{TauS}\rangle \quad \frac{P_1 \xrightarrow{a[A](s)} P_1' \quad P_2 \xrightarrow{\overline{a}[A'](s)} P_2' \quad A\cap A'=\emptyset,\ A\cup A' \text{ complete w.r.t } \max(A')}{P_1 \mid P_2 \xrightarrow{\tau} (\nu\ s)(P_1' \mid P_2')}$$

We omit the synmetric case of $\langle\text{Par}\rangle$ and conditonals.

Figure 6.9: Labelled transition system for processes

Rule $\langle\text{Alpha}\rangle$ closes the transition relation under structural equivalence. We write $\Longrightarrow$ for the reflexive and transitive closure of $\longrightarrow$, $\overset{\ell}{\Longrightarrow}$ for the transitions $\Longrightarrow\xrightarrow{\ell}\Longrightarrow$ and $\overset{\hat{\ell}}{\Longrightarrow}$ for $\overset{\ell}{\Longrightarrow}$ if $\ell \neq \tau$ otherwise $\Longrightarrow$.

Labelled Transition System for Environments

We use labels ℓ to introduce the definition of an environment labelled transition system $\xrightarrow{\ell}$ on the environment structure (Γ,Δ), defined in Figure 6.10. $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$ means that an environment (Γ,Δ) allows an action to take place, and the resulting environment is (Γ',Δ').

$$\Gamma(a) = \langle G \rangle, s \text{ fresh implies } (\Gamma, \Delta) \xrightarrow{a[A](s)} (\Gamma, \Delta \cdot \{s[i] : G\lceil i\}_{i \in A})$$

$$\Gamma(a) = \langle G \rangle, s \text{ fresh implies } (\Gamma, \Delta) \xrightarrow{\overline{a}[A](s)} (\Gamma, \Delta \cdot \{s[i] : G\lceil i\}_{i \in A})$$

$$\Gamma \vdash v : U, s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (\Gamma, \Delta \cdot s[\mathtt{p}] : T)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta), a \notin \mathtt{dom}(\Gamma) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [q]!\langle U \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} (\Gamma \cdot a : U, \Delta \cdot s[\mathtt{p}] : T)$$

$$\Gamma \vdash v : U, s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(U); T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (\Gamma, \Delta \cdot s[\mathtt{p}] : T)$$

$$a \notin \mathtt{dom}(\Gamma), s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(U); T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle a \rangle} (\Gamma \cdot a : U, \Delta \cdot s[\mathtt{p}] : T)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s'[\mathtt{p}'] : T' \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T' \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta \cdot s[\mathtt{p}] : T)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T' \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot \{s'[\mathtt{p}_i] : T_i\})$$

$$s[\mathtt{q}], s'[\mathtt{p}'] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(T'); T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle s'[\mathtt{p}'] \rangle} (\Gamma, \Delta \cdot s'[\mathtt{p}'] : T' \cdot s[\mathtt{p}] : T)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_i\}_{i \in I}) \xrightarrow{s[\mathtt{p}][\mathtt{q}] \oplus l_k} (\Gamma, \Delta \cdot s[\mathtt{p}] : T_k)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \text{ implies } (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}] \& \{l_i : T_i\}_{i \in I}) \xrightarrow{s[\mathtt{p}][\mathtt{q}] \& l_k} (\Gamma, \Delta \cdot s[\mathtt{p}] : T_k)$$

$$\Delta \longrightarrow \Delta' \text{ or } \Delta = \Delta' \text{ implies } (\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta')$$

Figure 6.10: Labelled Transition Relation for Environments

The basic intuition for this labelled system is that observables on session channels occur when the corresponding endpoint is not present in the linear typing environment Δ, and when the type of the object of the action respects the environment (Γ, Δ). In the cases when new names are created or received the environment (Γ, Δ) is extended.

The first rule says that reception of a message via a is possible when a's type $\langle G \rangle$ is recorded into Γ and the resulting session environment records projected types from G ($\{s[i] : G\lceil i\}_{i \in A}$). The second rule is for the send of a message via a and it is dual to the first rule. The next four

rules are free value output, bound name output, free value input and name input. The rest of the rules are free session output, bound session output, and session input as well as selection and branching rules. The bound session output records a set of session types $s'[\mathrm{p}_i]$ at opened session s'. The final rule ($\ell = \tau$) follows the reduction rules for linear session environment defined in § 6.2.4 ($\Delta = \Delta'$ is the case for the reduction at hidden sessions). Note that if Δ already contains destination ($s[\mathrm{q}]$), the environment cannot perform the visible action, but only the final τ-action.

Typed Labelled Transition System

We define a typed labelled transition system using the environment transition to control the untyped labelled transition on typed processes. The typed LTS requires that a process can perform an untyped action ℓ and that its typing environment (Γ, Δ) can match the action ℓ.

Definition 6.2.10 (Typed transition)**.** The *typed transition* relation is defined as:

$$\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$$

if

(1) $P_1 \xrightarrow{\ell} P_2$ and (2) $(\Gamma_1, \Delta_1) \xrightarrow{\ell} (\Gamma_2, \Delta_2)$

with $\Gamma_1 \vdash P_1 \triangleright \Delta_1, \Gamma_2 \vdash P_2 \triangleright \Delta_2$.

6.3 Asynchronous Multiparty Session Calculus

In this section we use the definition of the Synchronous MSP (§6.2) as a reference calculus to define three versions of the asynchronous Multiparty Session π-calculus: i) the output Asynchronous MSP; ii) the input Asynchronous MSP; and iii) the input/output Asynchronous MSP.

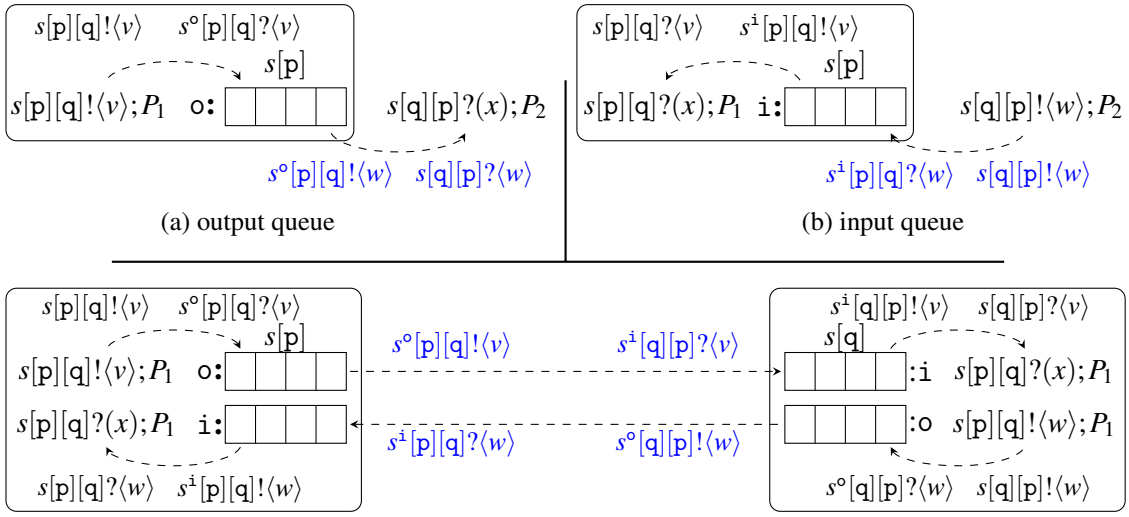

Figure 6.11: Three asynchronous semantics

For their definition we use the construct of session endpoint configuration (see § 3.1). A session endpoint configuration is a first-in, first-out intermediate communication buffer, used to store sent values before they are being received, in order to simulate asynchrony. The semantic definition of the interaction between multiparty session channels and session endpoint configurations allows us to control the input and output meaning of the asynchronous communication at the same time offer a coarse-grained and a fine-grained version of the asynchronous communication.

Figure 6.11 clarifies the last paragraph through a schematic representation of the three asynchronous semantics. The reader can intuitively assume syntax and label transition semantics for processes and session endpoint configurations. The formal definition is given later in this section.

Output asynchrony is described in (a) where the output queue is located in the process side. Following the arrows, an output message v is first enqueued by the sender process $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle;P$ in the local output queue at endpoint $s[\mathtt{p}]$, which intuitively represents a communication pipe extending from the sender's locality to the receiver's. Hence an external observer can observe the output from the queue and input to the receiver, if we assume that enqueuing actions

within a location are local to each process and are therefore invisible (τ-actions) to external observers. The observable actions are coloured by blue. The output asynchrony MSP definition is essentially the same with the system described in [B+08]. On the dual side, input asynchrony is described in case (b). Following the arrows an external observer can observe the session configuration input action $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle w\rangle$. The reception of the value by process $s[\mathtt{p}][\mathtt{q}]?(x);P_1$ is done locally and therefore invisible to the external observer. A more fine-grained asynchrony is captured by the diagram in (c) where output and input asynchrony are combined. The communication medium (i.e. the connection) is responsible for transporting the message from the sender's locality to the receiver's locality, formalised as a message transfer from the sender's output queue (at $s[\mathtt{p}]$) to the receiver's input queue. For the receiver process, the message can only be received after this transfer takes place. In (c), both dequeuing and enqueuing actions within a location are local to each process and invisible to external observers.

We proceed with the definition of the Asynchronous Multiparty Session π-Calculi. We are going to present all three calculi by extending in a uniform way the definition of the Synchronous Multiparty Session π-Calculus in § 6.2.

6.3.1 Syntax and Operational Semantics

We extend the syntax of the Synchronous MSP in Figure 6.2 (description in § 6.2.1) with the following session endpoint configuration terms:

$$
\begin{array}{lcll}
P & ::= & \vdots & \\
 & & s[\mathtt{p}][\mathtt{o}:\vec{h}] & (\text{ConfigurationO}) \\
 & \mid & s[\mathtt{p}][\mathtt{i}:\vec{h}] & (\text{ConfigurationI}) \\
h & ::= & [\mathtt{p}](v) \mid [\mathtt{p}]l \mid [\mathtt{p}](s[\mathtt{q}]) & (\text{Message})
\end{array}
$$

We define the output session endpoint configuration $s[\mathtt{p}][\mathtt{o} : \vec{h}]$ and the input session endpoint configuration $s[\mathtt{p}][\mathtt{i} : \vec{h}]$. Both session configurations are used to store message vectors. A message is denoted as h and it is either a value v, a label l, or a session role $s[\mathtt{p}]$ along with a participant $\mathtt{p}$. The participant in the case of the output queue denotes the receiver of the message and in the case of the input queue the sender of the message.

We distinguish the syntax between the three calculi:

- The **Output Asynchronous MSP** syntax uses only the output configuration session endpoint $s[\mathtt{p}][\mathtt{o} : \vec{h}]$.
- The **Input Asynchronous MSP** syntax uses only the input configuration session endpoint $s[\mathtt{p}][\mathtt{i} : \vec{h}]$.
- The **Input/Output Asynchronous MSP** syntax uses both configuration session endpoints $s[\mathtt{p}][\mathtt{o} : \vec{h}]$ and $s[\mathtt{p}][\mathtt{i} : \vec{h}]$.

A runtime process is a closed asynchronous MSP term that contains session endpoint configurations, while a program is a closed asynchronous MSP term without session endpoint configurations and free session names.

Structural Congruence

We extend the structural congruence definition in Figure 6.3 (description in §6.2.1) with the following rules:

$$\vdots$$

$$s[\mathtt{p}][\mathtt{o} : \vec{h} \cdot [\mathtt{q}](v) \cdot [\mathtt{q}'](v') \cdot \vec{h}] \equiv s[\mathtt{p}][\mathtt{o} : \vec{h} \cdot [\mathtt{q}'](v') \cdot [\mathtt{q}](v) \cdot \vec{h}] \qquad \mathtt{q} \neq \mathtt{q}'$$

$$s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}](v) \cdot [\mathtt{q}'](v') \cdot \vec{h}] \equiv s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}'](v') \cdot [\mathtt{q}](v) \cdot \vec{h}] \qquad \mathtt{q} \neq \mathtt{q}'$$

$$(\nu\, s[\mathtt{p}])(s[\mathtt{p}][\mathtt{o} : \varepsilon]) \equiv \mathbf{0}$$

$$(\nu\, s[\mathtt{p}])(s[\mathtt{p}][\mathtt{i} : \varepsilon]) \equiv \mathbf{0}$$

We capture asynchrony using the first two rules. To achieve output (resp. input) asynchrony we allow permutation of values inside output (resp. input) session endpoint configurations, if the receiver (resp. sender) of the value is different. This condition allows asynchrony between the communication of different session participants and maintains the order-preserving property inside a role to role communication. The last two rules are used for garbage collection of session endpoints that cannot interact anymore. Each of the three versions of the asynchronous MSP calculi is restricted to the use of the rules defined in their syntax.

Operational Semantics

The operational semantics for asynchronous MSP, clarify the use of session endpoint configurations, to achieve the different semantics for asynchrony. The definition is based on the operational semantics for the synchronous MSP, in Figure 6.4 (description in §6.2.1).

Operational Semantics for the Output Asynchronous MSP: The output asynchronous MSP assumes an output locality of the session endpoint configuration, meaning that messages that are to be sent from a session role $s[\mathtt{p}]$ are enqueued in the corresponding $s^{\circ}[\mathtt{p}]$ configuration endpoint.

$$
\begin{array}{rcll}
a[1](x).P_1 \mid \ldots \mid \overline{a}[n](x).P_n & \longrightarrow & (\nu\ s)(P_1\{s[1]/x\} \mid \ldots \mid P_n\{s[n]/x\} \mid & \\
 & & s[\mathtt{p}][\mathtt{o} : \varepsilon] \mid \ldots \mid s[n][\mathtt{o} : \varepsilon]) & [\text{Link}] \\
s[\mathtt{p}][\mathtt{q}]!\langle v\rangle;P \mid s[\mathtt{p}][\mathtt{o} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](v) \cdot \vec{h}] & [\text{Send}] \\
s[\mathtt{p}][\mathtt{q}]?(x);P \mid s[\mathtt{q}][\mathtt{o} : \vec{h} \cdot [\mathtt{p}](v)] & \longrightarrow & P\{v/x\} \mid s[\mathtt{q}][\mathtt{o} : \vec{h}] & [\text{Rcv}] \\
s[\mathtt{p}][\mathtt{q}] \oplus l;P \mid s[\mathtt{p}][\mathtt{o} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{p}][\mathtt{o} : [\mathtt{q}]l \cdot \vec{h}] & [\text{Sel}] \\
s[\mathtt{p}][\mathtt{q}]\&\{l_i : P_i\}_{i\in I} \mid s[\mathtt{q}][\mathtt{o} : [\mathtt{p}]l_k \cdot \vec{h}] & \longrightarrow & P_k \mid s[\mathtt{q}][\mathtt{o} : \vec{h}] & [\text{Bra}]
\end{array}
$$

Rule [Link] describes session initiation. All session participants should be present before each participant p synchronously reduces to create a fresh role $s[\mathtt{p}]$ and the corresponding output

session configuration $s[\mathtt{p}][\mathtt{o} : \varepsilon]$. Session communication is described as session configuration interactions. Rule [Send] describes an enqueue operation from role $s[\mathtt{p}]$ of a value v as a message $[\mathtt{q}](v)$ in session configuration $s[\mathtt{p}][\mathtt{o} : \vec{h}]$. Dually rule [Rcv] describes the dequeue operation and reception of a value v from role $s[\mathtt{q}]$ out of session configuration $s[\mathtt{q}][\mathtt{o} : \vec{h} \cdot [\mathtt{p}](va)]$. The reception happens on the substitution of value v on variable x on the continuation process of the receive action. Rule [Sel] and [Bra] send and receive labels l interacting with the session endpoints (in a similar way with rules [Send] and [Rcv] respectively) to perform select and branch operations respectively. A branch operation upon the reception of a label, decides the continuation of the process with respect to the label received. Operational semantics are completed with the standard π-calculus rules (see § 6.2.1).

Operational Semantics for the Input Asynchronous MSP: By contrast with the output asynchronous MSP, the input asynchronous MSP assumes an input locality of the session endpoint configuration. This means that messages intended to be received from a session role $s[\mathtt{p}]$ are received by the corresponding $s^{\mathtt{i}}[\mathtt{p}]$ configuration endpoint.

$$
\begin{array}{rcll}
a[1](x).P_1 \mid \ldots \mid \overline{a}[n](x).P_n & \longrightarrow & (\nu\, s)(P_1\{s[1]/x\} \mid \ldots \mid P_n\{s[n]/x\} \mid & \\
 & & s[\mathtt{p}][\mathtt{i} : \varepsilon] \mid \ldots \mid s[n][\mathtt{i} : \varepsilon]) & [\text{Link}] \\
s[\mathtt{p}][\mathtt{q}]!\langle v\rangle; P \mid s[\mathtt{q}][\mathtt{i} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{q}][\mathtt{i} : [\mathtt{p}](v) \cdot \vec{h}] & [\text{Send}] \\
s[\mathtt{p}][\mathtt{q}]?(x); P \mid s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}](v)] & \longrightarrow & P\{v/x\} \mid s[\mathtt{p}][\mathtt{i} : \vec{h}] & [\text{Rcv}] \\
s[\mathtt{p}][\mathtt{q}] \oplus l; P \mid s[\mathtt{q}][\mathtt{i} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{q}][\mathtt{i} : [\mathtt{p}] l \cdot \vec{h}] & [\text{Sel}] \\
s[\mathtt{p}][\mathtt{q}] \& \{l_i : P_i\}_{i \in I} \mid s[\mathtt{p}][\mathtt{i} : [\mathtt{q}] l_k \cdot \vec{h}] & \longrightarrow & P_k \mid s[\mathtt{p}][\mathtt{i} : \vec{h}] & [\text{Bra}]
\end{array}
$$

The key difference from the output asynchronous MSP operational semantics is the use of input session configurations. Rule [Link] apart from session initiation, creates the corresponding input configurations $s[\mathtt{p}][\mathtt{i} : \varepsilon]$. Session communication is done on the input session configuration basis. A participant $\mathtt{p}$ receives values from the corresponding configuration $s[\mathtt{p}][\mathtt{i} : \varepsilon]$

(in contrast with output asynchronous MSP), while it sends a value to the corresponding receiving $s[\mathtt{q}][\mathtt{i} : \varepsilon]$ configurations. Rule [Send] describes an enqueue operation from role $s[\mathtt{p}]$ of a value v as a message $[\mathtt{p}](v)$ in session configuration $s[\mathtt{q}][\mathtt{i} : \vec{h}]$. Dually rule [Rcv] describes the dequeue operation and reception of a value v from role $s[\mathtt{q}]$ out of session configuration $s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}](va)]$. The reception happens on the substitution of value v on variable x on the continuation process of the receive action. There is a similar use for labels for rules [Sel] and [Bra]. The rest of the rules are standard π-calculus rules (similar to the synchronous MSP in § 6.2.1).

Operational Semantics for the Input/Output Asynchronous MSP: The input/output asynchronous MSP combines both input and output localities for input and output session configurations. Messages being send from $s[\mathtt{p}]$ are locally stored in session configuration $s^{\circ}[\mathtt{p}]$ while messages are received from session configuration $s^{\mathtt{i}}[\mathtt{q}]$.

$$
\begin{array}{rcll}
a[1](x).P_1 \mid \ldots \mid \overline{a}[n](x).P_n & \longrightarrow & (\nu\, s)(P_1\{s[1]/x\} \mid \ldots \mid P_n\{s[n]/x\} \mid & \\
 & & s[\mathtt{p}][\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid \ldots \mid s[n][\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) & \text{[Link]} \\
s[\mathtt{p}][\mathtt{q}]!\langle v\rangle; P \mid s[\mathtt{p}][\mathtt{o} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](v) \cdot \vec{h}] & \text{[Send]} \\
s[\mathtt{p}][\mathtt{q}]?(x); P \mid s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}](v)] & \longrightarrow & P\{v/x\} \mid s[\mathtt{p}][\mathtt{i} : \vec{h}] & \text{[Rcv]} \\
s[\mathtt{p}][\mathtt{q}] \oplus l; P \mid s[\mathtt{p}][\mathtt{o} : \vec{h}] & \longrightarrow & P \mid s[\mathtt{p}][\mathtt{o} : [\mathtt{q}]l \cdot \vec{h}] & \text{[Sel]} \\
s[\mathtt{p}][\mathtt{q}]\&\{l_i : P_i\}_{i \in I} \mid s[\mathtt{p}][\mathtt{i} : \vec{h} \cdot [\mathtt{q}]l_k] & \longrightarrow & P_k \mid s[\mathtt{p}][\mathtt{i} : \vec{h}] & \text{[Bra]} \\
s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](v) \cdot \vec{h}] \mid s[\mathtt{q}][\mathtt{i} : \vec{h}'] & \longrightarrow & s[\mathtt{p}][\mathtt{o} : \vec{h}] \mid s[\mathtt{q}][\mathtt{i} : [\mathtt{p}](v) \cdot \vec{h}'] & \text{[Comm]}
\end{array}
$$

The message typing system for the Input/Output Asynchronous MSP is a combination of the message typing systems for the output and the input Asynchronous MSP. The [Link] rule creates both input and output session endpoint configurations. Rules for sending [Send] and [Sel] are identical with the corresponding rules for output message type rules and rules for receiving [Rcv] and [Bra] are identical with the corresponding rules for input message type rules.

The key difference is rule [Comm] where it describes the interaction between session endpoints (session participant locality) for the exchange of a message. Notably session endpoint $s[\mathtt{p}][\mathtt{o} : h]$ dequeues a message $[\mathtt{q}](v)$ and enqueues it in the corresponding $s[\mathtt{q}][\mathtt{i} : h']$ session endpoint. The rest of the rules are standard π-calculus rules (similar to the synchronous MSP in § 6.2.1).

6.3.2 Typing for Asynchronous Multiparty Session π-calculus

In this subsection the synchronous MSP type theory in § 6.2.2 and § 6.2.3 , is extended to define a type theory for the asynchronous MSP. The main focus of this section is the development of a *runtime typing system* (see § 3.2.4, § 4.2.4), for the typing of session endpoint configurations.

The typing foundations and typing system for programs for the Asynchronous MSP are identical with the typing system for the Synchronous MSP typing system. More specifically we assume the definitions for global and local types and their projections in § 6.2.2. We assume the typing judgements from § 6.2.3:

$$\Gamma \vdash e : S \quad \text{and} \quad \Gamma \vdash P \triangleright \Delta$$

and the typing system in Figure 6.8 (description in § 6.2.3), with the exception of the rule [SRes], is used to type program terms for the Asynchronous MSP calculi. Note that rule [SRes] is defined as a runtime typing rule in the Asynchronous MSP typing semantics.

Essentially the typing semantics for the output Asynchronous MSP are identical to the system developed in [B+08] and the semantics for the other two Asynchronous MSP calculi are variations.

6.3.3 Runtime Typing for Asynchronous Multiparty Session π-calculus

A runtime process is a closed asynchronous multiparty session π-calculus term. We extend the typing system for programs to type session endpoint configurations. Note that for each of the asynchronous calculi we use only the minimum definitions that respect its syntax.

We extend the linear session environment Δ with the message type M:

$$\Delta \ ::= \ \Delta \cdot c[\mathtt{p}] : T \ \mid \ s^{\mathtt{o}}[\mathtt{p}] : M \ \mid \ s^{\mathtt{i}}[\mathtt{p}] : M \ \mid \ \emptyset$$

where

M	$::=$	$M_{\mathtt{o}} \mid M_{\mathtt{i}}$	Message Type
$M_{\mathtt{o}}$	$::=$	$[\mathtt{q}]!U;M_{\mathtt{o}} \mid [\mathtt{q}] \oplus l;M_{\mathtt{o}} \mid \emptyset$	Output Message Type
$M_{\mathtt{i}}$	$::=$	$\emptyset \mid [\mathtt{q}]?U;M_{\mathtt{i}} \mid [\mathtt{q}]\&l;M_{\mathtt{i}} \mid \emptyset$	Input Message Type

Δ is extended to include configuration endpoints $s^{\mathtt{o}}[\mathtt{p}], s^{\mathtt{i}}[\mathtt{q}]$ types, which is notation $s^{\mathtt{o}}[\mathtt{p}]$ mapped to the message type M. A message type is defined as a sequence of output message types, which are $[\mathtt{q}]!U$ and select message types $[\mathtt{q}] \oplus l$, or a sequence of input message types, which are $[\mathtt{q}]?U$ and branch message types $[\mathtt{q}]\&l$.

The Output Asynchronous MSP typing syntax uses only the output message type syntax, while on the dual side the input Asynchronous MSP typing syntax uses the input message type syntax. The input/output Asynchronous MSP typing syntax uses the entire definition.

We define a permutation relation as the smallest congruence over message types using the rules:

Definition 6.3.1 (Message Type Permutation)**.**

$$
\begin{array}{rcl}
M;[\mathtt{p}]!U;[\mathtt{q}]!U';M' & \approx & M;[\mathtt{q}]!U';[\mathtt{p}]!U;M' \\
M;[\mathtt{p}]\oplus l_i;[\mathtt{q}]\oplus l_j;M' & \approx & M;[\mathtt{q}]\oplus l_j;[\mathtt{p}]\oplus l_i;M' \\
M;[\mathtt{p}]!U;[\mathtt{q}]\oplus l;M' & \approx & M;[\mathtt{q}]\oplus l;[\mathtt{p}]!U;M' \\
 & & \\
M;[\mathtt{p}]?U;[\mathtt{q}]?U';M' & \approx & M;[\mathtt{q}]?U';[\mathtt{p}]?U;M' \\
M;[\mathtt{p}]\&l_i;[\mathtt{q}]\&l_j;M' & \approx & M;[\mathtt{q}]\&l_j;[\mathtt{p}]\&l_i;M' \textit{ if} \\
M;[\mathtt{p}]?U;[\mathtt{q}]\&l;M' & \approx & M;[\mathtt{q}]\&l;[\mathtt{p}]?U;M'
\end{array}
$$

The $\approx$ relation is a congruence on message type permutations. A message type (both input and output) sequence can be permuted on two message types if they have different recipients.

We define a concatenation operator $*$ between message types M and local types T. The result of the concatenation operator is a local type T. We use the concatenation operator to reconstruct a complete local type $T = M * T'$out of the local type $s[\mathtt{p}] : T'$ of a process and the message type $s^{\circ}[\mathtt{p}] : M$ of an output session configuration and/or $s^{\mathtt{i}}[\mathtt{p}] : M$ of an input session configuration.

Definition 6.3.2 (Message Type Concatenation)**.**

$$
\begin{array}{rcl}
\emptyset * T & = & T \\
[\mathtt{q}]!U;M_{\mathtt{o}} * T & = & [\mathtt{q}]!\langle U\rangle;(M_{\mathtt{o}} * T) \\
[\mathtt{q}] \oplus l_k;M_{\mathtt{o}} * T & = & [\mathtt{q}] \oplus \{l_i : M_{\mathtt{o}} * T\}_{i\in I}, \quad k \in I \\
[\mathtt{q}]?U;M_{\mathtt{i}} * [\mathtt{q}]?(U);T & = & M_{\mathtt{i}} * T \\
[\mathtt{q}]\&l_k;M_{\mathtt{i}} * [\mathtt{q}]\&\{l_i : T_i\}_{i\in I} & = & M_{\mathtt{i}} * T
\end{array}
$$

$$
\begin{array}{rcll}
M_{\mathtt{i}} * [\mathtt{q}]!\langle U\rangle;T & = & [\mathtt{q}]!\langle U\rangle;(M_{\mathtt{i}} * T) & M_{\mathtt{i}} \neq [\mathtt{q}]?U;M'_{\mathtt{i}}, M_{\mathtt{i}} \neq [\mathtt{q}]\&l;M'_{\mathtt{i}} \\
M_{\mathtt{i}} * [\mathtt{q}] \oplus \{l_i : T_i\}_{i\in I} & = & [\mathtt{q}] \oplus \{l_i : M_{\mathtt{i}} * T_i\}_{i\in I} & M_{\mathtt{i}} \neq [\mathtt{q}]?U;M'_{\mathtt{i}}, M_{\mathtt{i}} \neq [\mathtt{q}]\&l;M'_{\mathtt{i}} \\
M_{\mathtt{i}} * [\mathtt{q}]?(U);T & = & [\mathtt{q}]?(U);(M_{\mathtt{i}} * T) & M_{\mathtt{i}} \neq [\mathtt{q}]?U';M'_{\mathtt{i}} \\
M_{\mathtt{i}} * [\mathtt{q}]\&\{l_i : T_i\}_{i\in M_{\mathtt{i}}} & = & [\mathtt{q}]\&\{l_i : M_{\mathtt{i}} * T_i\}_{i\in I} & M_{\mathtt{i}} \neq [\mathtt{q}]\&l;M'_{\mathtt{i}}
\end{array}
$$

The message type concatenation operator $*$, is defined separately for output and input message types, inductively on the structure of local and message types. The empty message type has no effect when concatenated with a local type. The concatenation of an output prefixed message type $[\mathtt{p}]!U;M$ and a local type T results in a type prefixed with $[\mathtt{p}]!U$ and continued inductively with the type $M * T$. Similarly for $[\mathtt{p}] \oplus l_k$ where the resulting type is a selection set $[\mathtt{p}] \oplus \{l_i : M * T\}_{i\in I}$ with $k \in I$. On the input message side a concatenation consumes matching input prefixes between the input message type and the input prefixed local type. If the two prefixes do not match then the concatenation algorithm proceeds inductively on the structure of T.

Runtime Typing System

We proceed with the definition of the runtime typing system for each of the three asynchronous MSP calculi.

Runtime Typing System for the Output Asynchronous MSP

$$\Gamma \vdash s[\mathtt{p}][\mathtt{o} : \varepsilon] \triangleright s^{\circ}[\mathtt{p}] : \emptyset \qquad (\text{QEmptyO})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](v) \cdot \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : [\mathtt{q}]!S;M} \qquad (\text{QVal})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](s'[\mathtt{q}']) \cdot \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : [\mathtt{q}]!T;M} \qquad (\text{QDel})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{o} : [\mathtt{q}]l \cdot \vec{h}] \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : [\mathtt{q}] \oplus l;M} \qquad (\text{QSel})$$

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \quad \Gamma \vdash P_2 \triangleright \Delta_2 \quad \mathtt{dom}(\Delta_1) \cup \mathtt{dom}(\Delta_2) = \emptyset}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \cdot \Delta_2} \qquad (\text{QConc})$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : M \quad M \approx M'}{\Gamma \vdash P \triangleright \Delta \cdot s^{\circ}[\mathtt{p}] : M'} \qquad (\text{EquivO})$$

Rule (QEmpty) maps the empty session configurations to the empty message type $\emptyset$. Rule (QVal) (and rules (QSel), (QDel)) requires an inductive typing of the session configuration $s^{\circ}[\mathtt{p}]$ without its message prefix $[\mathtt{q}](v)$ ($[\mathtt{q}]l, [\mathtt{q}](s'[\mathtt{p}'])$ respectively) to get the type mapping $s^{\circ}[\mathtt{p}] : M$. The resulting message type for $s^{\circ}[\mathtt{p}]$ is prefixed with the message type of value v together with the receiver $\mathtt{q}$ to get $s^{\circ}[\mathtt{p}] : [\mathtt{q}]!U;M$. For rule (QSel) we prefix with the select message type $[\mathtt{q}] \oplus l$ and for rule (QDel) we prefix with $[\mathtt{q}]!s'[\mathtt{p}']$. Rules (QConc) and (EquivO) are defined to type and keep consisted the runtime syntax. The parallel operator in rule (QConc) is identical to the parallel operator for programs (Figure 6.8), and requires disjoint linear session environments of the two operands. The result typing is the union of the two linear session environments. Rule (EquivO) requires that a runtime process can be typed

up-to message permutation. Session restriction rule (SRes), defined as:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s[1] : T_1 \cdot s^{\circ}[1] : M_1 \ldots s[n] : T_n \cdot s^{\circ}[n] : M_n \quad \mathtt{co}(\{s[1] : M_1 \ * \ T_1 \ldots s[n] : M_n \ * \ T_n\})}{\Gamma \vdash (\nu\ s)P \triangleright \Delta} \ (\text{SRes})$$

has a distinct definition between the three asynchronous MSP calculi. The rule first require to construct the local types for all session roles $s[\mathtt{p}]$ using the $*$ operator to concatenate the message type $s^{\circ}[\mathtt{p}]M$ and $s[\mathtt{p}]T$ and then check that the resulting local types to be coherent.

Runtime Typing System for the Input Asynchronous MSP:

$$\Gamma \vdash s[\mathtt{p}][\mathtt{i} : \varepsilon] \triangleright s^{\mathtt{i}}[\mathtt{p}] : \emptyset \qquad (\text{QEmptyI})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : [\mathtt{q}](v) \cdot \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}]?S; M} \ (\text{QRcv})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : [\mathtt{q}](s'[\mathtt{q}']) \cdot \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}]?T; M} \ (\text{QRcvS})$$

$$\frac{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : M}{\Gamma \vdash s[\mathtt{p}][\mathtt{i} : [\mathtt{q}]l \cdot \vec{h}] \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}]\&l; M} \ (\text{QBra})$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : M \quad M \approx M'}{\Gamma \vdash P \triangleright \Delta \cdot s^{\mathtt{i}}[\mathtt{p}] : M'} \ (\text{EquivI})$$

The typing system for the input asynchronous MSP is very similar to the typing system for the output asynchronous MSP. It requires the typing of configuration messages as input message types. Rules (Conc) is identical and rule (EquivI) requires message permutation in input session configurations . The session endpoint configuration typing rules follow the same principles as the rules for session endpoints configurations in the output asynchronous MSP, with the key difference that they are typed as input messages $M_{\mathtt{i}}$. The session restriction rule

(SRes), defined as:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s[1] : T_1 \cdot s^{\mathtt{i}}[1] : M_1 \ldots s[n] : T_n \cdot s^{\mathtt{i}}[n] : M_n \quad \mathsf{co}(\{s[1] : M_1 \ * \ T_1 \ldots s[n] : M_n \ * \ T_n\})}{\Gamma \vdash (\nu\ s)P \triangleright \Delta} \ (\text{SRes})$$

requires construction of the local types for all session roles for s in Δ using the input concatenation rules (contrast with the (SRes) rule for output asynchronous MSP).

Runtime Typing System for the Input/Output Asynchronous MSP: The runtime typing system for the input/output asynchronous MSP is the union of the two preceding type systems for output and input asynchronous MSP. The key difference lies in rule (SRes), defined as:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s[1] : T_1 \cdot s^{\mathtt{o}}[1] : M_1^o \cdot s^{\mathtt{i}}[1] : M_1^i \ldots s[n] : T_n \cdot s^{\mathtt{i}}[n] : M_n^o \cdot s^{\mathtt{i}}[1] : M_n^i \quad \mathsf{co}(\{s[1] : M_1^o \ * \ (M_1^i \ * \ T_1) \ldots s[n] : M_n^o \ * \ (M_n^i \ * \ T_n)\})}{\Gamma \vdash (\nu\ s)P \triangleright \Delta} \ [\text{SRes}]$$

where we expect the construction of the local types for all roles of a session name s by concatenating both message types M^o and M^i with the local type T.

6.3.4 Type Soundness

In this subsection we prove the soundness for the typing theories developed for the Asynchronous MSP, through a subject reduction theorem.

Before we proceed we define a reduction relation on session environments (see § 6.2.4, § 3.2.5).

Definition 6.3.3 (Session Environment Reduction)**.**
We define a set of Session environment reduction semantics for each of the asynchronous MSP calculi.

The basic congruence rule, is the same for all the sets of semantics:

1. $\Delta \cup \Delta' \longrightarrow \Delta \cup \Delta''$ if $\Delta' \longrightarrow \Delta''$

Output Asynchronous MSP

2. $\{s[\mathrm{p}] : [\mathrm{q}]!\langle U\rangle; T \cdot s^{\mathrm{o}}[\mathrm{p}] : M\} \longrightarrow \{s[\mathrm{p}] : T \cdot s^{\mathrm{o}}[\mathrm{p}] : [\mathrm{q}]!U; M\}$
3. $\{s[\mathrm{q}] : [\mathrm{p}]?(U); T \cdot s^{\mathrm{o}}[\mathrm{p}] : M; [\mathrm{p}]!U\} \longrightarrow \{s[\mathrm{q}] : T \cdot s^{\mathrm{o}}[\mathrm{p}] : M\}$
4. $\{s[\mathrm{p}] : [\mathrm{q}] \oplus \{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{o}}[p] : M\} \longrightarrow \{s[\mathrm{p}] : T_k \cdot s^{\mathrm{o}}[\mathrm{p}] : [\mathrm{q}] \oplus l_k; M\}$
5. $\{s[\mathrm{q}] : [\mathrm{p}]\&\{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{o}}[\mathrm{p}] : M; [\mathrm{p}] \oplus l_k\} \longrightarrow \{s[\mathrm{q}] : T_k \cdot s^{\mathrm{o}}[\mathrm{p}] : M\}$

Input Asynchronous MSP

2. $\{s[\mathrm{p}] : [\mathrm{q}]!\langle U\rangle; T \cdot s^{\mathrm{i}}[\mathrm{q}] : M\} \longrightarrow \{s[\mathrm{p}] : T \cdot s^{\mathrm{i}}[\mathrm{q}] : [\mathrm{p}]?U; M\}$
3. $\{s[\mathrm{q}] : [\mathrm{p}]?(U); T \cdot s^{\mathrm{i}}[\mathrm{q}] : M; [\mathrm{p}]?U\} \longrightarrow \{s[\mathrm{q}] : T \cdot s^{\mathrm{i}}[\mathrm{q}] : M\}$
4. $\{s[\mathrm{p}] : [\mathrm{q}] \oplus \{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{i}}[\mathrm{q}] : M\} \longrightarrow \{s[\mathrm{p}] : T_k \cdot s^{\mathrm{i}}[\mathrm{q}] : [\mathrm{p}] \oplus l_k; M\}$
5. $\{s[\mathrm{q}] : [\mathrm{p}]\&\{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{i}}[\mathrm{q}] : M; [\mathrm{p}] \oplus l_k\} \longrightarrow \{s[\mathrm{q}] : T_k \cdot s^{\mathrm{i}}[\mathrm{q}] : M\}$

Input/Output Asynchronous MSP

2. $\{s[\mathrm{p}] : [\mathrm{q}]!\langle U\rangle; T \cdot s^{\mathrm{o}}[\mathrm{p}] : M\} \longrightarrow \{s[\mathrm{p}] : T \cdot s^{\mathrm{o}}[\mathrm{p}] : [\mathrm{q}]!U; M\}$
3. $\{s[\mathrm{p}] : [\mathrm{q}]?(U); T \cdot s^{\mathrm{i}}[\mathrm{p}] : M; [\mathrm{q}]?U\} \longrightarrow \{s[\mathrm{p}] : T \cdot s^{\mathrm{i}}[\mathrm{p}] : M\}$
4. $\{s[\mathrm{p}] : [\mathrm{q}] \oplus \{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{o}}[\mathrm{p}] : M\} \longrightarrow \{s[\mathrm{p}] : T_k \cdot s^{\mathrm{i}}[\mathrm{p}] : [\mathrm{q}] \oplus l; M\}$
5. $\{s[\mathrm{p}] : [\mathrm{q}]\&\{l_i : T_i\}_{i\in I} \cdot s^{\mathrm{i}}[\mathrm{p}] : M; [\mathrm{q}]\&l_k\} \longrightarrow \{s[\mathrm{p}] : T_k \cdot s^{\mathrm{i}}[\mathrm{p}] : M\}$

6. $\{s^{\circ}[\mathtt{p}] : M^{o}; [\mathtt{q}]!U \cdot s^{\mathtt{i}}[\mathtt{q}] : M^{i}\} \longrightarrow \{s^{\circ}[\mathtt{p}] : M^{o} \cdot s^{\mathtt{i}}[\mathtt{q}] : [\mathtt{p}]?U; M^{i}\}$

7. $\{s^{\circ}[\mathtt{p}] : M^{o}; [\mathtt{q}] \oplus l \cdot s^{\mathtt{i}}[\mathtt{q}] : M^{i}\} \longrightarrow \{s^{\circ}[\mathtt{p}] : M^{o} \cdot s^{\mathtt{i}}[\mathtt{q}] : [\mathtt{p}]\&l; M^{i}\}$

Session types interaction for each asynchronous MSP calculus, follows the intuition of the interaction for the corresponding operational semantics. For the output asynchronous MSP a session output on role $s[\mathtt{p}]$ interacts with the corresponding $s^{\circ}[\mathtt{p}]$ for enqueuing a type, and a session input from participant $\mathtt{q}$ to role $s[\mathtt{p}]$ interacts with the corresponding $s^{\circ}[\mathtt{q}]$ to dequeue a type. On the dual side the input asynchronous MSP, session outputs from role $s[\mathtt{p}]$ towards $\mathtt{q}$, interacts with $s^{\mathtt{i}}[\mathtt{q}]$ to enqueue a type and a session input on $s[\mathtt{p}]$ interacts with $s^{\mathtt{i}}[\mathtt{p}]$. The basic reduction rule for the input/output MSP is the interaction between $s^{\circ}[\mathtt{p}]$ and $s^{\mathtt{i}}[\mathtt{q}]$ for the transition of a type from the former to the latter. The rest of the reduction rules are defined in the output and input asynchronous MSP.

It is convenient for our typing theory to include, as separate types, the local types and message types in the session typing environment. The separation of the two types implies a notion of *locality* for each session role, since we can use the separation to extract information about the asynchronous state of each session role at runtime. For the subject reduction theorem it is useful to construct the local type for a session role using its local and its message type. For this we define the following operator:

Definition 6.3.4. *Let Δ be session typing environment. We define*

$$
\begin{aligned}
*(\Delta) \;=\; & \{s[\mathtt{p}] : T \mid s[\mathtt{p}] : T \in \Delta, s^{\mathtt{i}}[\mathtt{p}], s^{\circ}[\mathtt{p}] \notin \mathtt{dom}(\Delta)\} \\
\cup\; & \{s^{\circ}[\mathtt{p}] : M \in \Delta \mid s[\mathtt{p}] \notin \mathtt{dom}(\Delta)\} \\
\cup\; & \{s^{\mathtt{i}}[\mathtt{p}] : M \in \Delta \mid s[\mathtt{p}] \notin \mathtt{dom}(\Delta)\} \\
\cup\; & \{s[\mathtt{p}] : M^{o} * T \mid s[\mathtt{p}] : T, s^{\circ}[\mathtt{p}] : M^{o} \in \Delta, s^{\mathtt{i}}[\mathtt{p}] \notin \mathtt{dom}(\Delta)\} \\
\cup\; & \{s[\mathtt{p}] : M^{i} * T \mid s[\mathtt{p}] : T, s^{\mathtt{i}}[\mathtt{p}] : M^{i} \in \Delta, s^{\circ}[\mathtt{p}] \notin \mathtt{dom}(\Delta)\} \\
\cup\; & \{s[\mathtt{p}] : M^{o} * M^{i} * T \mid s[\mathtt{p}] : T, s^{\mathtt{i}}[\mathtt{p}] : M^{i}, s^{\circ}[\mathtt{p}] : M^{o} \in \Delta\}
\end{aligned}
$$

The $*(\Delta)$ operator reconstructs the local types for the session roles inside a linear session environment. It uses the $*$ operator to concatenate roles $s^{\circ}[\mathtt{p}] : M_o, s^{\mathtt{i}}[\mathtt{p}] : M_i$ with $s[\mathtt{p}] : T$ for each session role $s[\mathtt{p}]$. The resulting linear session environment is used for coherency checking in the subject reduction theorem. Note that for each of the asynchronous MSP calculi we only use the part of the definition given by its syntax and its typing system.

Note: Because we want to maintain a uniform framework for reasoning about all the MSP calculi, we set:

$$*(\Delta) = \Delta$$

for the context of the synchronous MSP calculus.

We are now ready to state a subject reduction theorem.

Theorem 6.3.1 (Subject Reduction)**.** Let $\Gamma \vdash P \triangleright \Delta$ with $\mathtt{co}(*(\Delta))$ and if $P \longrightarrow\!\!\!\!\!\rightarrow P'$ then $\Gamma \vdash P' \triangleright \Delta'$ with $\Delta \longrightarrow\!\!\!\!\!\rightarrow \Delta'$ and $\mathtt{co}(*(\Delta'))$.

Proof. The subject reduction proof follows an induction on the length of $\longrightarrow\!\!\!\!\!\rightarrow$. See Appendix C.2.2 for details. □

6.3.5 Labelled Transition System

We extend the label definition ℓ in § 6.2.5 to include action labels on input configurations:

$$\begin{array}{rcl} \ell & = & \vdots \\ & \mid & s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v\rangle \mid s^{\circ}[\mathtt{p}][\mathtt{q}]!(v) \mid s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v\rangle \mid s^{\circ}[\mathtt{p}][\mathtt{q}] \oplus l \mid s^{\circ}[\mathtt{p}][\mathtt{q}]\&l \\ & \mid & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle \mid s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(v) \mid s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v\rangle \mid s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}] \oplus l \mid s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\&l \end{array}$$

Labels are divided into actions from output session configurations and actions from input session configurations. Their intuitive meaning is the same in both cases. Labels $s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s^{\circ}[\mathtt{p}][\mathtt{q}]!(v)$ (resp. $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(v)$) denote the output of value v and output

of bound value v respectively from session configuration $s^{\circ}[\mathtt{p}]$ (resp. $s^{\mathtt{i}}[\mathtt{p}]$) to participant $\mathtt{q}$. Dually action $s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v\rangle$ (resp. $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v\rangle$) denotes the reception of value v by session configuration $s^{\circ}[\mathtt{p}]$ (resp. $s^{\mathtt{i}}[\mathtt{p}]$) sent by participant $\mathtt{q}$. Actions $s^{\circ}[\mathtt{p}][\mathtt{q}] \oplus l$ and $s^{\circ}[\mathtt{p}][\mathtt{q}]\&l$ (resp. $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}] \oplus l$ and $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\&l$) respectively describe the send (select) and receive (branch) of label l from participant $\mathtt{p}$ to participant $\mathtt{q}$.

We use the definitions for *role set* A and $\mathtt{max}(A)$ from § 6.2.5.

Label semantics are clarified, with the definition of the duality relation $\asymp$ between labels. The definition of the synchronous MSP label duality relation can be found in § 6.2.5. The duality relation is defined for each of the asynchronous MSP.

Label Duality for the Output Asynchronous MSP:

$$\begin{array}{lcllcl} s[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v\rangle & s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \\ s[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v\rangle & s^{\circ}[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s[\mathtt{q}][\mathtt{p}]?\langle v\rangle \\ s[\mathtt{p}][\mathtt{q}] \oplus l & \asymp & s^{\circ}[\mathtt{p}][\mathtt{q}]\&l & s^{\circ}[\mathtt{p}][\mathtt{q}] \oplus l & \asymp & s[\mathtt{q}][\mathtt{p}]\&l \end{array}$$

Process output actions $s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ interact with the corresponding session configuration input actions $s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v\rangle$. Similarly for process bound output. Process input actions $s[\mathtt{q}][\mathtt{p}]?\langle v\rangle$ interact with the corresponding session configuration output action ($s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v\rangle$) of the receiver participant $\mathtt{q}$. Similarly when the session configuration output action is bound. Select and branching label duality follows the value send and receive semantics.

Label Duality for the Input Asynchronous MSP:

$$\begin{array}{lcllcl} s[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle v\rangle & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \\ s[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle v\rangle & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \\ s[\mathtt{p}][\mathtt{q}] \oplus l & \asymp & s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]\&l & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}] \oplus l & \asymp & s[\mathtt{p}][\mathtt{q}]\&l \end{array}$$

Essentially label duality for the input asynchronous MSP, corresponds to the label duality for the output asynchronous MSP, by reversing the participants in a session configuration action i.e. we reverse the participants $\mathtt{p},\mathtt{q}$ in $s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle, s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!(v), s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]?\langle v\rangle, s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]\oplus l$ to get $s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]!\langle v\rangle, s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]!(v), s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle v\rangle, s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]\oplus l$.

Label Duality fot the Input/Output Asynchronous MSP:

$$
\begin{array}{rclcrcl}
s[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]?\langle v\rangle & \qquad & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \\
s[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]?\langle v\rangle & & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(v) & \asymp & s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \\
s[\mathtt{p}][\mathtt{q}]\oplus l & \asymp & s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]\&l & & s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\oplus l & \asymp & s[\mathtt{p}][\mathtt{q}]\&l \\
s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle & \asymp & s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle v\rangle & & s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]\oplus l & \asymp & s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]\&l
\end{array}
$$

The duality relation on labels is a combination of the duality between process output labels and output configuration input labels from the output asynchronous MSP definition and the duality between process input labels and input configuration output labels from the input asynchronous MSP definition. The key difference is the duality between configuration output actions and configuration input actions (e.g. $s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ and $s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle v\rangle$) for interaction between session configurations for message exchange.

Untyped Labelled Transition System

The labelled transition system in Figure 6.12 extends the synchronous MSP labelled transition system in Figure 6.9 (description in § 6.2.5) to define actions on output and input session configurations. Each of the Asynchronous MSP calculi is limited to use the part of the labelled transition system, consisted with its syntax.

We give a description for the labelled transition system for output configurations. In rule $\langle$QSendO$\rangle$ the observation of an output action $s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!\langle v\rangle$ on a non-empty output configuration queue $s^{\mathtt{o}}[\mathtt{p}]$ sends (dequeues) a value v towards an observer role $\mathtt{q}$. Dually, in rule

$$\langle \text{QSendO} \rangle \quad s[\mathtt{p}][\mathtt{o} : h \cdot [\mathtt{q}](v)] \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v \rangle} s[\mathtt{p}][\mathtt{o} : h]$$

$$\langle \text{QRcvO} \rangle \quad s[\mathtt{p}][\mathtt{o} : h] \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v \rangle} s[\mathtt{p}][\mathtt{o} : [\mathtt{q}](v) \cdot h]$$

$$\langle \text{QSelO} \rangle \quad s[\mathtt{p}][\mathtt{o} : h \cdot [\mathtt{q}]l] \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]\oplus l} s[\mathtt{p}][\mathtt{o} : h]$$

$$\langle \text{QBraO} \rangle \quad s[\mathtt{p}][\mathtt{o} : h] \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]\& l} s[\mathtt{p}][\mathtt{o} : [\mathtt{q}]l \cdot h]$$

$$\langle \text{QSendI} \rangle \quad s[\mathtt{p}][\mathtt{i} : h \cdot [\mathtt{q}](v)] \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle v \rangle} s[\mathtt{p}][\mathtt{i} : h]$$

$$\langle \text{QRcvI} \rangle \quad s[\mathtt{p}][\mathtt{i} : h] \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v \rangle} s[\mathtt{p}][\mathtt{i} : [\mathtt{q}](v) \cdot h]$$

$$\langle \text{QSelI} \rangle \quad s[\mathtt{p}][\mathtt{i} : h \cdot [\mathtt{q}]l] \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\oplus l} s[\mathtt{p}][\mathtt{i} : h]$$

$$\langle \text{QBraI} \rangle \quad s[\mathtt{p}][\mathtt{i} : h] \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\& l} s[\mathtt{p}][\mathtt{i} : [\mathtt{q}]l \cdot h]$$

$$\langle \text{QOpenSO} \rangle \quad \frac{P \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} P'}{(\nu\, s[\mathtt{p}])P \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} P'} \qquad \langle \text{QOpenNO} \rangle \quad \frac{P \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle a \rangle} P'}{(\nu\, a)P \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!(a)} P'}$$

$$\langle \text{QOpenSI} \rangle \quad \frac{P \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle} P'}{(\nu\, s[\mathtt{p}])P \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} P'} \qquad \langle \text{QOpenNI} \rangle \quad \frac{P \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!\langle a \rangle} P'}{(\nu\, a)P \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]!(a)} P'}$$

Figure 6.12: Labelled Transition System for the Asynchrnous MSP calculi.

$\langle$QRcvO$\rangle$, an input action $s^{\circ}[\mathtt{p}][\mathtt{q}]?\langle v \rangle$ receives (enqueues) a value v from role $\mathtt{p}$. Similarly rules $\langle$QSelO$\rangle$ and $\langle$QBraA$\rangle$, describe the select and branch interactions on labels. Rules $\langle$QOpenS$\rangle$ and $\langle$QOpenN$\rangle$ respectively extend scope opening on actions $s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}'] \rangle$ and $s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle a \rangle$ with bound actions $s^{\circ}[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])$ and $s^{\circ}[\mathtt{p}][\mathtt{q}]!(a)$ respectively. Transitions on input configurations, act on the corresponding input labels, and in the same way as the output configuration lts is defined.

Localisation

The notion of localisation is presented for the Asynchronous MSP (refer to the localisation definition for the ASP in Definition 4.3.1). A localised linear typing has the property of having present all session configurations for each session name used, while a localised process is a typed process with localised linear typing.

Definition 6.3.5 (Localisation)**.**

- Let Δ be a linear session typing. Then we say that Δ is *localised* if
  for each $s[\mathtt{p}] \in \mathtt{dom}(\Delta)$, $s^{\circ}[\mathtt{p}], s^{\mathtt{i}}[\mathtt{p}] \in \Delta$.

- Let P be closed and $\Gamma \vdash P \triangleright \Delta$. Then we say P is *localised* if Δ is localised.

The requirement $s^{\circ}[\mathtt{p}], s^{\mathtt{i}}[\mathtt{p}] \in \Delta$ is relaxed depending on the linear session environment syntax of the underlying Asynchronous MSP. P is localised if for all free session roles in P then all corresponding endpoint configurations exist in P. Bound session roles are implicitly checked for locality by exploiting the fact that P is typed and rule (SRes) was used to check endpoint configuration presence.

Labelled Transition System for Environments

We define a labelled transition system for linear environments in the presence of multiparty session asynchrony. We follow the definitions developed in § 3.3 and the lts for environment for the synchronous MSP § 6.2.5.

Before we proceed with the definition of a labelled transition system for session environments, it is convenient to define a context relation for local types:

Definition 6.3.6 (Local Type Context)**.**

$$\begin{array}{rcl} \mathscr{T} & ::= & - \mid [\mathtt{p}]!\langle U\rangle;\mathscr{T} \mid [\mathtt{p}]?(U);\mathscr{T} \\ & \mid & [\mathtt{p}]\oplus\{l_i:\mathscr{T}_i\}_{i\in I} \mid [\mathtt{p}]\&\{l_i:\mathscr{T}_i\}_{i\in I} \\ & \mid & \mu X.\mathscr{T} \end{array}$$

A local type $\mathscr{T}[T]$ is defined if we replace the occurrences of $-$ with T in the context $\mathscr{T}$.

The local type context is used to define a labelled transition system for session environments (see § 6.2.5, § 3.3.1), written $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma,\Delta)$ where ℓ ranges over the labels defined for the untyped labelled transition system for asynchronous MSP.

We proceed with the definition of the session environment labelled transition system. A general point of attention is that the labelled transition system for the Asynchronous MSP calculi assumes a localised session environment, i.e. it assumes that the typing for a role $s[p]$ includes both the local type T and message types $M_{\mathtt{i}}$ and/or $M_{\mathtt{o}}$. A second point is that the environment LTS does not allow observable delegation actions. This is because a delegation action may result in a non-localised (see Definition 6.3.5) linear session environment Δ. However internal delegation can happen using the τ action. Internal delegation always results in a localised session environment. Finally we can always observe a τ action on any environment without changing its state. A third point is the fact that session actions happen when their dual counterpart in a communication is not present in the linear environment. This restriction enforces a linearity on session actions, forcing a session action to interact only with its dual counterpart.

Environment LTS for the Output Asynchronous MSP:

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{a[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\circ}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\overline{a}[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\circ}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma \vdash v : U, s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\circ}[\mathtt{p}] : M; [q]!U) \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (\Gamma, \Delta \cdot s^{\circ}[\mathtt{p}] : M)$$

$$a \notin \mathtt{dom}(\Gamma), s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\circ}[\mathtt{p}] : M; [\mathtt{q}]!U) \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}]!(a)} (\Gamma \cdot a : U, \Delta \cdot s^{\circ}[\mathtt{p}] : M)$$

$$s^{\circ}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]?(U); T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (\Gamma \cdot v : U, \Delta \cdot s[\mathtt{p}] : T)$$

$$s[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\circ}[\mathtt{p}] : M; [\mathtt{q}] \oplus l_k) \xrightarrow{s^{\circ}[\mathtt{p}][\mathtt{q}] \oplus l_k} (\Gamma, \Delta \cdot s^{\circ}[\mathtt{p}] : M)$$

$$s^{\circ}[\mathtt{q}] :\notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}] \& \{l_i : T_i\}) \xrightarrow{s[\mathtt{p}][\mathtt{q}] \& l_k} (\Gamma, \Delta \cdot s[\mathtt{p}] : T_k)$$

$$\Delta \longrightarrow \Delta' \vee \Delta = \Delta' \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta')$$

Actions $a[A](s)$ and $\overline{a}[A](s)$ extend a session environment to include the local type of the session roles included in A. Types for each role derive from local typing of a in the shared environment Γ. Action $s^{\circ}[\mathtt{p}][\mathtt{q}]!\langle v \rangle$ happens on a message type. The rule checks for the type of v in the shared environment Γ to agree with the object of the output prefix of message type for $s^{\circ}[\mathtt{p}]$. Output session actions carrying a bounded shared names $s^{\circ}[\mathtt{p}][\mathtt{q}]!(a)$ check that a shared name is not included in the shared environment Γ, with the transition to extend Γ to include the type of a. Input action is $s[\mathtt{p}][\mathtt{q}]?\langle v \rangle$ observed on an input prefixed local type. After the action, the shared environment Γ is extended to include the received value. Similar with the send and receive actions are the select and branch actions respectively, which carry labels and proceed with selecting (resp. branching) the continuation type. Hidden actions τ follow the session environment reduction for output asynchronous MSP in Definition 6.3.3. Finally we can always observe a τ action on any environment without changing its state.

Environment LTS for the Input Asynchronous MSP:

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\overline{a}[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\mathtt{i}}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{a[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\mathtt{i}}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma \vdash v : U, s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s[\mathtt{p}] : [q]!\langle U \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (\Gamma, \Delta \cdot s[\mathtt{p}] : T)$$

$$a \notin \mathtt{dom}(\Gamma), s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} (\Gamma \cdot a : U, \Delta \cdot s[\mathtt{p}] : T)$$

$$\Gamma \vdash v : U, s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle l_i : T_i \rangle;) \xrightarrow{s[\mathtt{p}][\mathtt{q}] \oplus l_k} (\Gamma \cdot v : U, \Delta \cdot s[\mathtt{p}] : T_k)$$

$$\frac{M * T = \mathscr{T}[[\mathtt{q}]?(S); T'] \quad \mathscr{T} \text{ not contain prefix on role } \mathtt{q} \quad s[\mathtt{q}] \notin \mathtt{dom}(\Delta)}{(\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : M) \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (\Gamma \cdot v : U, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}]?S; M)}$$

$$\frac{M * T = \mathscr{T}[[\mathtt{q}]\&\{l_i : T_i\}] \quad \mathscr{T} \text{ not contain prefix on role } \mathtt{q} \quad s[\mathtt{q}] \notin \mathtt{dom}(\Delta)}{(\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : M) \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]\&l_k} (\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}] \oplus l_k; M)}$$

$$\Delta \longrightarrow \Delta' \vee \Delta = \Delta' \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta')$$

As with the session environment semantics for the output asynchronous MSP, actions $a[A](s)$ and $\overline{a}[A](s)$ extend a session environment to include type mapping of the session roles included in A. In contrast with the output asynchronous MSP environment lts, action $s[\mathtt{p}][\mathtt{q}]!\langle v \rangle$ happens on a local type. Input action $s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v \rangle$ is observed on message types M. The main requirement for an input action on a session configuration is to check that the constructed local type for $s[\mathtt{p}] : T$ and $s^{\mathtt{i}}[\mathtt{p}] : M$, written $M * T$ has a message input prefix up to a local type context $\mathscr{T}$ with $\mathscr{T}$ not containing any other prefix of interaction with $\mathtt{q}$. This condition enforces input asynchrony when observing input actions on message types. After the input action, the shared environment Γ is extended to include the received value. Similar with the send and receive actions are the select and branch actions respectively. Output session actions with shared name objects and the τ action is treated similarly with the output asynchronous MSP case.

Environment LTS for the Input/Output Asynchronous MSP:

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{a[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\mathtt{o}}[\mathtt{p}] : \emptyset \cdot s^{\mathtt{i}}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma(a) = \langle G \rangle, s \text{ fresh} \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\overline{a}[A](s)} (\Gamma, \Delta \cdot \{s[\mathtt{p}] : G\lceil \mathtt{p} \cdot s^{\mathtt{o}}[\mathtt{p}] : \emptyset \cdot s^{\mathtt{i}}[\mathtt{p}] : \emptyset\}_{\mathtt{p} \in A})$$

$$\Gamma \vdash v : U, s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M; [q]!U) \xrightarrow{s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (\Gamma, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M)$$

$$a \notin \mathtt{dom}(\Gamma), s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M; [\mathtt{q}]!U) \xrightarrow{s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}]!(a)} (\Gamma \cdot a : U, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M)$$

$$s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta) \quad \text{implies} \quad (\Gamma, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M; [\mathtt{q}] \oplus l_k) \xrightarrow{s^{\mathtt{o}}[\mathtt{p}][\mathtt{q}] \oplus l_k} (\Gamma, \Delta \cdot s^{\mathtt{o}}[\mathtt{p}] : M)$$

$$\frac{M * T = \mathscr{T}[[\mathtt{q}]?(S); T'] \quad \mathscr{T} \text{ not contain prefix on role } \mathtt{q} \quad s^{\mathtt{o}}[q] \notin \mathtt{dom}(\Delta)}{(\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : M) \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (\Gamma \cdot v : U, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}]?U; M)}$$

$$\frac{T * M = \mathscr{T}[[\mathtt{q}]\&\{l_i : T_i\}] \quad \mathscr{T} \text{ not contain prefix on role } \mathtt{q} \quad s^{\mathtt{o}}[\mathtt{q}] \notin \mathtt{dom}(\Delta)}{(\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : M) \xrightarrow{s^{\mathtt{i}}[\mathtt{p}][q] \oplus l_k} (\Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{p}] : [\mathtt{q}] \oplus l_k; M)}$$

$$\Delta \longrightarrow \Delta' \vee \Delta = \Delta' \quad \text{implies} \quad (\Gamma, \Delta) \xrightarrow{\tau} (\Gamma, \Delta')$$

The environment labelled transition system for the input/output asynchronous MSP, combines the output rules for the output asynchronous MSP and the input rules for the input asynchronous MSP. The only adjustment made is on the requirement for the non-presence of the dual corresponding type.

Typed Labelled Transition System

The typed labelled transition system for each of the asynchronous MSP calculi is defined as in the definition for the typed transition relation for the synchronous MSP (Definition 6.2.10), by using the corresponding untyped and environment labelled transition systems.

6.4 Global Environment Semantics

On the way towards a bisimulation theory we develop a semantic theory for local and global types (see § 6.2.5 and § 6.3.5). Our intention is to use the semantics developed in this section, to control the transition behaviour of a process in order to define different classes of bisimulation relations. In this section we establish a semantic theory for global types. The theory allows a fine-grain control of the session environment (Γ, Δ) using a set of global types, called the *global environment*.

6.4.1 Global Environments

We formally define the *global environment*:

Definition 6.4.1. We write $E, E', ...$, called *global environment*, for a mapping from session names s to global types G:

$$E \;::=\; E \cdot s : G \;\mid\; \emptyset$$

The definition of projection (Definition 6.2.3) is extended to include global environments:

$$\mathtt{proj}(E) = \bigcup_{s:G \in E} \mathtt{proj}(s : G)$$

Labelled Reduction Relation for Global Environments

We define a labelled reduction relation, $E \xrightarrow{\ell} E'$, on global environments. The environment reduction relation corresponds to $\Delta \longrightarrow \Delta'$ in Definition 6.2.6.

To annotate the reduction relation we define the labels:

Definition 6.4.2 (Global Reduction Labels)**.** Global reduction labels λ are defined to be:

$$\lambda \;::=\; s : \mathtt{p} \to \mathtt{q} : U \;\mid\; s : \mathtt{p} \to \mathtt{q} : l$$

$$\{s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s : G\} \qquad \{s : \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I}\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l_k} \{s : G_k\}$$

$$\frac{\{s : G\} \xrightarrow{\lambda} \{s : G'\} \quad (\star)}{\{s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G\} \xrightarrow{\lambda} \{s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G'\}}$$

$$\frac{\forall i \in I, \{s : G_i\} \xrightarrow{\lambda} \{s : G_i'\} \quad (\star)}{\{s : \mathtt{p} \to \mathtt{q} : \{l_i : G_i\}_{i\in I}\} \xrightarrow{\lambda} \{s : \mathtt{p} \to \mathtt{q} : \{l_i : G_i'\}_{i\in I}\}}$$

$$\frac{E \xrightarrow{\lambda} E'}{E \cdot E_0 \xrightarrow{\lambda} E' \cdot E_0}$$

Figure 6.13: Labelled Reduction Relation for Global Environments

with $\mathtt{out}(\lambda)$ and $\mathtt{inp}(\lambda)$:

1. $\mathtt{out}(s : \mathtt{p} \to \mathtt{q} : U) = \mathtt{out}(s : \mathtt{p} \to \mathtt{q} : l) = \mathtt{p}$.
2. $\mathtt{inp}(s : \mathtt{p} \to \mathtt{q} : U) = \mathtt{inp}(s : \mathtt{p} \to \mathtt{q} : l) = \mathtt{q}$.
3. $\mathtt{p} \in \ell$ if $\mathtt{p} \in \mathtt{out}(\ell) \cup \mathtt{inp}(\ell)$.

Figure 6.13 describes the global environment reduction relation. The first rule is the axiom reduction for the input and output interaction between two parties; the second rule is the axiom reduction for the choice; the third and fourth rules formulate the case that the action λ can be performed under the assumption $(\star)$ where $(\star)$ is a condition on the participants of the two actions according the underlying MSP calculus semantics. We summarise the conditions:

1. Synchronous MSP: $\mathtt{p}, \mathtt{q} \notin \lambda$.

   Synchronous MSP requires no relation between the participants of two actions. Intuitively it allows permutations on actions between participants that are not linearly related.

2. Output Asynchronous MSP: $\mathtt{q} \notin \lambda$.

   For the output asynchronous MSP we require that the receiver $\mathtt{q}$ of the second action is not related to the participants in the first action. This condition subsumes synchronous permutations and the asynchrony on the senders (output asynchrony).

3. Input Asynchronous MSP: $\mathtt{p},\mathtt{q} \notin \mathtt{out}(\lambda)$.

   In the case of global input asynchrony we require that the sender of the first action $\mathtt{out}(\lambda)$ is not related to the participants $\mathtt{p},\mathtt{q}$ of the first action. Again this condition subsumes synchronous permutations and input asynchrony.

4. Input/Output MSP: $\mathtt{q} \notin \lambda \vee \mathtt{p},\mathtt{q} \notin \mathtt{out}(\lambda)$.

   Input/output asynchrony requires either the condition for output asynchrony to hold or the condition for input asynchrony to hold.

The fifth rule of the labelled reduction system is the congruence rule. We often omit the label λ by writing $\longrightarrow$ for $\xrightarrow{\lambda}$ and $\longrightarrow^*$ for $(\xrightarrow{\lambda})^*$.

As a simple example of the above LTS, consider the synchronous MSP process:

$$s : \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle.\mathtt{p}' \to \mathtt{q}' : \{l_1 : \mathtt{end}, l_2 : \mathtt{p}' \to \mathtt{q}' : \langle U_2 \rangle.\mathtt{end}\}$$

Since $\mathtt{p},\mathtt{q},\mathtt{p}',\mathtt{q}'$ are pairwise distinct, we can apply the second and third rules for the synchronous MSP to obtain:

$$s : \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle.\mathtt{p}' \to \mathtt{q}' : \{l_1 : \mathtt{end}, l_2 : \mathtt{p}' \to \mathtt{q}' : \langle U_2 \rangle.\mathtt{end}\} \xrightarrow{s:\mathtt{p}'\to\mathtt{q}':l_1} s : \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle.\mathtt{end}$$

6.4.2 Global Configurations

We introduce the *environment configuration* structure.

Definition 6.4.3 (Environment Configuration and Labelled Transition Semantics)**.** We write

$$(E, \Gamma, \Delta) \text{ if } \exists E' \cdot E \longrightarrow^* E' \text{ and } \Delta \subseteq \mathtt{proj}(E')$$

The global environment E records the knowledge for the session environment Δ and its dual (observer) environment with respect to E. The side condition ensures that E respects the linear environment Δ up-to reduction.

In Figure 6.14, we define a labelled transition system over well-formed environment configurations, that refines the LTS over environments (i.e. $(\Gamma, \Delta) \xrightarrow{\ell} (\Gamma', \Delta')$) both for the synchronous case (§ 6.2.5) and the asynchronous case (§ 6.3.5)

Each rule requires a environment LTS, corresponding to either Figure 6.10 for the synchronous MSP or § 6.3.5 for the asynchronous MSP, in order to control a transition following the global protocols that derive from the global environment E. Rule [Acc] is the rule for accepting a session initialisation so that it creates a new mapping $s : G$ which matches Γ in a governed environment E. Rule [Req] is the rule for requesting a new session and it is dual to [Acc].

The next seven rules are the transition relations on session channels and we assume the condition $\mathtt{proj}(E_1) \supseteq \Delta$ to ensure that the base action of the environment matches with the action in the global environment. Rule [Out] defines the output action, where the type of the value and the action of (Γ, Δ) meets those in E. Rule [In] defines the input action and it is dual to rule [Out]. Rule [ResN] is a scope opening rule for a name so that the environment can perform the corresponding type $\langle G \rangle$ of a. Rule [ResS] is a scope opening rule for a session channel which creates a set of mappings for the opened session channel s' corresponding to the LTS of the environment. Rules [Sel] and [Bra] define the selection and branching, which are similar to [Out] and [In]. In rule [Tau], we annotated the reduction relation on Δ (Definition 6.3.3) with λ labels as follows:

$$[\text{Acc}]\quad \frac{\Gamma \vdash a : \langle G \rangle \quad (\Gamma, \Delta_1) \xrightarrow{a[A](s)} (\Gamma, \Delta_2)}{(E, \Gamma, \Delta_1) \xrightarrow{a[A](s)} (E \cdot s : G, \Gamma, \Delta_2)} \qquad [\text{Req}]\quad \frac{\Gamma \vdash a : \langle G \rangle \quad (\Gamma, \Delta_1) \xrightarrow{\overline{a}[A](s)} (\Gamma, \Delta_2)}{(E, \Gamma, \Delta_1) \xrightarrow{\overline{a}[A](s)} (E \cdot s : G, \Gamma, \Delta_2)}$$

$$[\text{Out}]\quad \frac{\Gamma \vdash v : U \quad (\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (\Gamma, \Delta_2) \quad \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E_2}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (E_2, \Gamma, \Delta_2)}$$

$$[\text{In}]\quad \frac{(\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (\Gamma \cdot v : U, \Delta_2) \quad \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:U} E_2}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v \rangle} (E_2, \Gamma \cdot v : U, \Delta_2)}$$

$$[\text{ResN}]\quad \frac{(\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} (\Gamma \cdot a : \langle G \rangle, \Delta_2) \quad \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:\langle G \rangle} E_2}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(a)} (E_2, \Gamma \cdot a : \langle G \rangle, \Delta_2)}$$

$$[\text{ResS}]\quad \frac{\begin{array}{c}(\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} (\Gamma, \Delta_2 \cdot \{s'[\mathtt{p}_i] : T_i\}_{i \in I}) \cdot \forall i . G \lceil \mathtt{p}_i = T_i \\ \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:T} E_2\end{array}}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])} (E_2 \cdot s' : G, \Gamma, \Delta_2 \cdot \{s'[\mathtt{p}_i] : T_i\}_{i \in I})}$$

$$[\text{Sel}]\quad \frac{(\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} (\Gamma, \Delta_2) \quad \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l} E_2}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\oplus l} (E_2, \Gamma, \Delta_2)}$$

$$[\text{Bra}]\quad \frac{(\Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\& l} (\Gamma, \Delta_2) \quad \Delta \subseteq \mathtt{proj}(E_1) \quad E_1 \xrightarrow{s:\mathtt{q}\to\mathtt{p}:l} E_2}{(E_1, \Gamma, \Delta_1) \xrightarrow{s[\mathtt{p}][\mathtt{q}]\& l} (E_2, \Gamma, \Delta_2)}$$

$$[\text{Tau}]\quad \frac{(\Delta_1 = \Delta_2,\ E_1 = E_2) \vee (\Delta_1 \xrightarrow{\lambda} \Delta_2,\ E_1 \xrightarrow{\lambda} E_2) \quad \Delta \subseteq \mathtt{proj}(E_1)}{(E_1, \Gamma, \Delta_1) \xrightarrow{\tau} (E_2, \Gamma, \Delta_2)}$$

$$[\text{Inv}]\quad \frac{E_1 \longrightarrow^* E_1' \quad (E_1', \Gamma_1, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma_2, \Delta_2)}{(E_1, \Gamma_1, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma_2, \Delta_2)}$$

Figure 6.14: The LTS for Environment Configuations

1. $\{s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle ; T \cdot s[\mathtt{q}] : [\mathtt{p}]?(U) ; T'\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s[\mathtt{p}] : T \cdot s[\mathtt{q}] : T'\}$.

2. $\{s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_i\}_{i \in I} \cdot s[\mathtt{q}] : [\mathtt{p}] \& \{l_j : T_j'\}_{j \in J}\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l_k} \{s[\mathtt{p}] : T_k \cdot s[\mathtt{q}] : T_k'\}\ I \subseteq J, k \in I$.

3. $\Delta \cup \Delta' \xrightarrow{\lambda} \Delta \cup \Delta''$ if $\Delta' \xrightarrow{\lambda} \Delta''$.

In rule [Tau], the τ reduction of the linear environment should match the reduction of the global environment. Rule [Inv] is the induction rule: the global environment E_1 reduces to E_1' to perform the observer's actions, hence the observed process can perform the action w.r.t. E_1'. Hereafter we write $\longrightarrow$ for $\xrightarrow{\tau}$.

Example 6.4.1 (LTS for environment configuration)**.** Let

1. $E = s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.\mathtt{p} \to \mathtt{q} : \langle U \rangle.G$, 2. $\Gamma = v : U$ 3. $\Delta = s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T_{\mathtt{p}}$

with

1. $\mathtt{partic}(G) = \{\mathtt{p}, \mathtt{q}\}$ 2. $G\lceil \mathtt{p} = T_{\mathtt{p}}$, $G\lceil \mathtt{q} = T_{\mathtt{q}}$

Then (E, Γ, Δ) is an environment configuration since:

$$E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G$$

and

$$\mathtt{proj}(s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G) = s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T_{\mathtt{p}} \cdot s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}$$

with

$$\mathtt{proj}(s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G) \supset \Delta$$

Then we can apply the global configuration LTS rule [Out] to both:

$$\begin{array}{rcl} s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G & \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} & s : G \\ (\Gamma, s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T_{\mathtt{p}}) & \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} & (\Gamma, s[\mathtt{p}] : T_{\mathtt{p}}) \end{array}$$

to obtain:

$$(s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G, \Gamma, \Delta) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (s : G, \Gamma, s[\mathtt{p}] : T_{\mathtt{p}})$$

By the last result and the fact that $E \longrightarrow s : \mathtt{p} \to \mathtt{q} : \langle U \rangle.G$, we use rule [Inv] to obtain:

$$(E, \Gamma, \Delta) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v \rangle} (s : G, \Gamma, s[\mathtt{p}] : T_{\mathtt{p}})$$

We clarify the semantics for environment configurations and the labelled transition system for environment configurations in the next two definitions.

We introduce the *governance judgement*, where a global environment respects the session typing environment of a process.

Definition 6.4.4 (Global Configuration)**.** Let $\Gamma \vdash P \triangleright \Delta$ be coherent. We write

$$E, \Gamma \vdash P \triangleright \Delta \text{ if } \exists E' \cdot E \longrightarrow^* E' \text{ and } \Delta \subseteq \mathtt{proj}(E')$$

Following the global configuration definition, the global environment E records the knowledge of both the session environment (Δ) of the observed process P and the session environment of its *observer*. The side conditions ensure that E is coherent with Δ: there exist E' reduced from E whose projection should cover the environment of P (since E should include the observer's information together with the observed process information recorded into Δ).

We define the governed typed transition relation for processes.

Definition 6.4.5 (Global configuration transition)**.** We write $E_1, \Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E_2, \Gamma' \vdash P_2 \triangleright \Delta_2$ if $E_1, \Gamma \vdash P_1 \triangleright \Delta_1$, $P_1 \xrightarrow{\ell} P_2$ and $(E_1, \Gamma, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma', \Delta_2)$.

A global configuration transition for a process P is controlled by the global environment E, in contrast to the typed transition (Figure 3.10 for the ASP, Figure 6.10 for the synchronous MSP and § 6.3.5 for the asynchronous MSP) where the transition is only controlled by the session environment Δ and the shared environment Γ.

The following proposition states that the configuration LTS preserves the well-formedness of the environment configuration.

Proposition 6.4.1 (Invariants)**.**

1. $(E_1, \Gamma_1, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma_2, \Delta_2)$ implies that $(E_2, \Gamma_2, \Delta_2)$ is an environment configuration.

2. If $\Gamma \vdash P \triangleright \Delta$ and $P \longrightarrow P'$ with $\mathtt{co}(\Delta)$, then $E, \Gamma \vdash P \triangleright \Delta \longrightarrow E, \Gamma \vdash P' \triangleright \Delta'$ and $\mathtt{co}(\Delta')$

Proof. The proof for Part 1 can be found in Appendix C.3.3. Part 2 is verified by simple transitions using [Tau] in Figure 6.14. □

6.5 Multiparty Session π-calculus Behavioural Theory

This section presents a typed behavioural theory for the Multiparty Session π-calculi, based on the typed and untyped semantics, developed previously in this chapter. We define two classes of bisimulation relations for each MSP calculi, based on the local typed transition in Definition 6.2.10 and on the global configuration transition in Definition 6.4.5 respectively. We also define the corresponding reduction-closed congruence relation. Note that the reduction-closed congruence for the globally governed semantics is defined based on the global environment semantics developed in § 6.4.

The results for this section are summarised in the inclusion relations between the locally typed bisimulation and the conditions for both the locally typed and globally governed bisimulation to coincide.

6.5.1 Local Multiparty Behavioural Theory

In this section we present the behavioural theory for the MSP calculi. The theory presented here is characterised as *local* multiparty behavioural theory since we use the information from the local type to restrict the behaviour of each process. The definitions in this section apply equally for all of the MSP calculi. The differences between each calculi arise from the different underlying reduction relations (for reduction congruence) and typed labelled transition relations (for bisimulation) for each calculi.

We define the typed relation as the binary relation over typed processes.

Definition 6.5.1 (Typed relation)**.** We define a relation $\mathscr{R}$ as a *typed relation* if it relates two closed, coherent typed terms $\Gamma \vdash P_1 \triangleright \Delta_1 \; \mathscr{R} \; \Gamma \vdash P_2 \triangleright \Delta_2$. We often write $\Gamma \vdash P_1 \triangleright \Delta_1 \; \mathscr{R} \; P_2 \triangleright \Delta_2$.

To define the relation counterparts for each of the MSP calculi we define:

Definition 6.5.2.

- $m \; ::= \; s \mid i \mid o \mid io$.
- We write $\mathscr{R}^s, \mathscr{R}^i, \mathscr{R}^o, \mathscr{R}^{io}$ for a typed relation $\mathscr{R}$ over the terms of the synchronous, the input asynchronous, the output asynchronous and the input/output asynchronous MSP respectively.
- Furthermore we define the partial order $\sqsubseteq$ on the rules: $s \sqsubset i$, $s \sqsubset o$, $i \sqsubset io$, $o \sqsubset io$.

Next we define the notion of the typed *barb* [ACS98]:

Definition 6.5.3 (Barbs)**.** We write

1. $\Gamma \vdash P \triangleright \Delta \downarrow_{s[\mathtt{p}][\mathtt{q}]}$ if $P \equiv (\nu \; \tilde{a}\tilde{s})(s[\mathtt{p}][\mathtt{q}]!\langle v \rangle; R \mid Q)$ with $s \notin \tilde{s}$ and $s[\mathtt{q}] \notin \mathtt{dom}(\Delta)$
2. $\Gamma \vdash P \triangleright \Delta \downarrow_a$ if $P \equiv (\nu \; \tilde{a}\tilde{s})(\overline{a}[n](s).R \mid Q)$ with $a \notin \tilde{a}$ and $a \in \mathtt{dom}(\Gamma)$.

We write n for either a or $s[\mathtt{p}][\mathtt{q}]$, to define $\Gamma \vdash P \triangleright \Delta \Downarrow_n$ if $\Gamma \vdash P \triangleright \Delta \twoheadrightarrow \Gamma \vdash P' \triangleright \Delta'$ and $\Gamma \vdash P' \triangleright \Delta' \downarrow_n$.

The typed barbed is controlled by the typing environment of the process, in the case of barbs on session channels. For a session barb $\downarrow_{s[\mathtt{p}][\mathtt{q}]}$, we require that the opposing role ($s[\mathtt{q}]$) is not present in the linear environment Δ, to ensure the linear usage of session channels (i.e a session endpoint interacts only with the its corresponding endpoint).

The context is defined as:

$$C \quad ::= \quad - \mid C \mid P \mid P \mid C \mid (\nu\, n)C \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C' \mid \mu X.C \mid$$
$$s!\langle v\rangle;C \mid s?(x);C \mid s \oplus l;C \mid s\&\{l_i : C_i\}_{i\in I} \mid \overline{a}(x).C \mid a(x).C$$

where $C[P]$ substitutes process P for each hole $(-)$ in context.

In equivalence relations between typed processes we require that the linear environments converge:

Definition 6.5.4 (Linear Environment Convergence)**.** We write $\Delta_1 \rightleftharpoons \Delta_2$ if there exists Δ such that $\Delta_1 \longrightarrow^* \Delta$ and $\Delta_2 \longrightarrow^* \Delta$.

We now define the contextual congruence based on the typed barb definition and [HY95].

Definition 6.5.5 (Reduction congruence)**.** A typed relation $\mathscr{R}$ is *reduction congruence* if it satisfies the following conditions for each $\Gamma \vdash P_1 \triangleright \Delta_1\ \mathscr{R}\ P_2 \triangleright \Delta_2$ with $\Delta_1 \rightleftharpoons \Delta_2$.

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow_m$ iff $\Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow_m$
2. Whenever $\Gamma \vdash P_1 \triangleright \Delta_1\ \mathscr{R}\ P_2 \triangleright \Delta_2$ holds, then
   - $P_1 \twoheadrightarrow P_1'$ implies $P_2 \twoheadrightarrow P_2'$ such that $\Gamma \vdash P_1' \triangleright \Delta_1'\ \mathscr{R}\ P_2' \triangleright \Delta_2'$ holds with $\Delta_1' \rightleftharpoons \Delta_2'$.
   - The symmetric case.
3. For all closed context C, such that $\Gamma \vdash C[P_1] \triangleright \Delta_1'$ and $\Gamma \vdash C[P_2] \triangleright \Delta_2'$ where $\Delta_1' \rightleftharpoons \Delta_2'$, $\Gamma \vdash C[P_1] \triangleright \Delta_1'\ \mathscr{R}\ \Gamma \vdash C[P_2] \triangleright \Delta_2'$.

The reduction congruence relation is denoted as $\cong$.

Definition 6.5.6 (Multiparty session bisimulation)**.** A typed relation $\mathscr{R}$ over closed processes is a (weak) *multiparty session bisimulation* or often a *bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1\ \mathscr{R}\ P_2 \triangleright \Delta_2$ holds, then:

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\hat{\ell}} \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $\Gamma' \vdash P_1' \triangleright \Delta_1' \mathrel{\mathscr{R}} P_2' \triangleright \Delta_2'$.

2. The symmetric case.

The maximum bisimulation exists which we call *bisimilarity*, denoted by $\approx$. We sometimes leave environments implicit, writing e.g. $P \approx Q$. We also write $\approx$ for untyped bisimilarity which is defined using only the untyped LTS in Figure 6.9 (together with its extension in § 6.3.5 for the asynchronous cases).

We use the following lemma, to derive Theorem 6.5.1. See Appendix C.3.2.

Lemma 6.5.1. $\Gamma \vdash P_1 \triangleright \Delta_1 \approx P_2 \triangleright \Delta_2$ *then* $\Delta_1 \rightleftharpoons \Delta_2$.

Proof. The proof uses the co-induction method and can be found in Appendix C.3.2. □

Theorem 6.5.1 (Soundness and completeness)**.** $\cong = \approx$.

Proof. The proof is a simplification of the proof of Theorem 6.5.3 in Appendix C.3.5. □

We explain our theory with an example over synchronous MSP processes:

Example 6.5.1 (Synchronous Multiparty Bisimulation)**.** Let:

$$\begin{aligned} P_1 &= \Gamma \vdash a[1](x).b[1](y).x[1][3]!\langle v\rangle;y[2]!\langle w\rangle;\mathbf{0} \triangleright \emptyset \\ P_2 &= \Gamma \vdash a[2](x).\overline{b}[2](y).(y[1]?(z);\mathbf{0} \mid x[2][3]!\langle v\rangle;\mathbf{0}) \triangleright \emptyset \\ P_3 &= \Gamma \vdash \overline{a}[3](x).x[3][1]?(z);x[3][2]?(y);\mathbf{0} \triangleright \emptyset \end{aligned}$$

First we explain the LTS for session initialisation from Figures 6.9 and 6.10. By $\langle \text{Acc} \rangle$ and $\langle \text{Req} \rangle$, we get:

$$\begin{aligned} \Gamma \vdash P_1 \triangleright \emptyset \xrightarrow{a[\{1\}](s_1)} P_1' &= \Gamma \vdash b[1](y).s_1[1][3]!\langle v\rangle;y[2]!\langle w\rangle;\mathbf{0} \triangleright s_1[1] : [3]!\langle U\rangle;\mathtt{end} \\ \Gamma \vdash P_2 \triangleright \emptyset \xrightarrow{a[\{2\}](s_1)} P_2' &= \Gamma \vdash \overline{b}[2](y).(y[1]?(z);\mathbf{0} \mid s_1[2][3]!\langle v\rangle;\mathbf{0}) \triangleright s_1[2] : [3]!\langle U\rangle;\mathtt{end} \\ \Gamma \vdash P_3 \triangleright \emptyset \xrightarrow{\overline{a}[\{3\}](s_1)} P_3' &= \Gamma \vdash s_1[3][1]?(z);s_1[3][2]?(y);\mathbf{0} \triangleright s_1[3] : [1]?(U);[1]?(U);\mathtt{end} \end{aligned}$$

If we apply rule ⟨AccPar⟩, on $P_1 \mid P_2$ we can have:

$$\Gamma \vdash P_1 \mid P_2 \triangleright \emptyset \xrightarrow{a[\{1,2\}](s_1)} \Gamma \vdash P_1' \mid P_2' \triangleright s_1[1] : [3]!\langle U\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle U\rangle;\mathtt{end}$$

Another possible initialisation on $P_1 \mid P_3$ would be:

$$\Gamma \vdash P_1 \mid P_3 \triangleright \emptyset \xrightarrow{\overline{a}[\{1,3\}](s_1)} \Gamma \vdash P_1' \mid P_3' \triangleright s_1[1] : [3]!\langle U\rangle;\mathtt{end} \cdot s_1[3] : [1]?(U);[1]?(U);\mathtt{end}$$

If in the above process we compose in parallel the third process P_2, the set $\{1,2,3\}$ becomes complete so that we can use the rule ⟨TauS⟩ to observe:

$$\Gamma \vdash P_1 \mid P_2 \mid P_3 \triangleright \emptyset \xrightarrow{\tau} \Gamma \vdash (\nu\, s_1)(P_1' \mid P_2' \mid P_3') \triangleright \emptyset$$

Further we can have:

$$\Gamma \vdash P_1' \mid P_2' \triangleright \Delta \xrightarrow{\tau} Q_1 = \Gamma \vdash (\nu\, s_2)(s_1[1][3]!\langle v\rangle;s_2[1][2]!\langle w\rangle;\mathbf{0} \mid s_2[2][1]?(z);\mathbf{0} \mid s_1[2][3]!\langle v\rangle;\mathbf{0}) \triangleright \Delta$$

with $\Delta = \{s_1[1] : [3]!\langle U\rangle;\mathtt{end} \cdot s_1[2] : [3]!\langle U\rangle;\mathtt{end}\}$.
We can now observe that

$$\Gamma \vdash Q_1 \mid P_3' \triangleright \Delta \cdot s_1[1] : [3]!\langle U\rangle;\mathtt{end} \cdot s_1[3] : [1]?(U);[1]?(U);\mathtt{end} \approx^s \mathbf{0} \triangleright \emptyset$$

since $(\Gamma,\Delta) \not\xrightarrow{\ell}$ for any $\ell \neq \tau$. However by the untyped synchronous bisimulation (we consider only the untyped LTS in Figure 6.9), we have that:

$$Q_1 \mid P_3' \not\approx \mathbf{0}$$

since, e.g. $Q_1 \mid P_3' \xrightarrow{s_1[1][3]!\langle v\rangle}$.

It is very convenient to define a common syntax for the MSP calculi, in order to compare

the behaviour of processes between the different calculi. The common syntax is easy to be achieved with the addition of empty endpoint configurations to respect the input/output syntax. Formally:

Definition 6.5.7 (Common MSP Syntax)**.**

- Let all MSP calculi share the syntax and the structural congruence definition for the input/output asynchronous MSP.
- For each calculi, we replace the operational rule [Link] with the operational rule [Link] for the input/output asynchronous MSP.
- We ensure that MSP processes are localised with empty queues: The synchronous MSP typing system is extended to use runtime typing and the localisation definition. The runtime typing system for the synchronous and the output asynchronous MSP is extended to use the rule (QEmptyI). Similarly the message typing system for the synchronous and the input asynchronous MSP is extended to use rule (QEmptyO). Furthermore, for all calculi we use the message typing rule (SRes) for the input/output asynchronous MSP.

The above definition ensures a common syntax for the MSP calculi up-to empty endpoint configurations. All other semantics remain same. We use the annotation m on relations, (e.g $\approx^s$) to specify the set of operational semantics we are using on the common syntax.

The MSP calculi are related with each other, based on their bisimulation definition. More specifically the bisimilarity relations form inclusions based on the partial order defined by $\sqsubset$:

Theorem 6.5.2 (Behavioural Inclusion)**.**

- $\approx_g^{m_1} \subsetneq \approx_g^{m_2}$ if $m_1 \sqsubset m_2$.
- $\approx_g^i$ and $\approx_g^o$ are incompatible.

Proof. To show the inclusion direction of the first part of the theorem we take advantage of the fact that the bisimilarity relation is closed under session transition (see Lemma C.3.4 in the Appendix). The incompatibility directions for Part 1 and Part 2 are shown by providing the proper counter-examples. See Appendix C.3.6 for details. □

The importance of the above result comes from the fact that we can relate and classify the different asynchronous calculi based on the way they handle asynchronous communication. The synchronous MSP behaviour is included in all the asynchronous cases, while the incompatible behaviours of the input and output asynchronous MSP are included in the input/output asynchronous MSP.

6.5.2 Globally Governed Multiparty Behavioural Theory

We introduce the bisimulation theory based on the globally governed semantics presented in § 6.4. The bisimulation theory presented in this section derives from the labelled transition system defined in Definition 6.4.5. The globally governed LTS offers fine-grained control over the behaviour of processes with respect to global environments.

To define the reduction-closed congruence, we first refine the barb, which is controlled by the global environment E:

Definition 6.5.8 (Global Barbs)**.** We write:

1. $E, \Gamma \vdash P \triangleright \Delta \downarrow_{s[\mathtt{p}][\mathtt{q}]}$ if
   - $P \downarrow_{s[\mathtt{p}][\mathtt{q}]}$.
   - $\exists E' \cdot E \longrightarrow^* E'$ and $\Delta \subseteq \mathtt{proj}(E')$.
   - $E' \xrightarrow{\lambda}$ where $\lambda \in \{s : \mathtt{p} \to \mathtt{q} : U, s : \mathtt{p} \to \mathtt{q} : l\}$.
   - $s[\mathtt{p}] \in \Delta, s[\mathtt{q}] \notin \Delta$.

2. $E, \Gamma \vdash P \triangleright \Delta \downarrow_a$ if $a \in \mathtt{dom}(\Gamma)$

We write $E, \Gamma \vdash P \triangleright \Delta \Downarrow_m$ if $E, \Gamma \vdash P \triangleright \Delta \twoheadrightarrow E, \Gamma \vdash P' \triangleright \Delta'$ and $E, \Gamma \vdash P' \triangleright \Delta' \downarrow_m$

Before we proceed with the definition of relations over global configurations (see Definition 6.4.4), we define the *global environment concatenation* operator. The concatenation operator returns, if it exists, the minimal global environment that describes a pair of global environments.

Definition 6.5.9 (Global Environment Concatenation)**.**

1. We write $T_1 \sqsubseteq T_2$ if the syntax tree of T_2 includes T_1.
2. We extend to $G_1 \sqsubset G_2$ if $\forall s[\mathtt{p}] : T_1 \in \mathtt{proj}(s : G_1)$ then $s[\mathtt{p}] : T_2 \in \mathtt{proj}(s : G_2)$ and $T_1 \sqsubseteq T_2$.
3. Then we define: $E_1 \sqcup E_2 = \{E_i(s) \mid E_j(s) \sqsubseteq E_i(s)\} \cup E_1 \setminus \mathtt{dom}(E_2) \cup E_2 \setminus \mathtt{dom}(E_1)$.

Example 6.5.2 (Global Environment Concatenation)**.**

- We write

$$[\mathtt{q}]?(U'); T \sqsubseteq [\mathtt{p}]!\langle U \rangle; [\mathtt{q}]?(U'); T$$

since $[\mathtt{q}]?(U'); T$ is included in the syntax tree of $[\mathtt{p}]!\langle U \rangle; [\mathtt{q}]?(U'); T$.

- Let:

$$\begin{aligned} E_1 &= s_1 : \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle.\mathtt{p}' \to \mathtt{q}' : \langle U_2 \rangle.\mathtt{p} \to \mathtt{q} : \langle U_3 \rangle.\mathtt{end} \cdot s_2 : \mathtt{p} \to \mathtt{q} : \langle W_2 \rangle.\mathtt{end} \\ E_2 &= s_1 : \mathtt{p} \to \mathtt{q} : \langle U_3 \rangle.\mathtt{end} \cdot s_2 : \mathtt{p}' \to \mathtt{q}' : \langle W_1 \rangle.\mathtt{p} \to \mathtt{q} : \langle W_2 \rangle.\mathtt{end} \end{aligned}$$

Then

$$\begin{aligned} E_1 \sqcup E_2 = \; & \mathtt{p} \to \mathtt{q} : \langle U_1 \rangle.\mathtt{p}' \to \mathtt{q}' : \langle U_2 \rangle.\mathtt{p} \to \mathtt{q} : \langle U_3 \rangle.\mathtt{end} \\ & \cdot s_2 : \mathtt{p}' \to \mathtt{q}' : \langle W_1 \rangle.\mathtt{p} \to \mathtt{q} : \langle W_2 \rangle.\mathtt{end} \end{aligned}$$

We define the relation over global configurations.

Definition 6.5.10 (Configuration relation)**.** The relation $\mathscr{R}$ is a *configuration relation* between two configurations $E_1, \Gamma \vdash P_1 \triangleright \Delta_1$ and $E_2, \Gamma \vdash P_2 \triangleright \Delta_2$, written

$$E_1 \sqcup E_2, \Gamma \vdash P \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$$

if $E_1 \sqcup E_2$ is defined.

Two global configurations can be related under a configuration relation if their global environment typing is defined up to syntax tree inclusion. The configuration relation allows us to prove:

Proposition 6.5.1 (Decidability)**.**

1. Given E_1 and E_2, a problem whether $E_1 \sqcup E_2$ is defined or not is decidable and if it is defined, the calculation of $E_1 \sqcup E_2$ terminates
2. Given E, a set $\{E' \mid E \longrightarrow^* E'\}$ is finite.

Proof. (1) Since $T_1 \sqsubseteq T_2$ is a syntactic tree inclusion, it is reducible to a problem to check the isomorphism between two types. This problem is decidable [YV07]. *(2)* The global LTS has one-to-one correspondence with the LTS of global automata in [DY12] whose reachability set is finite. □

We define the governed reduction congruence relation:

Definition 6.5.11 (Governed reduction congruence)**.** A configuration relation $\mathscr{R}$ is a *governed reduction congruence* if whenever $E, \Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ then

1. $E, \Gamma \vdash P_1 \triangleright \Delta_1 \Downarrow_n$ if and only if $E, \Gamma \vdash P_2 \triangleright \Delta_2 \Downarrow_n$

2. $P_1 \twoheadrightarrow P_1'$ if and only if $P_2 \twoheadrightarrow P_2'$ and $E, \Gamma \vdash P_1' \triangleright \Delta_1' \mathrel{\mathscr{R}} P_2' \triangleright \Delta_2'$

3. For all closed contexts C, such that $E, \Gamma \vdash C[P_1] \triangleright \Delta_1'$ and $E, \Gamma \vdash C[P_2] \triangleright \Delta_2'$ then $E, \Gamma \vdash C[P_1] \triangleright \Delta_1' \mathrel{\mathscr{R}} C[P_2] \triangleright \Delta_2'$.

The union of all governed reduction congruence relations is denoted as $\cong_g$.

We define the globally governed bisimulation relation:

Definition 6.5.12 (Globally governed bisimulation)**.** A configuration relation $\mathscr{R}$ is a *globally governed weak bisimulation* (or governed bisimulation) if whenever $E, \Gamma \vdash P_1 \triangleright \Delta_1 \mathrel{\mathscr{R}} P_2 \triangleright \Delta_2$ holds, then:

1. $E, \Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E_1', \Gamma' \vdash P_1' \triangleright \Delta_1'$ implies $E, \Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\hat{\ell}} E_2', \Gamma' \vdash P_2' \triangleright \Delta_2'$ such that $E_1' \sqcup E_2', \Gamma' \vdash P_1' \triangleright \Delta_1' \mathrel{\mathscr{R}} P_2' \triangleright \Delta_2'$.

2. The symmetric case.

The maximum bisimulation exists which we call *governed bisimilarity*, denoted by $\approx_g$. We sometimes leave environments implicit, writing e.g. $P \approx_g Q$.

Lemma 6.5.2 (Weakening)**.**

1. If $E, \Gamma \vdash P \triangleright \Delta$ then

   - $E \cdot s : G, \Gamma \vdash P \triangleright \Delta$.
   - $E = E' \cdot s : G$ and $\exists G' \cdot \{s : G'\} \twoheadrightarrow \{s : G\}$ then $E' \cdot s : G', \Gamma \vdash P \triangleright \Delta$.

2. If $(E, \Gamma, \Delta) \xrightarrow{\ell} (E, \Gamma', \Delta')$ then

   - $(E \cdot s : G, \Gamma, \Delta) \xrightarrow{\ell} (E \cdot s : G, \Gamma', \Delta')$
   - If $E = E' \cdot s : G$ and $\{s : G'\} \twoheadrightarrow \{s : G\}$ then $(E' \cdot s : G', \Gamma, \Delta) \xrightarrow{\ell} (E' \cdot s : G', \Gamma', \Delta')$

3. If $E, \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$

   - $E \cdot s : G, \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$
   - If $E = E' \cdot s : G$ and $\{s : G'\} \twoheadrightarrow \{s : G\}$ then $E' \cdot s : G', \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$

Proof. We only show Part 1. Other parts are similar.

- From the governance judgement definition we have that $E \longrightarrow^* E_1$ and $\mathtt{proj}(E_1) \supseteq *(\Delta)$.

  Let $E \cdot s : G \longrightarrow E_1 \cdot s : G$. Then $\mathtt{proj}(E_1 \cdot s : G) = \mathtt{proj}(E_1) \cup \mathtt{proj}(s : G) \supseteq \mathtt{proj}(E_1) \supseteq *(\Delta)$.

- From the governance judgement definition we have that $E \cdot s : G \longrightarrow^* E_1 \cdot s : G_1$ and $\mathtt{proj}(E_1 \cdot s : G_1) \supseteq *(\Delta)$.

  Let $E \cdot s : G' \longrightarrow^* E_1 \cdot s : G' \longrightarrow^* E_1 \cdot S : G_1$. Then the result is immediate.

□

Lemma 6.5.3 (Strengthening)**.**

1. If $E \cdot s : G, \Gamma \vdash P \triangleright \Delta$ then

   - If $s \notin \mathtt{fn}(P)$ then $E, \Gamma \vdash P \triangleright \Delta$
   - If $\exists G', E \cdot s : G \twoheadrightarrow E_2 \cdot s : G' \twoheadrightarrow E_1 \cdot s : G_1$ with $\mathtt{proj}(E_1 \cdot s : G_1) \supseteq \Delta$ then $E \cdot s : G', \Gamma \vdash P \triangleright \Delta$

2. If $(E \cdot s : G, \Gamma, \Delta) \xrightarrow{\ell} (E' \cdot s : G, \Gamma', \Delta')$ then

   - $(E, \Gamma, \Delta) \xrightarrow{\ell} (E', \Gamma', \Delta')$
   - If $\exists G', E \cdot s : G \twoheadrightarrow E_2 \cdot s : G' \twoheadrightarrow E_1 \cdot s : G_1$ with $\mathtt{proj}(E_1 \cdot s : G_1) \supseteq \Delta$ then $(E \cdot s : G', \Gamma, \Delta) \xrightarrow{\ell} (E' \cdot s : G', \Gamma', \Delta')$

3. If $E \cdot s : G, \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$ then

   - If $s \notin \mathtt{fn}(P)$ then $E, \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$
   - If $\exists G', E \cdot s : G \twoheadrightarrow E_2 \cdot s : G' \twoheadrightarrow E_1 \cdot s : G_1$ with $\mathtt{proj}(E_1 \cdot s : G_1) \supseteq \Delta$ then $E \cdot s : G', \Gamma \vdash P_1 \triangleright \Delta_2 \approx_g P_2 \triangleright \Delta_2$

Proof. We prove part 1. Other parts are similar.

- From the governance judgement definition we have that $E \cdot s : G \longrightarrow^* E_1 \cdot s : G_1$ and $\mathtt{proj}(E_1 \cdot s : G_1) = \mathtt{proj}(E_1) \cup \mathtt{proj}(s : G_1) \supseteq *(\Delta)$. Since $s \notin \mathtt{fn}(P)$ then $s \notin \mathtt{dom}(\Delta)$, then $\mathtt{proj}(s : G_1) \cap *(\Delta) = \emptyset$. So $\mathtt{proj}(E_1) \supseteq *(\Delta)$ and $E \longrightarrow^* E_1$.
- The result is immediate from the definition of governance judgement.

□

Lemma 6.5.4. $\approx_g$ is congruence.

Proof. The proof is by a case analysis on the context structure. The interesting case is the parallel composition, which uses Proposition 6.4.1. See Appendix C.3.4. □

Lemma 6.5.5. $\cong_g \subseteq \approx_g$

Proof. The proof follows the facts that bisimulation has a stratifying definition (the proof method uses the technique from [ACS98]) and that the external actions can always be tested (the technique from [Hen07]). The proof can be found in Appendix C.3.5. □

By Lemmas A.3.1 and 6.5.5, we have:

Theorem 6.5.3 (Soundness and completeness)**.** $\approx_g = \cong_g$.

Proof. The proof was done in Lemmas A.3.1 and 6.5.5. □

We study the relationship between $\approx$ and $\approx_g$.

Theorem 6.5.4. If for all E, $E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx \Gamma \vdash P_2 \triangleright \Delta_2$.

Also if $\Gamma \vdash P_1 \triangleright \Delta_1 \approx \Gamma \vdash P_2 \triangleright \Delta_2$, then for all E, $E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$.

Proof. See Appendix C.3.7. □

To clarify the difference between $\approx$ and $\approx_g$, we introduce the notion of a *simple multiparty process* defined in [HYC08]. A simple process contains only a single session so that it satisfies the progress property as proved in [HYC08]. Formally a process P is *simple* when it is typable with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to at most a single session (i.e. for any $\Gamma \vdash P \triangleright \Delta$ which appears in a derivation, Δ contains at most one session channel in its domain; see [HYC08]). Thus each prefixed sub-term in a simple process has a unique session. Since there is no interleaving of sessions in simple processes, the difference between $\approx^s$ and $\approx^s_g$ disappears.

Theorem 6.5.5 (Coincidence)**.** Assume P_1 and P_2 are simple. If $\exists E \cdot E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx P_2 \triangleright \Delta_2$.

Proof. The proof follows the fact that if P is simple and $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} P' \triangleright \Delta'$ then $\exists E \cdot E, \Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} P' \triangleright \Delta'$ to continue that if P_1, P_2 are simple and $\exists E \cdot E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$ then $\forall E, E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$. The result then comes by applying Lemma 6.5.4.

Details of the proof are in Appendix C.3.8. □

Example 6.5.3 (Governed bisimulation)**.** Recall Example 6.5.1, with $\Gamma \vdash Q_1 \triangleright \Delta$ being the process corresponding to Example 6.5.1. Let process:

$$R_2 = \Gamma \vdash a[2](x).\overline{b}[2](y).(y[1]?(z);x[2][3]!\langle v\rangle;\mathbf{0}) \triangleright \emptyset$$

with

$$P_1 \mid R_2 \xrightarrow{a[1,2](s_1)} \xrightarrow{\tau}$$

$$Q_2 \quad = \quad \Gamma \vdash (\nu\, s_2)(s_1[1][3]!\langle v\rangle; s_2[1][2]!\langle w\rangle; \mathbf{0} \mid s_2[2][1]?(x); s_1[2][3]!\langle v\rangle; \mathbf{0}) \triangleright \Delta$$

Recall that $\Delta = \{s_1[1] : [3]!\langle U\rangle; \mathtt{end} \cdot s_1[2] : [3]!\langle U\rangle; \mathtt{end}\}$. Note that R_2 has a sequential composition of actions instead of the parallel composition for actions in P_2. Assume the two global witnesses:

$$\begin{aligned} E_1 &= s_1 : 1 \to 3 : \langle S\rangle.2 \to 3 : \langle S\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle S\rangle.\mathtt{end} \\ E_2 &= s_1 : 2 \to 3 : \langle S\rangle.1 \to 3 : \langle S\rangle.\mathtt{end} \cdot s_2 : 1 \to 2 : \langle S\rangle.\mathtt{end} \end{aligned}$$

Then the projection of E_1 and E_2 are given as:

$$\begin{aligned} \mathtt{proj}(E_1) &= s_1[1] : [3]!\langle S\rangle; \mathtt{end} \cdot s_1[2] : [3]!\langle S\rangle; \mathtt{end} \cdot s_1[3] : [1]?(S); [2]?(S); \mathtt{end} \\ &\quad\; s_2[1] : [2]!\langle S\rangle; \mathtt{end} \cdot s_2[2] : [1]?(S); \mathtt{end}\cdot \\ \mathtt{proj}(E_2) &= s_1[1] : [3]!\langle S\rangle; \mathtt{end} \cdot s_1[2] : [3]!\langle S\rangle; \mathtt{end} \cdot s_1[3] : [2]?(S); [1]?(S); \mathtt{end}\cdot \\ &\quad\; s_2[1] : [2]!\langle S\rangle; \mathtt{end} \cdot s_2[2] : [1]?(S); \mathtt{end} \end{aligned}$$

with $\Delta \subset \mathtt{proj}(E_1)$ and $\Delta \subset \mathtt{proj}(E_2)$. The reader should note that the difference between E_1 and E_2 is the type of participant 3 at s_1.

By the definition of the global environment configuration, we can write:

$$E_i, \Gamma \vdash Q_1 \triangleright \Delta$$
$$E_i, \Gamma \vdash Q_2 \triangleright \Delta$$

for $i = 1, 2$. Both processes are well-formed global configurations under both witnesses. Now we can observe

$$\Gamma \vdash Q_1 \triangleright \Delta \xrightarrow{s[2][3]!\langle v\rangle} \Gamma \vdash Q_1' \triangleright \Delta'$$

but

$$\Gamma \vdash Q_2 \triangleright \Delta \overset{s[2][3]!\langle v\rangle}{\not\longrightarrow}$$

Hence

$$\Gamma \vdash Q_1 \triangleright \Delta \not\approx^s Q_2 \triangleright \Delta$$

Using the same argument, we have:

$$E_2, \Gamma \vdash Q_1 \triangleright \Delta \not\approx^s_g Q_2 \triangleright \Delta$$

On the other hand, since E_1 *forces* the action $s[2][3]!\langle v\rangle$ to wait:

$$E_1, \Gamma \vdash Q_1 \triangleright \Delta \overset{s[2][3]!\langle v\rangle}{\not\longrightarrow}$$

Hence Q_1 and Q_2 are bisimilar under E_1, i.e.

$$E_1, \Gamma \vdash Q_1 \triangleright \Delta \approx^s_g Q_2 \triangleright \Delta$$

We conclude that the optimisation is correct.

6.6 A Service Oriented Usecase

The bisimulation techniques developed in this paper present interests in both the theoretical and the applied aspects. We have developed semantics for typed environments and use them to define labelled transition semantics for typed processes. We show that session type bisimulations can be defined, either by taking only the local session information of each process into account ($\approx$) or by taking the global session protocols into account ($\approx^s_g$).

The session type restriction on behavioural semantics and the sound and complete bisimulation relations that derive, can be used as a tool to optimise and verify of distributed systems,

and to prove the correctness of service communication.

In this section, we present a usecase based on the real world usecase *UC.R2.13 "Acquire Data From Instrument"* from the Ocean Observatories Initiative (OOI) [OOI], where we intent to show the optimisation and verification of network services.

In this usecase we assume a user program (U) which is connected to the Integrated Observatory Network (ION). The ION provides the interface between users and remote sensing instruments. The user requests, via the ION agent services (A), the acquisition of processed data from an instrument (I). More specifically the user requests from the ION two different formats of the instrument data. In the above usecase we distinguish two points of communication coordination: i) an internal ION multiparty communication and ii) an external communication between ION instruments and agents and the user. In other words it is natural to require the initiation of two multiparty session types to coordinate the services and clients involved in the usecase.

The behaviour of the multiparty session connection between the User (U) and ION is dependent on the implementation and the synchronisation of the internal ION session.

Next we present three possible implementation scenarios and compare their behaviour with respect to the user program. Depending on the ION requirements we can chose the best implementation with the correct behaviour. See Figure 6.15 for a graphic representation of the three scenarios.

6.6.1 Usecase Scenario 1

In the first scenario the user program (U) wants to acquire the first format of data from the instrument (I) and at the same time acquire the second format of the data from an agent service (A). The communication between the agent (A) and the instrument happens internally in the ION on a separate private session.

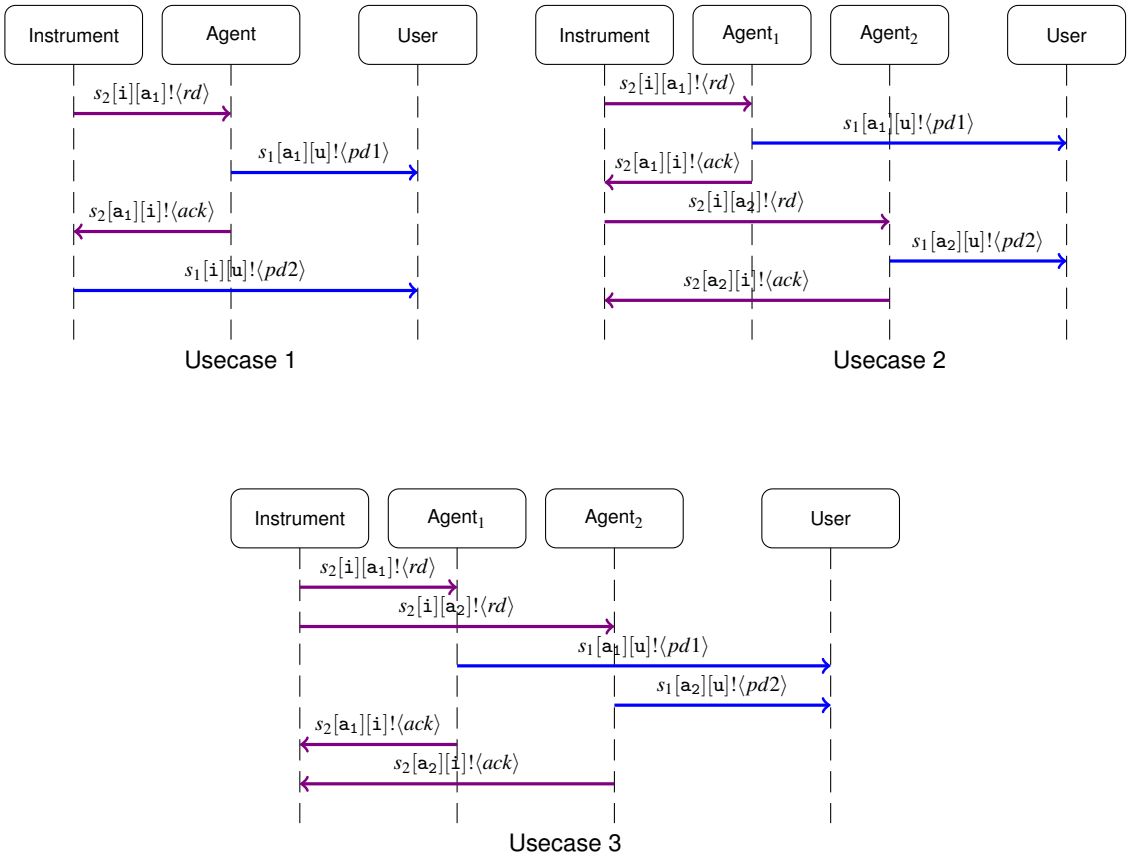

Figure 6.15: Three usecases from UC.R2.13 “Acquire Data From Instrument” in [OOI]

- A new session connection s_1 is established between (U), (I) and (A).
- A new session connection s_2 is established between (A) and (I).
- (I) sends raw data through s_2 to (A).
- (A) sends processed data (format 1) through s_1 to (U).
- (A) sends acknowledgement through s_2 to (I).
- (I) sends processed data (format 2) through s_1 to (U).

The above scenario is implemented as follows:

$$I_0 \mid A \mid U$$

where

$$
\begin{aligned}
I_0 &= a[\mathtt{i0}](s_1).\overline{b}[\mathtt{i0}](s_2).s_2[\mathtt{i0}][\mathtt{a_1}]!\langle \mathtt{rd}\rangle; s_2[\mathtt{i0}][\mathtt{a_1}]?(x); s_1[\mathtt{i0}][\mathtt{u}]!\langle \mathtt{pd}\rangle; \mathbf{0} \\
A &= a[\mathtt{a_1}](s_1).b[\mathtt{a_1}](s_2).s_2[\mathtt{a_1}][\mathtt{i0}]?(x); s_1[\mathtt{a_1}][\mathtt{u}]!\langle \mathtt{pd}\rangle; s_2[\mathtt{a_1}][\mathtt{i0}]!\langle \mathtt{ack}\rangle; \mathbf{0} \\
U &= \overline{a}[\mathtt{u}](s_1).s_1[\mathtt{u}][\mathtt{a_1}]?(x); s_1[\mathtt{u}][\mathtt{i0}]?(y); \mathbf{0}
\end{aligned}
$$

and $\mathtt{i}$ is the instrument role, $\mathtt{a_1}$ is the agent role and $\mathtt{u}$ is the user role.

6.6.2 Usecase scenario 2

Use case scenario 1 implementation requires from the instrument program to process raw data in a particular format (format 2) before sending them to the user program. In a more modular and fine-grain implementation, the instrument program should only send raw data to the ION interface for processing and forwarding to the user. A separate session between the instrument and the ION interface and a separate session between the ION interface and the user make a distinction into different logical and processing levels.

To capture the above implementation we assume a scenario with the user program ($\mathtt{U}$), the instrument ($\mathtt{I}$) and agents ($\mathtt{A_1}$) and ($\mathtt{A_2}$):

- A new session connection s_1 is established between ($\mathtt{U}$), ($\mathtt{A_1}$) and ($\mathtt{A_2}$).
- A new session connection s_2 is established between ($\mathtt{A_1}$, $\mathtt{A_2}$) and ($\mathtt{I}$).
- ($\mathtt{I}$) sends raw data through s_2 to ($\mathtt{A_1}$).
- ($\mathtt{A_1}$) sends processed data (format 1) through s_1 to ($\mathtt{U}$).
- ($\mathtt{A_1}$) sends acknowledgement through s_2 to ($\mathtt{I}$).
- ($\mathtt{I}$) sends raw data through s_2 to ($\mathtt{A_2}$).

- ($\mathtt{A}_2$) sends processed data (format 2) through s_1 to ($\mathtt{U}$).
- ($\mathtt{A}_2$) sends acknowledgement through s_2 to ($\mathtt{I}$).

The above scenario is implemented as follows:

$$I_1 \mid A_1 \mid A_2 \mid U$$

where

$$\begin{aligned}
I_1 &= \overline{b}[\mathtt{i}](s_2).s_2[\mathtt{i}][\mathtt{a}_1]!\langle \mathtt{rd}\rangle; s_2[\mathtt{i}][\mathtt{a}_1]?(x); s_2[\mathtt{i}][\mathtt{a}_2]!\langle \mathtt{rd}\rangle; s_2[\mathtt{i}][\mathtt{a}_1]?(x); \mathbf{0} \\
A_1 &= a[\mathtt{a}_1](s_1).b[\mathtt{a}_1](s_2).s_2[\mathtt{a}_1][\mathtt{i}]?(x); s_1[\mathtt{a}_1][\mathtt{u}]!\langle \mathtt{pd}\rangle; s_2[\mathtt{a}_1][\mathtt{i}]!\langle \mathtt{ack}\rangle; \mathbf{0} \\
A_2 &= a[\mathtt{a}_2](s_1).b[\mathtt{a}_2](s_2).s_2[\mathtt{a}_2][\mathtt{i}]?(x); s_1[\mathtt{a}_2][\mathtt{u}]!\langle \mathtt{pd}\rangle; s_2[\mathtt{a}_2][\mathtt{i}]!\langle \mathtt{ack}\rangle; \mathbf{0} \\
U &= \overline{a}[\mathtt{u}](s_1).s_1[\mathtt{u}][\mathtt{a}_1]?(x); s_1[\mathtt{u}][\mathtt{a}_2]?(y); \mathbf{0}
\end{aligned}$$

and $\mathtt{i}$ is the instrument role, $\mathtt{a}_1$ and $\mathtt{a}_2$ are the agent roles and $\mathtt{u}$ is the user role. Furthermore for session s_1 we have that role $\mathtt{i0}$ (from scenario 1) $= \mathtt{a}_2$, since we want to maintain the session s_1 as it is defined in the scenario 1.

6.6.3 Usecase scenario 3

A step further is to enhance the performance of usecase scenario 2 if the instrument ($\mathtt{I}$) code in usecase scenario 2 can have a different implementation, where raw data are sent to both agents ($\mathtt{A}_1$, $\mathtt{A}_2$) before any acknowledgement is received. ION agents can process data in parallel resulting in an optimised implementation.

- A new session connection s_1 is established between ($\mathtt{U}$), ($\mathtt{A}_1$) and ($\mathtt{A}_2$).
- A new session connection s_2 is established between ($\mathtt{A}_1$, $\mathtt{A}_2$) and ($\mathtt{I}$).

- ($\mathtt{I}$) sends raw data through s_2 to ($\mathtt{A}_1$).
- ($\mathtt{I}$) sends raw data through s_2 to ($\mathtt{A}_2$).
- ($\mathtt{A}_1$) sends processed data (format 1) through s_1 to ($\mathtt{U}$).
- ($\mathtt{A}_1$) sends acknowledgement through s_2 to ($\mathtt{I}$).
- ($\mathtt{A}_2$) sends processed data (format 2) through s_1 to ($\mathtt{U}$).
- ($\mathtt{A}_2$) sends acknowledgement through s_2 to ($\mathtt{I}$).
- A new session connection s_1 is established between ($\mathtt{U}$), ($\mathtt{A}_1$) and ($\mathtt{A}_2$).

The process is now refined as

$$I_2 \mid A_1 \mid A_2 \mid U$$

where

$$I_2 \quad = \quad \overline{b}[\mathtt{i}](s_2).s_2[\mathtt{i}][\mathtt{a}_1]!\langle \mathtt{rd} \rangle; s_2[\mathtt{i}][\mathtt{a}_2]!\langle \mathtt{rd} \rangle; s_2[\mathtt{i}][\mathtt{a}_1]?(x); s_2[\mathtt{i}][\mathtt{a}_1]?(x); \mathbf{0}$$

and $\mathtt{i}$ implements the instrument role, $\mathtt{a}_1$ and $\mathtt{a}_2$ are the agent roles and $\mathtt{u}$ is the user role.

6.6.4 Behavioural Equivalence

The main concern of the three scenarios is to implement the Integrated Ocean Network interface respecting the multiparty communication protocols.

Having the user process as the observer we can see that typed processes for Usecase scenario 1

and Usecase scenario 2:

$$\Gamma \vdash I_0 \mid A \triangleright \Delta_0$$

$$\Gamma \vdash I_1 \mid A_1 \mid A_2 \triangleright \Delta_1$$

are bisimilar (using $\approx$). We give the bisimulation closure that characterises the two processes. Recall that $\mathtt{i0} = \mathtt{a}_2$. Let:

$$\begin{array}{lllll}
\Gamma \vdash I_0 \mid A \triangleright \Delta_0 & \xrightarrow{a[s](\mathtt{a}_1,\mathtt{a}_2)} & \Gamma \vdash P_1 \triangleright \Delta_{01} & \xrightarrow{\tau} & \Gamma \vdash P_2 \triangleright \Delta_{02} \\
 & \xrightarrow{\tau} & \Gamma \vdash P_3 \triangleright \Delta_{03} & \xrightarrow{s_1[\mathtt{a}_1][\mathtt{u}]!\langle \mathtt{pd} \rangle} & \Gamma \vdash P_4 \triangleright \Delta_{04} \\
 & \xrightarrow{\tau} & \Gamma \vdash P_5 \triangleright \Delta_{05} & \xrightarrow{s_1[\mathtt{i0}][\mathtt{u}]!\langle \mathtt{pd} \rangle} & \Gamma \vdash P_6 \triangleright \Delta_{06}
\end{array}$$

$$\begin{array}{lllll}
\Gamma \vdash I_1 \mid A_1 \mid A_2 \triangleright \Delta_1 & \xrightarrow{a[s](\mathtt{a}_1,\mathtt{a}_2)} & \Gamma \vdash Q_1 \triangleright \Delta_{11} & \xrightarrow{\tau} & \Gamma \vdash Q_2 \triangleright \Delta_{12} \\
 & \xrightarrow{\tau} & \Gamma \vdash Q_3 \triangleright \Delta_{13} & \xrightarrow{s_1[\mathtt{a}_1][\mathtt{u}]!\langle \mathtt{pd} \rangle} & \Gamma \vdash Q_4 \triangleright \Delta_{14} \\
 & \xrightarrow{\tau} & \Gamma \vdash Q_5 \triangleright \Delta_{15} & \xrightarrow{\tau} & \Gamma \vdash Q_6 \triangleright \Delta_{16} \\
 & \xrightarrow{s_1[\mathtt{a}_2][\mathtt{u}]!\langle \mathtt{pd} \rangle} & \Gamma \vdash P_7 \triangleright \Delta_{17} & \xrightarrow{\tau} & \Gamma \vdash Q_8 \triangleright \Delta_{18}
\end{array}$$

The bisimulation closure is:

$$\begin{array}{rcl}
\mathscr{R} & = & \{(\Gamma \vdash I_0 \mid A \triangleright \Delta_0, \Gamma \vdash I_1 \mid A_1 \mid A_2 \triangleright \Delta_1), (\Gamma \vdash P_1 \triangleright \Delta_{01}, \Gamma \vdash Q_1 \triangleright \Delta_{11}) \\
 & & (\Gamma \vdash P_2 \triangleright \Delta_{02}, \Gamma \vdash Q_2 \triangleright \Delta_{12}), (\Gamma \vdash P_3 \triangleright \Delta_{03}, \Gamma \vdash Q_3 \triangleright \Delta_{13}) \\
 & & (\Gamma \vdash P_4 \triangleright \Delta_{04}, \Gamma \vdash Q_4 \triangleright \Delta_{14}), (\Gamma \vdash P_5 \triangleright \Delta_{05}, \Gamma \vdash Q_5 \triangleright \Delta_{15}) \\
 & & (\Gamma \vdash P_5 \triangleright \Delta_{05}, \Gamma \vdash Q_6 \triangleright \Delta_{16}), (\Gamma \vdash P_6 \triangleright \Delta_{06}, \Gamma \vdash Q_7 \triangleright \Delta_{17}) \\
 & & (\Gamma \vdash P_6 \triangleright \Delta_{06}, \Gamma \vdash Q_8 \triangleright \Delta_{18})\}
\end{array}$$

The two implementations (scenario 1 and scenario 2) are completely interchangeable with respect to $\approx$.

If we proceed with the case of the scenario 3 we can see that typed process $\Gamma \vdash I_2 \mid A_1 \mid A_2 \triangleright \Delta_2$

cannot be simulated (using $\approx$) by scenarios 1 and 2, since we can observe the execution:

$$\Gamma \vdash I_1 \mid A_1 \mid A_2 \triangleright \Delta_1 \xrightarrow{\tau} \xrightarrow{a[s](\mathtt{a}_1,\mathtt{a}_2)} \xrightarrow{s_1[\mathtt{a}_2][\mathtt{u}]!\langle \mathtt{pd} \rangle}$$

By changing the communication ordering in the ION private session s_2 we changed the communication behaviour on the external session channel s_1. Nevertheless, the communication behaviour remains the same if we take into account the global multiparty protocol of s_1 and the way it governs the behaviour of the three usecase scenarios.

Hence we use $\approx_g^s$. The definition of the global environment is as follows:

$$E \quad = \quad s_1 : \mathtt{a}_1 \to \mathtt{u} : \langle \mathtt{PD} \rangle . \mathtt{a}_2 \to \mathtt{u} : \langle \mathtt{PD} \rangle .$$

The global protocol governs processes $I_1 \mid A_1 \mid A_2$ (similarly $I_0 \mid A$) and $I_2 \mid A_1 \mid A_2$ to always observe action $\xrightarrow{s_1[\mathtt{a}_2][\mathtt{u}]!\langle \mathtt{pd} \rangle}$ after action $\xrightarrow{s_1[\mathtt{a}_1][\mathtt{u}]!\langle \mathtt{pd} \rangle}$ for both processes.

Also note that the global protocol for s_2 is not present in the global environment, because s_2 is restricted. The specification and implementation of session s_2 are abstracted from the behaviour of session s_1.

Part III

Chapter 7

Conclusion

This is the concluding chapter of this dissertation, in which we include an extended comparison of the different aspects of this work with related works. The dissertation ends with a concluding remark.

7.1 Related Work

Confluence: The confluence theory for session types is based on the confluence theory for the π-calculus initially proposed in [PW97], which presents a theory of constructing confluent and determinate processes in the general case of π-calculus transitions. We use the ideas and definitions from that work to prove that session channels construct confluent systems and to reason about concurrent systems (see the Lauer-Needham transform in § 5.5).

Although we use the main intuitions and definitions from [PW97] to construct confluent processes, we follow a session oriented approach to the subject of confluence, rather than a general approach to confluence for typed π-calculi. This work investigates the confluent behaviour of a typed restricted labelled transition system in the presence of asynchronous input/output queues. We reason about systems based on the confluence property of partial

transitions (i.e. the confluence property on session transitions). For example, when reasoning about the Lauer-Needham transform in § 5.5 we need to show that non-session transitions are also confluent. We use assumptions in the construction of event-driven confluent processes, because event-driven transitions are not confluent in the presence of the `arrive`-predicate (see example in § 5.1).

Expressiveness: There is a number of works on expressiveness that are directly or indirectly related with the work in this thesis.

The work [BPV08] examines the encodability of various messaging media in the asynchronous π-calculus [HT91a]. Specifically, it shows that a message bag (no ordering) medium is encodable in the asynchronous π-calculus, while stack media (LIFO policy) and message queues (FIFO policy) are impossible to be encoded. The impossibility in encoding message queues implies the impossibility of encoding the asynchronous calculi developed in this dissertation in terms of the asynchronous π-calculus. Furthermore it does not study the effects of typed transitions and event-driven programming on encodings.

In [DH11] the linear types for the asynchronous π-calculus are studied in the presence of subtyping to provide a fully abstract encoding of the synchronous binary session types, that is proved based on the may and must barbed equivalences. The linear typed π-calculus is based on [HT91a] and uses no message queues for communication. Furthermore it is interested in the encodings between different calculi, in contrast to this work that encodes a program in the same calculus. It would also be interesting to see the extensions of linear types to encode multiparty and asynchronous session types and as well the use of the typed behavioural techniques developed in this dissertation as the behavioural basis for proving full abstraction.

The relations between a session type system and linear logic [Gir87] are studied in [CP10, Wad12]. Both papers present a strict subset of session typed calculi that are in correspondence with different forms of sequent calculi, with the intention to be typed under a session type system with direct correspondence to linear logic. The attempt is to prove a Curry-Howard like

correspondence between the π-calculus, session types and linear logic, with the reductions of the π-calculus corresponding to cut-eliminated proof steps in linear logic. The results of these two papers have a deep impact to our understanding of sessions.

A subsequent paper [PCPT12] of [CP10] extends the above intuitions. To be more specific, the work in [CP10] proposes the session typed πDILL calculus, with DILL standing for dual intuitionistic linear logic and shows the correspondence between πDILL and the DILL calculus. The work in [PCPT12] develops a theory for the study of termination and liveness properties and proposes an observational theory based on the typed context bisimilarity. Termination and liveness are important in the context of session types and are the main intuition for the development of multiparty session types. In this work we propose a uniform behavioural theory for session types, which in correlation with the above works gives further intuitions about the relation of asynchronous and multiparty session types with linear logic and creates new perspectives for understanding the behavioural theory of types in terms of linear logic.

The expressiveness and encodability results for programming constructs that can test the presence of actions or events have been studied in the context of the Linda language [BGZ00] and CSP [Low09, Low10].

The work in [BGZ00] compares the expressive powers between three variants of asynchronous Linda-like calculi, with a construct for inspecting the presence of messages in the tuple space (i.e. the message medium that defines asynchronous communication), which is reminiscent of the *inp* predicate of Linda. The first calculus (called *instantaneous*) corresponds to the asynchronous π-calculus [HT91a], the second calculus (called *ordered*) formalises emissions of messages to the tuple spaces, and the third one (called *unordered*) models unordered outputs in the tuple space by decomposing one messaging into two stages — emission from an output process and *rendering* from the tuple space.

The semantics for message inspecting in the three Linda-like calculi assume the observation of labels on inspection transitions, in contrast to this work which treats `arrive`-inspection

as internal expression evaluation. Based on the labelled transition semantics developed the authors define a behavioural theory based on barbed bisimulation.

A study of the expressiveness of the semantics shows that the instantaneous and ordered calculi are Turing powerful, while the unordered calculus is not.

The work in [Low09] uses the term *availability* to describe whether a channel is ready to input a message. The work extends CSP with a construct that checks if a parallel process is available to perform an output action on a given channel. The construct is similar to the `arrive`-predicate in the sense that it uses the `if ready` a `then` P `else` Q to test for the availability of channel a. It studies operational and denotational semantics, demonstrating the interest to investigate event primitives using process calculi. A subsequent work [Low10] studies the expressiveness of the calculus defined in [Low09] and the pure CSP (i.e. the calculus in [Low09] without the `if ready` a `then` P `else` Q predicate) that is extended to describe availability tests through its semantics. The latter work focusses on trace equivalence and proves the full abstraction of the two approaches.

A contrast between the session typed π-calculi developed in this dissertation and the calculi in [BGZ00, Low09, Low10] is difficult to make, due to the differences in the base calculi (Linda-like calculus and CSP in contrast to the π-calculus). Nevertheless, we should mention the lack of FIFO queues as communication mediums and the absence of a typing system in the related work. Linda-like calculi defines asynchrony based on the notion of tuple-space, while CSP does not address the issue of asynchrony. The behavioural theory in [Low09, Low10] is based on denotational semantics and trace equivalence. Furthermore, there is no study of a large application to demonstrate the applicative aspects of the constructs under consideration.

Session typed formalisms: A number of session type-based systems that guarantee advanced progress properties in the context of Web services have been proposed recently [C$^+$09, CV10, CP09, B$^+$08]. In [C$^+$09] the authors study a foundational approach on session types,

that is concerned with the development of the algorithms for a sound and safe session type system. Caires and Viera propose in [CV10] conversation types as a type discipline for a service-oriented π-calculus called the conversation calculus [VCS08]. Another service oriented approach is found in [CP09] where services are implemented in the π-calculus and are typed under the contract type system. In [B+08] the authors develop a multiparty session type system for implementing multiparty web applications.

The techniques for type-safe and dynamic event inspection that were developed in this thesis cannot be found in the above bibliography. These techniques can be used for the construction of type-safe services with a reactive control flow that results in optimised service implementations. In the presence of multiparty session types this thesis develops a set of modular and extensible calculi for that can be used for the development of multiparty network applications. In fact Part 3 of this work extends in an elegant way the work in [B+08]. Furthermore, it develops the behavioural theory for reasoning in each different calculi and studies general properties of the event-driven framework with the study of the Lauer-Needham transform. In contrast none of the above work studies neither the behavioural theory (bisimulation) nor the applications in eventful programming.

Dynamic types. An important contribution of this dissertation is the ability to statically type programs with dynamic control flow. We attack this problem by reducing the notion of dynamic control flow to a type-driven control flow and we use constructs from the literature of the λ-calculus to type such a program.

Static analysis for a dynamic flow of control for the λ-calculus was studied in [ACPP89, ACPR95], where (i) the `typecase` construct is applied for general expressions e; (ii) the type of e can be matched against type patterns with free variables; and (iii) the default case is selected if there is no matching (motivated by the use of untyped input/output). In this work we use the `typecase` construct to match the runtime type of a session with sessions closed session types (in contrast to open type patterns for λ-expressions). We impose a stronger

constraint on the typecase construct with the use of session set types, dispensing with the default case. Finally we use subtyping to keep session duality in a consistent state.

Implementation. The event-driven ideas developed in this work were implemented as an extension of Java with session types [HKP+10], based on Session Java [HYH08]. The work considers both programming abstraction and performance aspects of the ESP in practice and provides programmers a typed session event selector API (a session-typed extension of the standard `java.nio.channels.Selector` API) for registering and selecting session endpoints (instances of a session-typed extension of `java.net.Socket`). The type checker of the existing Session Java [HYH08] compiler was extended to handle the above constructs together with session set types, with the adoption of the presented type system to Java expressions and statement control flow to ensure communication and event-handling safety for event-driven Session Java programs. The Eventful Session Java Runtime is designed to uniformly incorporate a variety of transports, including TCP, HTTP and shared memory, under the Session Java session abstraction; this means a single Session Java selector instance is capable of monitoring sessions running over heterogeneous transports as well as being of heterogeneous types.

Eventful Session Java was used to implement an event-driven SMTP server and a client as a real-world application use case [HKP+10]. The server is interoperable with standard, non-Session Java (i.e. not session-typed) SMTP clients such as Outlook, Thunderbird and Apple Mail (and likewise for the Session Java client). While the Session Java implementations are, of course, checked to be session type-safe by the Session Java compiler, the Session Java Runtime also performs run-time monitoring of the (SMTP) session to ensure that non-session-typed peers indeed conform to the same protocol.

Performance and scalability benchmarks for the Eventful SJ Runtime [HKP+10, SJ10] demonstrate the feasibility of integrating session types and event-driven programming, and affirm the application of the Lauer-Needham transform in practice. The benchmarks include basic multi-threaded and event-driven implementations in standard Java as base cases. Micro-benchmarks

and a macro-benchmark using the SMTP server show that thread-eliminated Eventful SJ programs exhibit higher average throughput and better response-time than the multi-threaded versions as the server is loaded by an increasing number of concurrent clients.

7.2 Conclusion

This dissertation presents a bisimulation theory for session typed calculi in both the binary and multiparty session types.

We initially investigate the possibility of developing an asynchronous version for session types based on the asynchronous π-calculus [HT91b, Bou92]. The asynchronous π-calculus allows for unordered delivery of messages, which is counter-intuitive with session types, since session types are based on the sequentiality of send/receive actions. To overcome this problem without compromising the order-preserving property of session types, we propose the semantic definition of intermediate FIFO session queues as processes with fine-grained communication semantics, that are used to store sent messages until their final reception. In the case of shared name interactions we use the asynchronous π-calculus approach together with an intermediate queue for the unordered delivery of session initiation messages. The resulting calculus is called the Asynchronous Session π-calculus (ASP) and it is used as the core calculus for studying bisimulation and equivalence theory in the context of session types.

The importance of the ASP is shown by its capabilities to model network communication. Networks use a series of intermediate buffers to store a message until its final delivery to the receiving application. The unordered delivery of session initiation messages correspond to asynchronous connection initiation in network protocols, such as the TCP. The buffered and ordered delivery inside a session corresponds to reliable network communication.

The session type system for pure ASP terms (i.e. terms with no session queues) is based on the classic systems [HVK98, YV07] for session types. We type session queues with the use

of a session type system for messages, where the FIFO ordering of each queue ensures the sequentiality of session messages.

The basic insight for a bisimulation theory for session types, comes from the fact that a session environment can control the observables of a process, so that its behaviour will conform to session type properties. We develop a typed bisimulation theory for the ASP, based on the labelled transition system for untyped processes and a labelled transition system for typing environments. We prove that the derived (weak) bisimilarity on closed session typed terms, is the maximum reduction-closed congruence that preserves observation, making two bisimilar processes indistinguishable under any observer.

Concerning the bisimulation theory we take a further step to study the confluence and determinacy properties of session transitions. Confluence is a property that is inherited by the communication structure of systems and it is used to reason about the correctness of large applications. Due to the session linearity that is enforced by the typing system, session transitions are confluent and determinate.

We follow asynchrony in the context of event-driven programming. Event-driven programming is one of the major frameworks that utilise asynchronous programming. Events are characterised by detectability, in the sense that event-driven computation can detect the presence of an event as a message in the communication medium. Event types may also drive the flow of the computation. The latter fact introduces a dynamic and reactive control flow in processes, where static analysis is obfuscated and non-trivial without the aid of type-oriented programming constructs.

We capture the event-driven framework in an extension of the ASP called the Eventful Session π-calculus. We use the event-driven paradigm to enhance our theory in numerous ways. From a theoretical point of view we are interested in the typing of processes with a reactive control flow. From an applied aspect, we believe that our eventful theory gives the basic primitives to implement the different event-driven programming models in a typed setting.

We chose to extend ASP with two key process terms: i) the `arrive`-predicate, which is used as an expression to check for the existence of messages in message buffers; and ii) the `typecase` process that type-checks the runtime type of session channels and proceeds with the computation accordingly. The `arrive`-predicate is typed as a Boolean expression. For the `typecase` process we propose the session set type, which is the set of the possible session types that are implemented by a `typecase` process. The session set type is transparent up-to subtyping with respect to the ASP session type syntax.

The bisimulation properties studied for the ASP continue to hold for the ESP, since ESP is a rather straightforward and transparent up-to subtyping extension of ASP. Specifically we prove that (weak) bisimilarity is the maximum reduction-closed congruence that preserves observation. The confluence and determinacy properties continue to hold for session transitions and the `typecase` transition. Transitions on the `arrive`-predicate though, are not confluent as we show with a simple counterexample.

We demonstrate the applicability of the event-driven theory with the encoding of basic event-driven constructs and routines. A basic event-driven routine is the event-loop, which is a single threaded flow of control that reacts to events (i.e. message arrivals) and proceeds with an event processing routine. After its completion the event-handling routine returns the control to the event-loop for the selection of the next ready event. The event-loop can be build on top of the selector programming construct. The selector registers in its structure a list of channels and iterates through them to check whether they have data for processing (message arrival). We define a set of session semantics for the selector construct to extend ESP to ESP^{+}. We next show that the selector semantics are encodable in the terms of ESP and prove the type and behavioural invariance of the encodings. We then construct a general event-loop using the encoded selector. We prove that the order of the registered channels on the structure of the selector is invariant with respect to the behaviour of the corresponding event-loop, provided that the event-handling routines of the event-loop are confluent and determinate.

The observation made by Lauer and Needham in [LN79] argues about the duality of the

two main approaches for concurrent programming: i) thread-based programming; and the ii) event-driven paradigm. The authors define a set of thread programming primitives and a set of event-driven programming primitives and make their argument by providing encodings for expressing one set in terms of the other. However there is no known result that studies the equivalence and meta-theoretic properties of such encodings.

In this dissertation we assume a threaded ESP server that spawns a parallel thread to service each client from an unbounded number of clients. We then define a transformation from the threaded server to a single thread event-loop server. We use the behavioural invariance result for the selector to prove that our transformation is type and semantic preserving, under the hypothesis that the threaded server handles clients in a non-recursive, sequential and confluent way. The assumption of a sequential and non-recursive handling of a client by each server thread is made to achieve an easier and less detailed proof of the main theorem (Theorem 5.5.1). Nevertheless the structure of the proof gives a strong intuition that the main theorem (Theorem 5.5.1) holds in the general case where the threaded server handles each client in a confluent way.

In the last part of the thesis we develop a behavioural theory for multiparty session types [HYC08, B$^+$08], based on the theory developed for binary session types. Multiparty session types were developed to overcome the limitations presented by binary session types. Binary session types are not powerful enough to enforce the sequentiality and linearity properties in a set of more than two communication participants. A global multiparty protocol is a structure that describes communication for all participants. The local projection of a global protocol results in a set of local types for all participants, that enforces session types properties in a concurrent computation.

We initially develop the theory for a synchronous multiparty session calculus, that serves as a core calculus to define family of asynchronous multiparty session calculi, each of them following a different approach on asynchronous buffered communication. Specifically we use intermediate FIFO buffers to define an asynchronous MSP calculus with output localised

endpoints, i.e. we store output messages in a local endpoint until their delivery. Similarly we define an asynchronous calculus with input localised endpoints, where messages are accumulated in a localised input buffer before reception. A third asynchronous calculus uses the approach developed for the ASP calculus where we use both input and output intermediate endpoints to achieve fine-grained communication. The session type systems for all cases are shown to be sound via the corresponding subject reduction theorems.

We follow the principles developed in the ASP case, to define a (weak) bisimulation relation for each one of the MSP calculi. We use the untyped labelled transition system together with a labelled transition system on local session types to define a local typed transition relation on session typed processes. Based on the typed transition relation we define a (weak) bisimulation relation for every MSP calculus. We show that all bisimulation relations coincide to the corresponding reduction closed and barb preserving congruences.

The intuition that a type environment can control a process transition has led to the development of semantics for global multiparty protocols. We use the above semantics to define a (weak) bisimulation relation which is controlled by the global multiparty protocol instead of the local session type. We call such a bisimulation globally governed bisimulation, which is coarser than the bisimulation defined using the local session type. We prove that the globally governed bisimulation, for each calculus, is a reduction closed and barb preserving congruence.

The final results relate the locally defined session bisimulation with the globally governed bisimulation. The elegant extension of the synchronous MSP to the asynchronous MSP calculi allow for a uniform framework to define all four calculi. Based on the uniform framework we show the inclusion relations between the four locally defined session bisimulations.

Throughout the thesis we have different definitions for session type calculi both for the binary and multiparty session types. In the case of binary session types we work with an asynchronous definition in contrast with the multiparty case where we define a synchronous

calculus as a reference calculus to define a family of asynchronous multiparty session semantics. A first question on this observation would be about the possibility of defining a family of calculi for the binary case similar to the family of calculi for the multiparty case. The answer to this question comes from the multiparty theory itself (i.e. the theory developed in Chapter 6) since it gives strong evidence about a possible structure and behaviour for binary session types. A second intuition about the behaviour of different models of binary session types comes from the work done in Section 5.2 where we compare the behaviour of the ASP with other known π-calculi.

An important result on the behaviour of asynchronous semantics is the fact that process behaviour differs between different asynchronous models (see Chapter 6, Theorem 6.5.2 and Section 5.2). A question that arises here is whether we can define a different set of asynchronous session type calculi and classify their behavioural relations. A suggestion towards this direction is to define a calculus with an intermediate buffer between the input and output endpoints. A generalisation of this suggestion would be to define such an intermediate buffer as an agent and use it to define a calculi that allows a finite and variable number of intermediate message mediums. The latter approach can be used to resemble and study the behaviour of actual networks and actual network communication.

The results of this dissertation intend to have an impact on the behaviour analysis of typed processes. Besides the purely mathematical interest on the bisimulation frameworks for session, we are interested in the application of the behavioural theory on real systems, since session types is a typing system that deals with the desired communication properties of such a system. It is an intention that the theory developed in this thesis will be used as a reference for developing behavioural frameworks and for specifying and verifying correct distributed applications.

Bibliography

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, *Dynamic typing in a statically-typed language*, Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '89, ACM, 1989, pp. 213–227.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy, *Dynamic typing in polymorphic languages*, J. Funct. Program. **5** (1995), no. 1, 111–130.

[ACS98] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi, *On bisimulations for the asynchronous pi-calculus*, TCS **195** (1998), no. 2, 291–324.

[Agh86] Gul Agha, *Actors: a model of concurrent computation in distributed systems*, MIT Press, Cambridge, MA, USA, 1986.

[AHT+02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur, *Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming*, In Proceedings of the 2002 Usenix ATC, 2002.

[B+08] Lorenzo Bettini et al., *Global progress in dynamically interleaved multiparty sessions*, CONCUR, LNCS, vol. 5201, Springer, 2008, pp. 418–433.

[BDM98] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul, *Better operating system features for faster network servers*, SIGMETRICS Performance Evaluation Review **26** (1998), no. 3, 23–30.

[BGZ00] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro, *Comparing three semantics for linda-like languages*, Theor. Comput. Sci. **240** (2000), no. 1, 49–90.

[BH13] G. Bernardi and M. Hennessy, *Using higher-order contracts to model session types*, ArXiv e-prints (2013).

[Bou92] Gerard Boudol, *Asynchrony and the pi-calculus*, Tech. Report 1702, INRIA, 1992.

[BPV08] Romain Beauxis, Catuscia Palamidessi, and Frank D. Valencia, *On the asynchronous nature of the asynchronous pi-calculus*, Concurrency, Graphs and Models, LNCS, vol. 5065, Springer, 2008, pp. 473–492.

[C+09] Giuseppe Castagna et al., *Foundations of session types*, PPDP'09, ACM, 2009, pp. 219–230.

[CDCY07] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida, *Asynchronous Session Types and Progress for Object-Oriented Languages*, FMOODS'07, LNCS, vol. 4468, 2007, pp. 1–31.

[CK05] Ryan Cunningham and Eddie Kohler, *Making events less slippery with eel*, HOTOS'05, 2005, pp. 3–3.

[Cli81] William D Clinger, *Foundations of actor semantics*, Tech. report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1981.

[CP09] G. Castagna and L. Padovani, *Contracts for mobile processes*, CONCUR 2009, LNCS, no. 5710, 2009, pp. 211–228.

[CP10] Luís Caires and Frank Pfenning, *Session types as intuitionistic linear propositions*, Proceedings of the 21st international conference on Concurrency theory (Berlin, Heidelberg), CONCUR'10, Springer-Verlag, 2010, pp. 222–236.

[CV10] Luís Caires and Hugo Torres Vieira, *Conversation types*, Theor. Comput. Sci. **411** (2010), no. 51-52, 4399–4440.

[DCMYD06] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou, *Session Types for Object-Oriented Languages*, ECOOP'06, LNCS, vol. 4067, 2006, pp. 328–352.

[DGS12] Ornela Dardha, Elena Giachino, and Davide Sangiorgi, *Session types revisited*, Proceedings of the 14th symposium on Principles and practice of declarative programming (New York, NY, USA), PPDP '12, ACM, 2012, pp. 139–150.

[DH11] Romain Demangeon and Kohei Honda, *Full abstraction in a subtyped pi-calculus with linear types*, CONCUR, 2011, pp. 280–296.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida, *Dynamic multirole session types*, POPL, 2011, pp. 435–446.

[DY12] ———, *Multiparty session types meet communicating automata*, ESOP, 2012, pp. 194–213.

[EJ09] Patrick Eugster and K. R. Jayaram, *Eventjava: An extension of Java for event correlation*, ECOOP, LNCS, vol. 5653, Springer, 2009, pp. 570–594.

[GH05] Simon Gay and Malcolm Hole, *Subtyping for Session Types in the Pi-Calculus*, Acta Informatica **42** (2005), no. 2/3, 191–225.

[Gir87] Jean-Yves Girard, *Linear logic*, TCS **50** (1987), 1–102.

[GV10] Simon Gay and Vasco T. Vasconcelos, *Linear type theory for asynchronous session types*, J. Funct. Program. **20** (2010), no. 1, 19–50.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger, *A universal modular actor formalism for artificial intelligence*, IJCAI, 1973, pp. 235–245.

[Hen07] Matthew Hennessy, *A Distributed Pi-calculus*, CUP, 2007.

[HKP+10] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda, *Type-safe eventful sessions in Java*, ECOOP'10, LNCS, vol. 6183, Springer-Verlag, 2010, pp. 329–353.

[HO08] Philipp Haller and Martin Odersky, *Scala actors: Unifying thread-based and event-based programming*, Theoretical Computer Science (2008).

[Hon93] Kohei Honda, *Types for Dyadic Interaction*, CONCUR'93 (Eike Best, ed.), LNCS, vol. 715, Springer-Verlag, 1993, pp. 509–523.

[HR04] Matthew Hennessy and Julian Rathke, *Typed behavioural equivalences for processes in the presence of subtyping*, Mathematical. Structures in Comp. Sci. **14** (2004), no. 5, 651–684.

[HT91a] Kohei Honda and Mario Tokoro, *An object calculus for asynchronous communication*, ECOOP'91, LNCS, vol. 512, 1991, pp. 133–147.

[HT91b] ———, *On asynchronous communication semantics*, Object-Based Concurrent Computing, 1991, pp. 21–51.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo, *Language primitives and type disciplines for structured communication-based programming*, ESOP'98, LNCS, vol. 1381, Springer, 1998, pp. 22–138.

[HY95] Kohei Honda and Nobuko Yoshida, *On reduction-based process semantics*, TCS **151** (1995), no. 2, 437–486.

[HY07] ———, *A uniform type structure for secure information flow*, ACM Trans. Program. Lang. Syst. **29** (2007), no. 6.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone, *Multiparty Asynchronous Session Types*, POPL'08, ACM, 2008, pp. 273–284.

[HYH08] Raymond Hu, Nobuko Yoshida, and Kohei Honda, *Session-Based Distributed Programming in Java*, ECOOP'08, LNCS, vol. 5142, Springer, 2008, pp. 516–541.

[KH11] Vasileios Koutavas and Matthew Hennessy, *A testing theory for a higher-order cryptographic language*, Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software (Berlin, Heidelberg), ESOP'11/ETAPS'11, Springer-Verlag, 2011, pp. 358–377.

[KKK07] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek, *Events can make sense*, ATC'07, USENIX Association, 2007, pp. 1–14.

[Kou09] Dimitrios Kouzapas, *A session type discipline for event driven programming models*, Master's thesis, Imperial College London, 2009, http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/d.kouzapas.pdf.

[KPT99] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner, *Linearity and the Pi-Calculus*, ACM TOPLAS **21** (1999), no. 5, 914–947.

[Kro04] Maxwell Krohn, *Building secure high-performance web services with OKWS*, ATEC'04, USENIX Association, 2004, pp. 15–15.

[Lea03] Doug Lea, Scalable IO in Java, http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf, November 2003.

[LN79] Hugh C. Lauer and Roger M. Needham, *On the duality of operating system structures*, SIGOPS Oper. Syst. Rev. **13** (1979), no. 2, 3–19.

[Low09] Gavin Lowe, *Extending CSP with tests for availability*, CPA, 2009, pp. 325–347.

[Low10] ———, *Models for CSP with availability information*, EXPRESS'10, 2010, pp. 91–105.

[LY99] Tim Lindholm and Frank Yellin, *Java virtual machine specification*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[LZ07] Peng Li and Steve Zdancewic, *Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives*, SIGPLAN Not. **42** (2007), no. 6, 189–199.

[Mil80] Robin Milner, *A calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92, Springer, Berlin, 1980.

[Mil89] Robin Milner, *Communication and concurrency*, Prentice Hall, 1989.

[Mil92] ———, *The polyadic π-calculus: A tutorial*, Proceedings of the International Summer School on Logic Algebra of Specification, Marktoberdorf, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker, *A calculus of mobile processes, I*, Inf. Comput. **100** (1992), no. 1, 1–40.

[MY07] Dimitris Mostrous and Nobuko Yoshida, *Two session typing systems for higher-order mobile processes*, TLCA'07, LNCS, vol. 4583, Springer, 2007, pp. 321–335.

[MY09] Dimitris Mostrous and Nobuko Yoshida, *Session-based communication optimisation for higher-order mobile processes*, TLCA'09, LNCS, vol. 5608, Springer, 2009, pp. 203–218.

[NIO] *New i/o apis*, http://java.sun.com/j2se/1.4.2/docs/guide/nio/.

[OAC+06] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger, *An Overview*

of the Scala Programming Language (2. edition), Tech. report, EPFL Lausanne, Switzerland, 2006.

[OOI] *Ocean Observatories Initiative (OOI)*, http://www.oceanobservatories.org/.

[Ous96] John Ousterhout, *Why threads are a bad idea (for most purposes)*, http://www.cs.utah.edu/~regehr/research/ouster.pdf, http://home.pacbell.net/ouster/threads.ppt, January 1996.

[Par81] David Park, *Concurrency and automata on infinite sequences*, Proceedings of the 5th GI-Conference on Theoretical Computer Science (London, UK, UK), Springer-Verlag, 1981, pp. 167–183.

[PCPT12] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho, *Linear logical relations for session-based concurrency*, ESOP, 2012, pp. 539–558.

[Pie02] Benjamin C. Pierce, *Types and programming languages*, MIT Press, 2002.

[PS96] B. Pierce and D. Sangiorgi, *Typing and subtyping for mobile processes*, MSCS **6** (1996), no. 5, 409–454.

[PW97] Anna Philippou and David Walker, *On confluence in the pi-calculus*, ICALP'97, Lecture Notes in Computer Science, vol. 1256, Springer, 1997, pp. 314–324.

[San92] Davide Sangiorgi, *Expressing mobility in process algebras: First-order and higher order paradigms*, Ph.D. thesis, University of Edinburgh, 1992.

[San09] Davide Sangiorgi, *On the origins of bisimulation and coinduction*, ACM Trans. Program. Lang. Syst. **31** (2009), no. 4, 15:1–15:41.

[SJ10] SJ, *SJ homepage*, http://www.doc.ic.ac.uk/~rhu/sessionj.html, 2010.

[SKS11] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii, *Environmental bisimulations for higher-order languages*, ACM Trans. Program. Lang. Syst. **33** (2011), no. 1, 5:1–5:69.

[SMI11] Sun Microsystems Inc., *New IO APIs*, http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html, 2011.

[SW01] Davide Sangiorgi and David Walker, *Pi-calculus: A theory of mobile processes*, Cambridge University Press, New York, NY, USA, 2001.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo, *An Interaction-based Language and its Typing System*, PARLE'94, LNCS, vol. 817, 1994, pp. 398–413.

[vB+03] Rob von Behren et al., *Capriccio: scalable threads for internet services*, SOSP '03, ACM, 2003, pp. 268–281.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer, *Why events are a bad idea (for high-concurrency servers)*, Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9 (Berkeley, CA, USA), HOTOS'03, USENIX Association, 2003, pp. 4–4.

[VCS08] Hugo T. Vieira, Lus Caires, and Joo C. Seco, *The conversation calculus: a model of service oriented computation*, In Proc. of ESOP08, LNCS, Springer, 2008.

[VWW96] Robert Virding, Claes Wikström, and Mike Williams, *Concurrent programming in ERLANG (2nd ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[Wad12] Philip Wadler, *Propositions as sessions*, ICFP, 2012, pp. 273–286.

[WCB01] Matt Welsh, David E. Culler, and Eric A. Brewer, *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*, SOSP'01, ACM Press, 2001, pp. 230–243.

[YDBH10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu, *Parameterised multiparty session types*, FOSSACS, 2010, pp. 128–145.

[YV07] Nobuko Yoshida and Vasco Thudichum Vasconcelos, *Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication*, Electr. Notes Theor. Comput. Sci. **171** (2007), no. 4, 73–93.

Appendix A

Appendix for the Eventful Session π-calculus

A.1 Properties of Subtyping

Proposition A.1.1 (Subtyping Properties)**.** The set of *composable* types of a session type S is defined as: $\mathtt{comp}(S) = \{S' \mid S' \leq \overline{S}\}$.

(i) $\leq$ is a preorder.

(ii) (semantics of $\leq$) $S_1 \leq S_2$ if and only if $\mathtt{comp}(S_2) \subseteq \mathtt{comp}(S_1)$.

Proof. **Part (i).** Transitivity and reflexivity are proved following [Pie02, Theorems 21.3.6–7]. We demonstrate the main cases for session set types.

For transitivity, a relation $\mathscr{R} \subseteq \mathscr{T} \times \mathscr{T}$ is transitive if closed under the monotone function $TR(\mathscr{R}) = \{(x,y) \mid \exists z \in \mathscr{T}.\{(x,z),(z,y)\} \subseteq \mathscr{R}\}$. We note that if $TR(\mathscr{F}(\mathscr{R})) \subseteq \mathscr{F}(TR(\mathscr{R}))$, then the greatest fixed point of $\mathscr{F}$ is transitive, and show $TR(\mathscr{F}(R)) \subseteq \mathscr{F}(TR(\mathscr{R}))$ by taking

$(T,T') \in TR(\mathscr{F}(\mathscr{R}))$. By definition of TR, there exists a T'' such that $(T,T''),(T'',T') \in \mathscr{F}(\mathscr{R})$, and we proceed by cases on T'' to show $(T,T') \in \mathscr{F}(TR(\mathscr{R}))$.

Case: $T'' = \{S''_k\}_{k\in K}$

By definition of $\mathscr{F}$, $(T,T'') \in \mathscr{F}(\mathscr{R})$ implies $T = \{S_i\}_{i\in I}, \forall k \in K, \exists i \in I.(S_i,S''_k) \in \mathscr{R}$. There are two subcases for $(T'',T') \in \mathscr{F}(\mathscr{R})$. First, $T' = \{S'_j\}_{j\in J}, \forall j \in J, \exists k \in K.(S''_k,S'_j) \in \mathscr{R}$. By definition of TR, $\forall j \in J, \exists i \in I.(S_i,S'_j) \in TR(\mathscr{R})$. Hence, by definition of $\mathscr{F}$, $(\{S_i\}_{i\in I},\{S'_j\}_{j\in J}) \in \mathscr{F}(TR(\mathscr{R}))$. Second, $T' = S'$, $|K| = 1, (S''_1,S') \in \mathscr{R}$. By definition of TR, $\exists i \in I.(S_i,S') \in TR(\mathscr{R})$. Hence, by definition of $\mathscr{F}$, $(\{S_i\}_{i\in I},S') \in \mathscr{F}(TR(\mathscr{R}))$.

The other cases are standard, with similar treatment of the subcases where T has the shape $\{S_i\}_{i\in I}$.

For reflexivity, let the identity relation $\mathscr{I} = \{(T,T) \mid T \in \mathscr{T}\}$, and $\mathscr{R} \subseteq \mathscr{T} \times \mathscr{T}$ is $\mathscr{F}$-consistent if $\mathscr{R} \subseteq \mathscr{F}(\mathscr{R})$. By the principle of coinduction, if $\mathscr{I}$ is $\mathscr{F}$-consistent, then the greatest fixed point of $\mathscr{F}$ contains $\mathscr{I}$. To show $\mathscr{I}$ is $\mathscr{F}$-consistent, we take $(T,T) \in \mathscr{I}$ and proceed by cases on T to show $(T,T) \in \mathscr{F}(\mathscr{I})$.

Case: $T = \{S_i\}_{i\in I}$

By definition of $\mathscr{I}$, $\forall I \in I.(S_i,S_i) \in \mathscr{I}$. Hence, by defintion of $\mathscr{F}$, $(\{S_i\}_{i\in I},\{S_i\}_{i\in I}) \in \mathscr{F}(\mathscr{I})$, since the condition $\forall j \in J, \exists i \in I.(S_i,S_j) \in \mathscr{I}$ is trivially satisfied when $I = J$.

The remaining cases are standard.

Part (ii). By Lemma 4.2.1, $S_1 \leq S_2$ iff $\overline{S_1} \geq \overline{S_2}$. But by definition $\overline{S_1} \geq \overline{S_2}$ iff $\mathsf{comp}(S_2) \subseteq \mathsf{comp}(S_1)$, as required. □

A.2 Subject Reduction and Communication and Event Handling Safety

This Appendix relates to the proof of subject reduction for the asynchronous session types typing system, and the communication and event-handling safety.

A.2.1 Weakening and Strengthening

Lemma A.2.1 (Weakening Lemma)**.** Let $\Gamma \vdash P \triangleright \Delta$.

(i) If $X \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$.

(ii) If $u \notin \mathtt{dom}(\Gamma)$, then $\Gamma \cdot u : U \vdash P \triangleright \Delta$.

(iii) If $k \notin \mathtt{dom}(\Delta)$ then $\Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}$.

Proof. Part (i):

For part (i) we apply induction on the definition of ESP process syntax. The base cases are trivial.

We demonstrate the inductive step. Let

$$\begin{aligned} P &= u(x:S).P_1 \\ &\quad \Gamma \vdash P_1 \triangleright \Delta \cdot x : S \end{aligned}$$

From the induction hypothesis we have that

$$\Gamma \cdot X : \Delta' \vdash P_1 \triangleright \Delta \cdot x : S$$

and $x \notin \Gamma$. We can now easily conclude that $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$.

We demonstrate the case for the typecase process.

$$P = \texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$$

From the induction hypothesis we get that for each $i \in I, \Gamma \cdot X : \Delta' \vdash P_i \triangleright \Delta_i$ and $X \notin \Gamma$. It easy to conclude that $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$.

The rest of the induction step cases are similar.

Part (ii):

Part (ii) is similar to part (i).

Part (iii):

For part (iii) we again use induction on the structure of ESP process syntax. It is easy to see the basic step for process $\mathbf{0}$, where we get $\Gamma \vdash \mathbf{0} \triangleright k : \texttt{end}$, from typing rule [Inact].

For the induction step we do a case analysis. Let $P = u(x : S).P_1$. From the induction hypothesis we get that $\Gamma \vdash P \triangleright \Delta k : \texttt{end}$ with $k \notin \texttt{dom}(\Delta)$. We can now easily conclude that $\Gamma \vdash P \triangleright \Delta k : \texttt{end}$.

The rest of the cases are similar. □

Lemma A.2.2 (Strengthening Lemma)**.**

(i) If $X \notin \texttt{fpv}(P)$, then $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright \Delta$.

(ii) If $u \notin \texttt{fn}(P) \cup \texttt{fv}(P)$, then $\Gamma \cdot u : U \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright \Delta$.

(iii) If $k \notin \texttt{fn}(P) \cup \texttt{fv}(P)$ then $\Gamma \vdash P \triangleright \Delta \cdot k : \texttt{end}$ implies $\Gamma \vdash P \triangleright \Delta$.

Proof. Part (i):

For part (i) we apply induction on the definition of ESP process syntax. The base cases are trivial.

We demonstrate the inductive step. Let

$$\begin{aligned} P &= u(x:S).P_1 \\ &\quad \Gamma \cdot X : \Delta' \vdash P_1 \triangleright \Delta \cdot x : S \end{aligned}$$

From the induction hypothesis we have that

$$\Gamma \vdash P_1 \triangleright \Delta \cdot x : S$$

We can now easily conclude that $\Gamma \vdash P \triangleright \Delta$.

Let

$$P = \texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$$

and for each $i \in I, \Gamma \cdot X : \Delta' \vdash P_i \triangleright \Delta_i$. From the induction hypothesis we get that for each $i \in I, \Gamma \vdash P_i \triangleright \Delta_i$ and $X \notin \Gamma$. It easy to conclude that $\Gamma \cdot \vdash P \triangleright \Delta$.

The rest of the induction step cases are similar.

Parts (ii) and (iii):

Parts (ii) and (iii) are similar to part (i). □

Lemma A.2.3 (Substitution Lemma)**.**

(i) If $\Gamma \cdot x : U, \Delta \vdash e : U'$ and $\Gamma \vdash v \triangleright U$, then $\Gamma, \Delta \vdash e\{v/x\} : U'$.

(ii) If $\Gamma, \Delta \cdot x : T \vdash e : U$ and s fresh, then $\Gamma, \Delta \cdot s : S \vdash e\{s/x\} : U$.

(iii) If $\Gamma \cdot x : U \vdash P \triangleright \Delta$ and $\Gamma \vdash v \triangleright U$, then $\Gamma \vdash P\{v/x\} \triangleright \Delta$.

(iv) If $\Gamma \vdash P \triangleright \Delta \cdot k : T$, then $\Gamma \vdash P\{s/k\} \triangleright \Delta \cdot s : T$.

Proof. Parts (i) and (ii):

Parts (i) and (ii) are proved with a simple induction on the structure of expressions e.

Part (iii):

We apply induction on the definition of ESP process syntax. The base cases are trivial. We demonstrate the inductive step. For

$$P = u(x : S).P_1$$

we have that

$$\Gamma \cdot x : U \vdash P_1 \triangleright \Delta$$

From the induction hypothesis we have that $\Gamma \cdot v : U \vdash P_1 \triangleright \Delta$. We can now easily conclude that $\Gamma \cdot v : U \vdash P \triangleright \Delta$.

For the typecase case let

$$P = \texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$$

From the induction hypothesis we get that for each $i \in I, \Gamma \cdot x : U \vdash P_i \triangleright \Delta_i$ and $\Gamma \vdash v : U$. It is easy to see that $\Gamma \vdash P\{v/x\} \triangleright \Delta$.

The rest of the cases are similar.

Part (iv):

For part (iv) we demonstrate the interesting case for the `typecase` construct.

Let

$$\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i\in I} \triangleright \Delta \cdot k' : T$$

From the induction hypothesis we get that for each $i \in I, \Gamma \vdash P_i\{s/k'\} \triangleright \Delta \cdot s : T$. From typing rule [Typecase] we conclude that $\Gamma \vdash \texttt{typecase}\ k\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i\in I}\{s/k'\} \triangleright \Delta \cdot s : T$. Note that the same results holds if $k' = k$.

The rest of the cases are trivial. □

A.2.2 Subject Reduction

Theorem A.2.1 (Subject Congruence and Reduction)**.** (Theorem 4.2.1)

(i) *If* $\Gamma \vdash P \triangleright \Delta$ *and* $P \equiv Q$*, then* $\Gamma \vdash Q \triangleright \Delta$.

(ii) *If* $\Gamma \vdash P \triangleright \Delta$ *with* Δ *well-configured and* $P \longrightarrow Q$*, then we have* $\Gamma \vdash Q \triangleright \Delta'$ *such that* $\Delta \rightarrow\!\!\!\!\rightarrow \Delta'$ *and* Δ' *is well-configured.*

Proof. Part (i):

The proof for (i) subject congruence uses a case analysis on the structural congruence rule and it is standard. We demonstrate one basic case with the rest being similar. Let $\Gamma \vdash P \mid Q \triangleright \Delta$ and $P \mid Q \equiv Q \mid P$. From typing rule [Cong] we have that $\Delta = \Delta_1 \cup \Delta_2$ with $\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta_2$. It is trivial to see that $\Gamma \vdash Q \mid P \triangleright \Delta$ because $\Delta_1 \cup \Delta_2 = \Delta_2 = \Delta_1$. Rest of the cases are trivial.

Part (ii):

For (ii) subject reduction, we prove by induction on the reduction relation.

Case: [Request1]

$\Gamma \vdash \overline{a}(x).P \triangleright \Delta \longrightarrow (\nu\ s)(P\{\overline{s}/x\} \mid \overline{a}\langle s\rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) \triangleright \Delta'$. By rule (Req), we have that

$\Gamma \vdash P \triangleright \Delta \cdot \overline{x} : \overline{S}$. By rules (InQ, OutQ), we obtain that $\Gamma \vdash s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \emptyset$. Then by rule (Areq), we have $\Gamma \vdash \overline{a}\langle s \rangle \triangleright s : S$. We now apply rule (Conc) to obtain $\Gamma \vdash P\{\overline{s}/x\} \mid \overline{a}\langle s \rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \Delta \cdot s : S \cdot \overline{s} : \overline{S}$. Rule (Sres) gives us $\Gamma \vdash (\nu\ s)(P\{\overline{s}/x\} \mid \overline{a}\langle s \rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) \triangleright \Delta$, as required.

Case: [Request2]

$\Gamma \vdash \overline{a}\langle s \rangle \mid a[\vec{s}] \triangleright \vec{s} : \vec{S} \cdot s : S \longrightarrow a[\vec{s} \cdot s] \triangleright \vec{s} : \vec{S} \cdot s : S$. We type the processes that compose the left hand side process using typing rules (Queue), (Areq). By rule (Conc) and the definition of $*$ we obtain the typing $\vec{s} : \vec{S} \cdot s : S$. The right hand side is typed using typing rule (Areq) to obtain the same result.

Case: [Accept]

$\Gamma \vdash a(x).P \mid a[s \cdot \vec{s}] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S \longrightarrow P\{s/x\} \mid a[\vec{s}] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S$. For the left hand side, we use rules (Queue), (Acc) and (Conc), to get the typing result. From rule (Acc) we have that $\Gamma \vdash P\{s/x\} \triangleright \Delta$. From here is easy to find the same typing for the right hand side.

Case: [Send] (Value)

$\Gamma \vdash s!\langle v \rangle; P \mid s[S_1, \mathtt{o} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P \mid s[S_2, \mathtt{o} : v \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_2]$, where $\Gamma \vdash \vec{h} : \vec{T}, \Gamma \vdash v : T$. For the left hand side we type $\Gamma \vdash s!\langle v \rangle; P \triangleright \Delta \cdot s : !\langle T \rangle; S'$ and $\Gamma \vdash s[!\langle T \rangle; S'_1, \mathtt{o} : \vec{h}] \triangleright s : O \cdot s[!\langle T \rangle; S'_1]$. Using (Conc) we get $!\langle T \rangle; S' * O = S$. Now if we type the right hand side we get $\Gamma \vdash s!\langle v \rangle; P \triangleright \Delta \cdot s : S'$ and $\Gamma \vdash s[S'_1, \mathtt{o} : v \cdot \vec{h}] \triangleright s : !\langle T \rangle; O \cdot s[S'_1]$. We compose to get $S' * !\langle T \rangle; O = !\langle T \rangle; S' * O = S$ and $S'_1 = S_2$.

Case: [Receive] (Value)

$\Gamma \vdash s?(x); P \mid s[S_1, \mathtt{i} : v \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P\{v/x\} \mid s[S_2, \mathtt{i} : \vec{s}] \triangleright \Delta \cdot s : S \cdot s[S_2]$. For the left hand side we have $\Gamma \vdash s?(x); P \triangleright \Delta \cdot s : ?(T); S'$ and $\Gamma \vdash s[?(T); S'_1, \mathtt{i} : v \cdot \vec{h}] \triangleright s : ?(T); I \cdot s[?(T); S'_1, \mathtt{i} :]$. We compose and get $?(T); S' * ?(T); I = S' * I = S$. For the right hand side we have $\Gamma \vdash P \triangleright \Delta \cdot s : S'$ and $\Gamma \vdash s[S_2, \mathtt{i} : \vec{h}] \triangleright s : I \cdot s[S_2, \mathtt{i} :]$. By composition we get $S' * I = S$ and $S'_1 = S_2$.

Case: [Receive] (Delegation)

$\Gamma \vdash s?(x); P \mid s[S_1, \mathtt{i} : s' \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s' : S' \cdot s[S_1] \longrightarrow P\{s'/x\} \mid s[S_2, \mathtt{i} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s' : S' \cdot s[S_2]$.

We have $\Gamma \vdash s?(x);P \triangleright \Delta \cdot s :?(S');S''$, $\Gamma \vdash s[S_1, \mathtt{i} : s' \cdot \vec{h}] \triangleright s :?(S');I \cdot s' : S' \cdot s[?(S');S'_1]$ and $\Delta \cdot s :?(S');S'' * s :?(S');I \cdot s' : S' \cdot s[?(S');S'_1] = \Delta \cdot s :?(S');S \cdot s' : S' \cdot s[?(S');S'_1]$. For the right hand side we have $\Gamma \vdash P\{s'/x\} \triangleright \Delta \cdot s : S'' \cdot s' : S'$, $\Gamma \vdash s[S_2, \mathtt{i} : \vec{h}] \triangleright s : I \cdot s[S'_1]$ and $\Delta \cdot s : S'' \cdot s' : S' * s : I \cdot s[S'_1] = \Delta \cdot s :?(S');S \cdot s' : S' \cdot s[S'_1]$.

Case: [Send] (Delegation)

Similar to the above case.

Case: [Sel]

$\Gamma \vdash s \oplus v;P \mid s[S_1, \mathtt{o} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P \mid s[S_2, \mathtt{o} : l \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_2]$. The proof is similar to [Send] case.

Case: [Bra]

$\Gamma \vdash s\&\{l_i : P_i\}_{i \in I} \mid s[S_1, \mathtt{i} : l_k \cdot \vec{h}] \triangleright \Delta \cdot s : S' \cdot s[S_1] \longrightarrow P_k \mid s[S_2, \mathtt{i} : \vec{s}] \triangleright \Delta \cdot s : S' \cdot s[S_2]$ where $S' = S_k * M_{\mathtt{i}}$ and $\Gamma \vdash s[S_1, \mathtt{i} : \vec{s}] \triangleright s : M_{\mathtt{i}} \cdot s[S_1]$. The proof is similar to the (Receive) case.

Case: [Comm]

$\Gamma \vdash P \mid s[S_1, \mathtt{o} : \vec{h} \cdot v] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'}]\varepsilon \triangleright \Delta_1 \longrightarrow P \mid s[S_1, \mathtt{o} : \vec{h}] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'} \cdot v] \triangleright \Delta_1$. $\Gamma \vdash P \triangleright \Delta \cdot s : S_1 \cdot \overline{s} : S_2$ with $\Gamma \vdash P \mid s[!\langle T\rangle;S_1, \mathtt{o} : \vec{h} \cdot v] \mid \overline{s}[?(T);S_2, \mathtt{i} : \vec{h'}] \triangleright \Delta \cdot s :!\langle T\rangle;S \cdot \overline{s} :?(T);\overline{S}$ from the induction hypothesis. If we type the right hand side we have that $\Gamma \vdash P \mid s[S_1, \mathtt{o} : \vec{h}] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'} \cdot v] \triangleright \Delta \cdot s : S \cdot \overline{s} : \overline{S}$ as required.

Case: [Typecase]

$\Gamma \vdash \mathtt{typecase}\ s\ \mathtt{of}\ \{S_i : P_i\}_{i \in I} \mid s[S_k, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Delta_1 \longrightarrow P_k \mid s[S_k, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Delta_2$.

$\Gamma \vdash \mathtt{typecase}\ s\ \mathtt{of}\ \{S_i : P_i\}_{i \in I} \mid s[S_k, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Delta \cdot s : \{S_i\}_{i \in I}$. From the right hand side of the reduction, we have $\Gamma \vdash P_k \mid s[S_k, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Delta \cdot s : S_k$. Since $S_k \leq \{S_i\}_{i \in I}$, we use [Subs] to obtain $\Delta_1 = \Delta_2$ and Δ_2 well-configured from the induction hypothesis.

Case: [arrive]

$\Gamma \vdash \mathtt{if}\ \mathtt{arrive}\ s\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid s[\mathtt{i} : \vec{h}] \triangleright \Delta \cdot s : T * s : M \longrightarrow P \mid s[\mathtt{i} : \vec{h}] \triangleright \Delta' \cdot s : T * s : M$.

$\Gamma \vdash \mathtt{if}\ \mathtt{arrive}\ s\ \mathtt{then}\ P\ \mathtt{else}\ Q \triangleright \Delta \cdot s : T$. From [If] we have that $\Gamma \vdash P \triangleright \Delta \cdot s : T$ and $\Gamma, \Delta \vdash \mathtt{arrive}\ s \triangleright \mathtt{bool}$ so, $\Gamma \vdash P \mid s[\mathtt{i} : \vec{s}] \triangleright \Delta \cdot s : T * s : M$ and thus $\Delta = \Delta'$ as required. □

A.2.3 Communication Safety

Theorem A.2.2 (Communication and Event-Handling Safety)**.** *(Theorem 4.2.2)* If P is a well-typed process, then P never reduces to an error.

Proof. Communication safety follows as a corollary from subject reduction (Theorem 4.2.1). Assuming the reduction of a typable process to an error (page 101), we show that the error is not typable, thus leading to a contradiction. We demonstrate key cases for `typecase` and `arrive`, corresponding to cases (h) and (f) in the definition of s-redexes (page 101). These cases ensure that a well-typed process does not reduce to a stuck `typecase` term where the current active type of the session cannot be matched to any of the specified type cases, nor a term in which the `arrive` predicate is used to check the arrival of a message of an unexpected type.

Assume a process $P \longrightarrow P'$, where $\Gamma \vdash P \triangleright \Delta$ and Δ is well-configured. By Theorem 4.2.1, $\Gamma \vdash P' \triangleright \Delta'$, $\Delta \longrightarrow \Delta'$ and Δ' is well-configured. Say P' is an error. Then P' contains, up to structural congruence, a term Q that is the parallel composition of two s-processes that do not form an s-redex. Note that the definition of the $*$ operator and (QConc) implicitly prevent the parallel composition of two s-processes from being well-typed unless one term is an s-configuration and the other is either an $\overline{s}$-configuration or an s-process that is not a configuration. We proceed by cases to show Q, and thus P', is not typable.

Case: $Q = \texttt{typecase}\ s\ \texttt{of}\ \{(s_i : S_i) : P_i\}_{i\in I} \mid s[S]$ where $\nexists i \in I.S_i \leq S$.
To type Q, rule (QConc) must compose for $\Delta'(s)$ some S', where $\{S_i\}_{i\in I} \leq S'$, and $M\,[S]$, where M is message type of the s-configuration. By definition of $*$ composition of linear environments, $S' = S$, contradicting $\nexists i \in I.S_i \leq S$.

Case: $Q = E[\texttt{arrive}\ s\ v] \mid s[?(U);S, \texttt{i} : \vec{h}]$ with v of type U, and $\vec{h} = v' \cdot \vec{h}'$, v' not of type U.
Consider the subcase where the E-context is the `if`-term (the others are similar). By (If) and (AVal), the type of s is of the shape $?(U);S'$. However, to type Q, rule (QConc) must compose

for $\Delta'(s)$ some $?(U);S''$, where $S' \leq S''$, and $?(U');M_{\mathtt{i}}\,[?(U);S]$, where $U' \neq U$, in which case the $*$ operator is not defined. □

Before we proceed, note that:

Lemma A.2.4. Let $P = Q \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]$ and $\Gamma \vdash P \triangleright \Delta$ with Δ well-configured. Then if $P \longrightarrow^* Q' \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \vec{h_2}]$ then $\vec{h_1} = \varepsilon$ or $\vec{h_2} = \varepsilon$.

The above lemma means one of queues is always empty during executions.

Proof. The proof is by induction on the length of $\twoheadrightarrow$. The basic step is trivial. For the inductive step we three cases:

(i) $\Gamma \vdash P_2 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \Delta$.

(ii) $\Gamma \vdash P_2 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \varepsilon] \triangleright \Delta$.

(iii) $\Gamma \vdash P_2 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \vec{h_1}] \triangleright \Delta$.

with Δ well-configured in all three cases (Subject Reduction Theorem 4.2.1) and $|\vec{h_1}| \geq 1$.

We prove part (ii), with parts (i) and (iii) being similar.

Part (ii):

Let

$$P_2 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \varepsilon] \longrightarrow P_2' \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : v]$$

This implies that $P_2 = (\nu\, \vec{n})(P_3 \mid s!\langle v\rangle; P_4)$ or $P_2 = (\nu\, \vec{n})(P_3 \mid s \oplus l; P_4)$ with $s \notin \vec{n}$. Because the input queue is non-empty, the induction hypothesis reduction would be:

$$P_1 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \twoheadrightarrow P_1' = (\nu\, \vec{n'})(Q_1 \mid \bar{s}!\langle w\rangle; Q_2 \mid s!\langle v\rangle; P_4 \mid s[\mathtt{i} : \vec{h_1'}, \mathtt{o} : \varepsilon])$$

The above is further reduced to:

$$P_1' \twoheadrightarrow P_2 = (\nu\, \vec{n})(Q_1 \mid Q_2 \mid s!\langle v\rangle; P_4 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : v])$$

with $P_3 = Q_1 \mid Q_2$. Obviously such P_1' is untypable since the endpoints of s do not have dual types. This leads to a contradiction. The rest of the cases rely on the untypability of reduction $\twoheadrightarrow$ to prove the case by contradiction. □

A.3 Bisimulation Properties

A.3.1 Proof for Theorem 4.3.1

Theorem (Coincidence): $\approx$ and $\cong$ coincide.

The above theorem requires to show the equality into two directions.

Lemma A.3.1 (Soundness)**.** $P \approx Q$ implies $P \cong Q$.

Proof. Reduction closeness and barb observation properties are easy to be verified. The only remaining property is showing that $\approx$ is a congruence.

Congruence for the output prefix, restriction construct, `if /else` construct and recursion construct are easy to be verified. Input congruence is similar to output congruence, since we are dealing with programs, which are processes without free variables. We give the result for congruence of the parallel operator.

Parallel Congruence

Assume the relation:

$$\mathscr{S} = \{((\nu\ \tilde{a},\tilde{s})(P \mid R),(\nu\ \tilde{a},\tilde{s})(Q \mid R)) \mid P \approx Q, \forall R \cdot P \mid R, Q \mid R \text{ are typable}, \forall \tilde{a},\tilde{s}\} \quad \text{(A.1)}$$

We show that $\mathscr{S}$ is a typed relation.

Since $P \approx Q$ we have that $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash Q \triangleright \Delta'$ with $\Delta \rightleftharpoons \Delta'$. Since P, Q are localised and R is localised and $P \mid R, Q \mid R$ are typable then $\mathtt{dom}(\Delta) \cap \mathtt{dom}(\Delta') = \emptyset$. We Use [Conc] and the $*$ definition, we obtain the result.

We show that S is a bisimulation. There are three cases:

Case (1) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \xrightarrow{\ell} P' \mid R \triangleright \Delta_1'$. Then $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta_P'$.

By the definition of $\mathscr{S}$, we have that $\Gamma \vdash Q \triangleright \Delta_Q \stackrel{\ell}{\Longrightarrow} Q' \triangleright \Delta_Q'$. Thus we have $\Gamma \vdash Q \mid R \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} Q' \mid R \triangleright \Delta_2'$.

Case (2) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \xrightarrow{\ell} P \mid R' \triangleright \Delta_1'$. Then $\Gamma \vdash R \triangleright \Delta_R \xrightarrow{\ell} R' \triangleright \Delta_R'$.

By the above, we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \xrightarrow{\ell} Q \mid R' \triangleright \Delta_2'$. By $\Delta_1' \rightleftharpoons \Delta_2'$, we conclude $P \mid R \approx Q \mid R$ as required.

Case (3) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \longrightarrow (\nu\ \tilde{a},\tilde{s})(P' \mid R') \triangleright \Delta_1'$. Then we have

$$\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta_P' \quad \text{(A.2)}$$

By the definition of S, we have:

$$\Gamma \vdash Q \triangleright \Delta_Q \Longrightarrow \xrightarrow{\ell} \Longrightarrow Q' \triangleright \Delta_Q' \quad \text{(A.3)}$$

By (A.3), we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \Longrightarrow (\nu\ \tilde{a},\tilde{s})(Q' \mid R') \triangleright \Delta_2'$. Then by $\Delta_1' \rightleftharpoons \Delta_2'$, we have $P \mid R \approx Q \mid R$, as required. □

The proof for the completeness direction follows the technique shown in [Hen07]. However we need to adapt it to session and input/output configurations.

Definition A.3.1 (definability)**.** An external action ℓ is *definable* if for a set of names N and action succ, $\notin N$ there is a *testing process* $T\langle N, \text{succ}, \ell\rangle$ with the property that for every process P and $\texttt{fn}(P) \subseteq N$:

- $\Gamma \vdash P \triangleright \Delta_1 \xrightarrow{\ell} P' \triangleright \Delta_1'$ implies that
  $\Gamma \vdash T\langle N, \text{succ}, \ell\rangle \mid P \triangleright \Delta \twoheadrightarrow (\nu\ \texttt{bn}(\ell), b)(\text{succ}[\texttt{o} : \texttt{bn}(\ell)] \mid R \mid P') \triangleright \Delta'$.
- $\Gamma \vdash T\langle N, \text{succ}, \ell\rangle \mid P \triangleright \Delta \twoheadrightarrow Q \triangleright \Delta'$, where $Q \Downarrow_{\text{succ}}$ implies that
  $Q = (\nu\ \texttt{bn}(\ell), b)(\text{succ}[\texttt{o} : \texttt{bn}(\ell)] \mid R \mid P')$ where $\Gamma \vdash P \triangleright \Delta_1 \overset{\ell}{\Longrightarrow} P' \triangleright \Delta_1'$.

where $R = b(x).R'$ or $R = \mathbf{0}$.

Note that $b(x).R$ is used to keep the composition $P \mid T\langle N, \text{succ}, \ell\rangle$ typable. Also $R \not\xrightarrow{\ell}$ either due to the restriction of b, or because $R = \mathbf{0}$.

Lemma A.3.2. Every external action is definable.

Proof. The input action cases are straightforward:

1. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{a\langle s\rangle} P' \triangleright \Delta'$ then $T\langle \emptyset, \text{succ}, a\langle s\rangle\rangle = \overline{a}(x).R \mid \text{succ}[\texttt{o} : \texttt{tt}]$.
2. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s?\langle v\rangle} P' \triangleright \Delta'$ then $T\langle \emptyset, \text{succ}, s?\langle v\rangle\rangle = (\nu\ b)(s!\langle v\rangle; b(x).R) \mid \text{succ}[\texttt{o} : \texttt{tt}]$.
3. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s\&l} P' \triangleright \Delta'$ then $T\langle \emptyset, \text{succ}, s\&l\rangle = (\nu\ b)(s \oplus l; b(x).R) \mid \text{succ}[\texttt{o} : \texttt{tt}]$.

The requirements of Definition A.3.1 can be verified with simple transitions.

Output actions cases:

1. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\overline{a}\langle s \rangle} P' \triangleright \Delta'$ then we have:

$$T\langle\{s\}, \text{succ}, a\langle s\rangle\rangle = (\nu\ b)(a(x).(\texttt{if}\ x = s\ \texttt{then}\ \text{succ}!\langle x\rangle; R\ \texttt{else}\ b(x).\text{succ}!\langle x\rangle; R)) \mid \text{succ}[\texttt{i} : \varepsilon, \texttt{o} : \varepsilon] \mid a[\varepsilon]$$

2. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s!\langle b \rangle} P' \triangleright \Delta'$ then we have that

$$T\langle\{b\}, \text{succ}, s!\langle b\rangle\rangle = (\nu\ b)(s?(x); (\texttt{if}\ x = b\ \texttt{then}\ \text{succ}!\langle x\rangle; b(x).R\ \texttt{else}\ b(x).(\text{succ}!\langle x\rangle; R)) \mid \text{succ}[\texttt{i} : \varepsilon, \texttt{o} : \varepsilon]$$

3. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s!(b)} P' \triangleright \Delta'$ then we have that:

$$T\langle\{b\}, \text{succ}, s!(b)\rangle = (\nu\ b)(s?(x); (\texttt{if}\ x = b\ \texttt{then}\ \text{succ}!\langle x\rangle; b(x).R\ \texttt{else}\ b(x).(\text{succ}!\langle x\rangle; R)) \mid \text{succ}[\texttt{i} : \varepsilon, \texttt{o} : \varepsilon]$$

4. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s \oplus l_k} P' \triangleright \Delta'$ then we have that:

$$T\langle\emptyset, \text{succ}, s \oplus l_k\rangle = (\nu\ b)(s\&\{l_k : \text{succ}!\langle \texttt{tt}\rangle; R, l_i : b(x).R\}_{i \in I}), 1 \leq i \leq n$$

Again the requirements of Definition A.3.1 can be verified by simple transitions for each case. □

Lemma A.3.3. If *succ* is fresh, $b \in \vec{a} \cdot \vec{s}$ and

$$\Gamma \vdash (\nu \vec{a}, \vec{s}, b)(P \mid succ[\texttt{o} : a'] \mid b[x](R).) \triangleright \Delta_1 \cong (\nu \vec{a}, \vec{s}, b)(Q \mid succ[\texttt{o} : a'] \mid b[x](R).) \triangleright \Delta_2 \quad \text{(A.4)}$$

then

$$\Gamma \vdash P \triangleright \Delta_P \cong Q \triangleright \Delta_Q \tag{A.5}$$

Proof. Let relation

$$\begin{aligned}\mathscr{S} \;=\; & \{(\Gamma \vdash P \triangleright \Delta_P, \Gamma \vdash Q \triangleright \Delta_Q) \mid \\ & \Gamma \vdash (\nu \vec{a}, \vec{s}, b)(P \mid \mathrm{succ}[\mathsf{o} : a'] \mid b[x](R).) \triangleright \Delta_1 \\ & \cong (\nu \vec{a}, \vec{s}, b)(Q \mid \mathrm{succ}[\mathsf{o} : a'] \mid b[x](R).) \triangleright \Delta_2, \quad \text{succ is fresh}\}\end{aligned}$$

We will show that the contextual properties hold in $\mathscr{S}$.

Typing: It should hold that $\mathscr{S}$ is a typed relation. From the definition of $\mathscr{S}$, we have that: $\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta \approx (\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta'$, $\Delta \rightleftharpoons \Delta'$. From here, by using typing rules (Nres), (Sres), (Conc), we get the required result.

Reduction Closedness: $\mathscr{S}$ is reduction closed by the freshness of succ. We cannot observe a reduction on succ or on $b(x).R$, so we conclude that if $\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta \twoheadrightarrow (\nu\ \tilde{a}, \tilde{s}, b)(P' \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta'$ implies $\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta \twoheadrightarrow (\nu\ \tilde{a}, \tilde{s})(Q' \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \triangleright \Delta'$ then $\Gamma \vdash P \triangleright \Delta_1 \twoheadrightarrow P' \triangleright \Delta_P$ implies $\Gamma \vdash Q \triangleright \Delta_1 \twoheadrightarrow Q' \triangleright \Delta_Q$

Preserve Observation: We do a case analysis on the cases where $P \downarrow_m$.

If $P \downarrow_m$, $m \notin \tilde{a} \cdot \tilde{s}$ and $(\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathsf{o} : a'] \mid b(x).R) \downarrow_m$ then $(\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathsf{o} : a] \mid b(x).R) \Downarrow_m$. From the definition of $\mathscr{S}$ and the freshness of succ, we conclude $Q \Downarrow_m$.

If $P \downarrow_m$, $m \notin \tilde{a} \cdot \tilde{s}$ and $(\nu\ \tilde{a},\tilde{s},b)(P \mid \text{succ}[\text{o} : a'] \mid b(x).R) \not\downarrow_m$ then by the environment typing transition we have that m is a session occurring free in $\text{succ}[\text{o} : a'] \mid b(x).R$, and also $(\nu\ \tilde{a},\tilde{s},b)(Q \mid \text{succ}[\text{o} : a] \mid b(x).R) \not\Downarrow_m$. The case where $Q \not\Downarrow_m$ does not hold, because it would be possible to have $(\nu\ \tilde{a},\tilde{s},b)(Q \mid \text{succ}[\text{o} : a] \mid b(x).R) \mid Q'$ with Q' having as a free name session m and have a typable process. But composition $(\nu\ \tilde{a},\tilde{s},b)(P \mid \text{succ}[\text{o} : a'] \mid b(x).R) \mid Q'$ is untypable because $P \downarrow_m$, thus breaking reduction congruence. This results to the conclusion that $Q \downarrow_m$.

Context Property: The interesting case is the parallel composition. We will show that if $\Gamma \vdash P \triangleright \Delta_P \mathscr{S} \Gamma \vdash Q \triangleright \Delta_Q$. Then for arbitrary process R we have that $\Gamma \vdash P \mid P_1 \triangleright \Delta'_P \mathscr{S}\ Q \mid P_1 \triangleright \Delta'_Q$

To show this, it is enough to show that
$\Gamma \vdash (\nu\ \tilde{a},\tilde{s},b)(P \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta''_P \cong (\nu\ \tilde{a},\tilde{s},b)(Q \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta''_Q$, considering that succ may occur in P_1 and succ' is fresh.

To prove this assume the process $T\langle \emptyset, \text{succ}', \ell \rangle = \text{succ}?(x); (\text{succ}'!\langle x \rangle; \mathbf{0} \mid P'_1) \mid \text{succ}'[\text{i} : \varepsilon, \text{o} : \varepsilon]$, where $P_1 = P_1\{a'/x\}$.

From the contextual property of the theorem's assumption and simple reductions, we have that:
$\Gamma \vdash (\nu\ \tilde{a},\tilde{s},b)(P \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta_1 \cong \Gamma \vdash (\nu\ \tilde{a},\tilde{s},b)(Q \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta'_1$.

We need to verify that
$\Gamma \vdash (\nu\ \tilde{a},\tilde{s},b)(P \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta_1 \approx (\nu\ \tilde{a},\tilde{s},b)(P \mid P_1 \mid \text{succ}'[\text{o} : a'] \mid R) \triangleright \Delta'_1$, which is simple because $R \approx \mathbf{0}$. By using Lemma A.3.1 we get the result. □

We are know ready to prove the completeness direction.

Lemma A.3.4 (Completness)**.** $P \cong Q$ implies $P \approx Q$

Proof. For the proof we show that if

$$\Gamma \vdash P \triangleright \Delta_P \quad \cong \quad Q \triangleright \Delta_Q \text{ and} \tag{A.6}$$

$$\Gamma \vdash P \triangleright \Delta_P \quad \xrightarrow{\ell} \quad P' \triangleright \Delta'_P \tag{A.7}$$

then $\Gamma \vdash Q \triangleright \Delta_Q \stackrel{\ell}{\Longrightarrow} Q' \triangleright \Delta'_Q$ and $\Gamma \vdash P' \triangleright \Delta'_P \cong Q' \triangleright \Delta'_Q$

Suppose (A.6) and (A.7). Then there are two cases.

If $\ell = \tau$ then by reduction closeness of $\cong$ the result follows.

In the case where ℓ is an external action we can do a definability test for P by choosing the appropriate test $T\langle N, \text{succ}, l\rangle$.

Because $\cong$ is context preserving we have that $\Gamma \vdash P \mid T\langle N, \text{succ}, l\rangle \triangleright \Delta_{PT} \cong Q \mid T\langle N, \text{succ}, l\rangle \triangleright \Delta_{QT}$. By Lemma A.3.2 we have that $\Gamma \vdash P \mid T\langle N, \text{succ}, l\rangle \triangleright \Delta_{PT} \Longrightarrow (\nu\ \mathtt{bn}(\ell))(\text{succ}[\mathtt{o} : \mathtt{bn}(\ell)] | P') \triangleright \Delta$ thus by the definition of $\cong$ (Definition 4.3.2), we have that $\Gamma \vdash T\langle N, \text{succ}, l\rangle \mid Q \triangleright \Delta_{QT} \Longrightarrow R \triangleright \Delta'$. According to the second part of the Definition A.3.1, we can write:

$$\Gamma \vdash Q' \quad = \quad \Gamma \vdash (\nu\ \mathtt{bn}(\ell))(\text{succ}[\mathtt{o} : \mathtt{bn}(\ell)] \mid b(x).R \mid Q'') \triangleright \Delta'' \tag{A.8}$$

$$\Gamma \vdash Q \triangleright \Delta_Q \quad \stackrel{\ell}{\Longrightarrow} \quad Q' \triangleright \Delta'_Q \tag{A.9}$$

Now we can derive
$\Gamma \vdash (\nu\ \mathtt{bn}(\ell), b)(\text{succ}[\mathtt{o} : \mathtt{bn}(\ell)] \mid b(x).R \mid P') \triangleright \cong (\nu\ \mathtt{bn}(\ell), b)(\text{succ}[\mathtt{o} : \mathtt{bn}(\ell)] \mid b(x).R \mid Q') \triangleright \Delta''$.
By Lemma A.3.3 we conclude that:

$$\Gamma \vdash P' \triangleright \Delta'_P \quad \cong \quad Q' \triangleright \Delta'_Q \tag{A.10}$$

We began with the assumption that $\Gamma \vdash P \triangleright \Delta_P \cong \Gamma \vdash Q \triangleright \Delta_Q$ and we concluded to (A.9) and (A.10). Thus $\cong$ implies $\approx$. □

A.4 Determinacy and Confluence

A.4.1 Proof for Lemma 3.3.1

Before we proceed with the proof of Lemma 3.3.1 we prove the following useful Lemma.

Lemma A.4.1.

- If $\Gamma \vdash P \triangleright \Delta \longrightarrow\xrightarrow{\ell} \Gamma \vdash P' \triangleright \Delta'$ and ℓ is an input action then $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell}\longrightarrow \Gamma \vdash P' \triangleright \Delta'$.
- If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell}\longrightarrow \Gamma \vdash P' \triangleright \Delta'$ and ℓ is an output action then $\Gamma \vdash P \triangleright \Delta \longrightarrow\xrightarrow{\ell} \Gamma \vdash P' \triangleright \Delta'$.

Proof. For the first part there are two cases

Case (1) P has the form $P = R \mid a[\vec{s}]$. $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \longrightarrow R' \mid a[\vec{s'}] \triangleright \Delta' \xrightarrow{a?\langle s\rangle} R' \mid a[\vec{s'} \cdot s] \triangleright \Delta'$ Now we can observe $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \xrightarrow{a?\langle s\rangle} \Gamma \vdash R \mid a[\vec{s} \cdot s] \triangleright \Delta \longrightarrow R' \mid a[\vec{s'} \cdot s] \triangleright \Delta'$ to conclude.

Case (2) Input communication takes place on a session channel. It is similar using a session queue.

For the second there are two cases.

Case (1) The action happens on a shared name. $\Gamma \vdash P \mid \overline{a}\langle s\rangle \triangleright \Delta \xrightarrow{a!\langle s\rangle} P \triangleright \Delta' \longrightarrow P' \triangleright \Delta''$.

From this we can always conclude that $\Gamma \vdash P\overline{a}\langle s\rangle \triangleright \Delta \longrightarrow P' \mid \overline{a}\langle s\rangle \triangleright \Delta''' \xrightarrow{a!\langle s\rangle} P' \triangleright \Delta''$. Hence we conclude the case.

Case (2) Output communication takes place on a session channel. Similar arguments by using a session queue. □

We are now ready to prove Lemma 3.3.1.

Proof. The proof in both parts is done by induction on the length of the silent transition. The base case is trivial.

For the first part of Lemma 3.3.1 we have:

$\Gamma \vdash P \triangleright \Delta \Longrightarrow \longrightarrow \xrightarrow{\ell} \Longrightarrow P' \triangleright \Delta'$. We use the first part of Lemma A.4.1 to permute actions $\longrightarrow$ and $\xrightarrow{\ell}$ and get $\Gamma \vdash P \triangleright \Delta \Longrightarrow \xrightarrow{\ell} \longrightarrow \Longrightarrow P' \triangleright \Delta'$. Then by the use of the induction hypothesis we get $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \Longrightarrow \longrightarrow \Longrightarrow P' \triangleright \Delta'$ as required.

The second part of the Lemma 3.3.1 follows similar arguments: $\Gamma \vdash P \triangleright \Delta \Longrightarrow \xrightarrow{\ell} \longrightarrow \Longrightarrow P' \triangleright \Delta'$. We use the second part of Lemma A.4.1 and then the induction hypothesis to permute as required: $\Gamma \vdash P \triangleright \Delta \Longrightarrow \longrightarrow \Longrightarrow \xrightarrow{\ell} P' \triangleright \Delta'$. □

A.4.2 Proof for Lemma 4.3.2

Proof. The proof considers induction on the length of $\Longrightarrow_s$ transition. The basic step is trivial. For the induction step we do a case analysis on $\longrightarrow_s$ transition.

Case: Receive.
By the typability of P, we have that $P' = s?(x);Q \mid s[\mathtt{i} : v \cdot \vec{h}] \mid R \longrightarrow_s P'' = Q\{v/x\} \mid s[\mathtt{i} : \vec{h}] \mid R$.

From the induction step, we have that $P \approx P'$. To show that $P \approx P''$ we need to show that $P' \approx P''$. We will use the fact that bisimulation is a congruence. Consider $R \approx R$ and $s?(x);Q \mid s[\mathtt{i} : v \cdot \vec{h}] \approx Q\{v/x\} \mid s[\mathtt{i} : \vec{h}]$. Due to $s \notin \mathtt{fn}(R)$ we can compose bisimilar processes in parallel and get that $P' \approx P''$ as required.

The rest of the cases follow similar arguments. □

A.4.3 Proof for Lemma 4.3.3

Proof. The result is an easy case analysis on all the possible combinations of ℓ_1, ℓ_2.

We give an interesting case. Let $(\nu\ a)(P \mid s_1[\mathsf{o} : \vec{h_1} \cdot a] \mid s_2[\mathsf{o} : \vec{h_2} \cdot a]) \xrightarrow{s_1!\langle a \rangle} P \mid s_1[\mathsf{o} : \vec{h_1}] \mid s_2[\mathsf{o} : \vec{h_2} \cdot a]$ and $(\nu\ a)(P \mid s_1[\mathsf{o} : \vec{h_1} \cdot a] \mid s_2[\mathsf{o} : \vec{h_2} \cdot a]) \xrightarrow{s_2!\langle a \rangle} P \mid s_1[\mathsf{o} : \vec{h_1} \cdot a] \mid s_2[\mathsf{o} : \vec{h_2}]$. Now it is easy to see that $P \mid s_1[\mathsf{o} : \vec{h_1}] \mid s_2[\mathsf{o} : \vec{h_2} \cdot a] \xrightarrow{s_2!\langle a \rangle} P \mid s_1[\mathsf{o} : \vec{h_1}] \mid s_2[\mathsf{o} : \vec{h_2}]$ and $P \mid s_1[\mathsf{o} : \vec{h_1} \cdot a] \mid s_2[\mathsf{o} : \vec{h_2}] \xrightarrow{s_1!\langle a \rangle} P \mid s_1[\mathsf{o} : \vec{h_1}] \mid s_2[\mathsf{o} : \vec{h_2}]$ as required. □

A.4.4 Proof for Lemma 4.3.4

Proof. There are two cases:

Case: τ:

Follow Lemma 4.3.2 to get $P \approx P'$ and $P \approx P''$. The result then follows.

Case: ℓ:

Suppose that $P \xrightarrow{\ell}_s P'$ and $P \overset{\ell}{\Longrightarrow}_s P''$ implies $P \Longrightarrow_s P_1 \xrightarrow{\ell}_s P_2 \Longrightarrow_s P''$. From Lemma 4.3.2, we can conclude that $P \approx P_1$ and because of the bisimulation definition, we have $P' \approx P_2$ to complete we call upon Lemma 4.3.2 once more to get $P' \approx P''$ as required. □

A.4.5 Proof for Lemma 4.3.5

Proof. The proof considers a case analysis on the combination of ℓ_1, ℓ_2.

Case: $\ell_1 = s_1!\langle v_1 \rangle, \ell_2 = s_2?\langle v_2 \rangle$

$$
\begin{array}{lll}
P \mid s_1[\mathtt{o} : \vec{h}_1 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2] & \xrightarrow{\ell_1}_s & P_1 \mid s_1[\mathtt{o} : \vec{h}_1] \mid s_2[\mathtt{i} : \vec{h}_2] \\
& \xrightarrow{\ell_2}_s & P_1' \mid s_1[\mathtt{o} : \vec{h}_1'] \mid s_2[\mathtt{i} : \vec{h}_2' \cdot v_2] \\
P \mid s_1[\mathtt{o} : \vec{h}_1 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2] & \Longrightarrow_s & P_0 \mid s_1[\mathtt{o} : \vec{h}_0 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_0'] \\
& \Longrightarrow_s & P_2 \mid s_1[\mathtt{o} : \vec{h}_2' \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2'' \cdot v_2] \\
& \xrightarrow{\ell_2}_s & P_2' \mid s_1[\mathtt{o} : \vec{h}_4] \mid s_2[\mathtt{i} : \vec{h}_4']
\end{array}
\qquad
\begin{array}{l}
\Longrightarrow_s \; P_1' \mid s_1[\mathtt{o} : \vec{h}_1'] \mid s_2[\mathtt{i} : \vec{h}_2'] \\
\Longrightarrow_s \; P' \mid s_1[\mathtt{o} : \vec{h}_1''] \mid s_2[\mathtt{i} : \vec{h}_2''] \\
\xrightarrow{\ell_2}_s \; P_0' \mid s_1[\mathtt{o} : \vec{h}_0 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_0' \cdot v_2] \\
\Longrightarrow_s \; P_2' \mid s_1[\mathtt{o} : \vec{h}_3 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_3'] \\
\Longrightarrow_s \; P'' \mid s_1[\mathtt{o} : \vec{h}'] \mid s_2[\mathtt{i} : \vec{h}'']
\end{array}
$$

By using Lemma 3.3.1, we have that $P \mid s_1[\mathtt{o} : \vec{h}_1 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2] \Longrightarrow_s \xrightarrow{\ell_1}_s \xrightarrow{\ell_2}_s \Longrightarrow_s P' \mid s_1[\mathtt{o} : \vec{h}_1''] \mid s_2[\mathtt{i} : \vec{h}_2'']$ and $P \mid s_1[\mathtt{o} : \vec{h}_1 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2] \xrightarrow{\ell_2}_s \Longrightarrow_s \xrightarrow{\ell_1}_s P'' \mid s_1[\mathtt{o} : \vec{h}'] \mid s_2[\mathtt{i} : \vec{h}'']$. We use Lemmas 4.3.3 and 3.3.1 to get $P \mid s_1[\mathtt{o} : \vec{h}_1 \cdot v_1] \mid s_2[\mathtt{i} : \vec{h}_2] \xrightarrow{\ell_2}_s \Longrightarrow_s \xrightarrow{\ell_1}_s P' \mid s_1[\mathtt{o} : \vec{h}_1''] \mid s_2[\mathtt{i} : \vec{h}_2'']$.

The rest of the proof is similar to Lemma 4.3.2.

□

Appendix B

Appendix for the Applications of the ESP

B.1 Comparison with Asynchronous/Synchronous Calculi

B.1.1 Proofs for Section 5.2

We prove the results in Section 5.2 for the two asynchronous session typed π-calculi, by either giving the bisimulation closures when a bisimulation holds or giving the counterexample when bisimulation does not hold. The results for the synchronous and asynchronous π-calculi are well-known, hence we omit.

1. **Case:** $s!\langle v\rangle;s!\langle w\rangle;P \mid s[\mathsf{o} : \varepsilon] \not\approx s!\langle w\rangle;s!\langle v\rangle;P \mid s[\mathsf{o} : \varepsilon]$
   On the left hand side process we can observe a τ transition and get $s!\langle w\rangle;P \mid s[\mathsf{o} : v] \xrightarrow{s!\langle v\rangle} s!\langle w\rangle;P \mid s[\mathsf{o} : \varepsilon]$ but $s!\langle w\rangle;s!\langle v\rangle;P \mid s[\mathsf{o} : \varepsilon] \not\stackrel{s!\langle v\rangle}{\Longrightarrow}$ as required.

2. **Case:** $s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\mathsf{o} : \varepsilon] \mid s_2[\mathsf{o} : \varepsilon] \approx s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\mathsf{o} : \varepsilon] \mid s_2[\mathsf{o} : \varepsilon]$

Relation:

$$\begin{aligned}\mathscr{R} = \{\ & (s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon], s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]),\\ & (s_2!\langle w\rangle;P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon], P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w], s_1!\langle v\rangle;P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w], P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]),\\ & (s_2!\langle w\rangle;P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon], P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon], P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]),\\ & (P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w], P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon], P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon])\}\end{aligned}$$

gives the result.

3. **Case:** $s?(x);s?(y);P \mid s[\mathtt{i}:\varepsilon] \not\approx s?(y);s?(x);P \mid s[\mathtt{i}:\varepsilon]$

   On both processes we can observe a $s?\langle v\rangle$ transition and get $s?(x);s?(y);P \mid s[\mathtt{i}:v] \xrightarrow{\tau} s?(y);P\{v/x\} \mid s[\mathtt{i}:\varepsilon]$ and $s?(w);s?(v);P \mid s[\mathtt{i}:v] \xrightarrow{\tau} s?(x);P\{v/y\} \mid s[\mathtt{i}:\varepsilon]$. From the substitution, we have that both processes are not bisimilar.

4. **Case:** $s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon] \approx s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]$

Relation

$$\begin{aligned}\mathscr{R} = \{\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]),\\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]),\\ & (s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w], s_1?(x);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w], P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon], s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w], s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]),\\ & (s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w], s_1?(x);P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon], P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon])\}\end{aligned}$$

gives the result.

B.2 Selector Properties

B.2.1 Proof for Proposition 5.3.1 (1)

Proof. We type left and right hand side of the selectors mapping.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash P \triangleright \Delta \cdot x_r : S \cdot x_{\overline{r}} : \overline{S}}{\Gamma \vdash b(x_r).P \triangleright \Delta \cdot x_{\overline{r}} : \overline{S}}}{\Gamma \vdash \overline{b}(x_{\overline{r}}).b(x_r).P \triangleright \Delta}}{\Gamma \vdash \overline{b}(x_{\overline{r}}).b(x_r).P \mid b[\varepsilon] \triangleright \Delta \cdot b}}{\Gamma \vdash (\nu\ b)(\overline{b}(x_{\overline{r}}).b(x_r).P \mid b[\varepsilon]) \triangleright \Delta}$$

The above result agrees with the typing rule [Selector].

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot s : S \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}{\Gamma \vdash \overline{r}!\langle s\rangle;P \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}$$

The above result conisides with the typing rule [Reg].

$$\frac{\dfrac{\dfrac{\Gamma \vdash P \triangleright \Delta \cdot s : S \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}{\Gamma \vdash \texttt{if arrive } x \texttt{ then } P \texttt{ else } \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}}{\Gamma \vdash r?(x);\texttt{if arrive } x \texttt{ then } P \texttt{ else } \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}}{\Gamma \vdash \mu Select.r?(x);\texttt{if arrive } x \texttt{ then } P \texttt{ else } \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}$$

The above result conisides with the typing rule [Select].

□

B.2.2 Selector Properties

For the following proofs, we let $B_i = s_i[\texttt{i} : h_i, \texttt{o} : h_i']$.

Definition B.2.1. $s[\texttt{i} : \vec{h_i'} \cdot \vec{h_i}, \texttt{o} : \vec{h_o}] \succ s[\texttt{i} : \vec{h_i}, \texttt{o} : \vec{h_o'} \cdot \vec{h_o}]$ where $\exists P \cdot \Gamma \vdash P \mid B_i \triangleright \Delta \Longrightarrow \Gamma \vdash Q \mid B_j \triangleright \Delta$

Lemma B.2.1. Let $B_i \succ B_j$ and assume ℓ is a visible action. Then $\Gamma \vdash P \mid B_i \triangleright \Delta \xrightarrow{\ell} P' \mid B_i' \triangleright \Delta'$ iff $\Gamma \vdash P \mid B_j' \triangleright \Delta \xrightarrow{\ell} P' \mid B_j' \triangleright \Delta'$.

Proof. The lemma is proved by the definitions of the label transition system and environment transition. □

Definition B.2.2.

$$\begin{array}{rcl} \mathsf{IfSel}_i^n = \texttt{def} \quad X_1 & = & \texttt{if arrive } s_1 \texttt{ then } C_1[X_2] \texttt{ else } X_2 \\ & \vdots & \\ X_n & = & \texttt{if arrive } s_n \texttt{ then } C_n[X_1] \texttt{ else } X_1 \quad \texttt{in } X_i \end{array}$$

with $C_i = \texttt{typecase}\ s_i\ \texttt{of}\ \{(x_i : S_i) : R_{ij}; -\}_{1\leq i\leq n, 1\leq j\leq m}$ where $R_{ij}\{s_i/x_i\}$ is a blocking prefixed sequential series of actions with no blocking terms other than its prefix. Furthermore $R_{ij}\{s_i/x_i\}$ is session determinate.

The next definition is used in the proofs.

Definition B.2.3. We define $\mathsf{IfSel}_i^{'n} \mid \prod_{1\leq i\leq n} B'_i$ and $\mathsf{Sel}_i^{'n} \mid \prod_{1\leq i\leq n} B'_i$ as

1. $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \Longrightarrow \mathsf{IfSel}_i^{'n} \mid \prod_{1\leq i\leq n} B'_i \triangleright \Delta' \Longrightarrow \mathsf{IfSel}_{i+1}^n \mid \prod_{1\leq i\leq n} B''_i \triangleright \Delta''$
2. $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \Longrightarrow \mathsf{Sel}_i^{'n} \mid \prod_{1\leq i\leq n} B'_i \triangleright \Delta' \Longrightarrow \mathsf{Sel}_{i+1}^n \mid \prod_{1\leq i\leq n} B''_i \triangleright \Delta''$.

Lemma B.2.2. $\mathsf{IfSel}_i^n \mid \prod_{1\leq i\leq n} B_i \ \approx\ \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i$

Proof. By unfolding Sel_i^n n times we can see the bisimulation relation between the two processes. Consider relation $\mathscr{R}$, such that:

$$\begin{aligned}\mathscr{R} = \{(P,Q) \quad \mid \quad & P = \mathsf{IfSel}_i^n \mid \textstyle\prod_{1\leq i\leq n} B_i,\ Q = \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i \\ & P = \mathsf{IfSel}_i^{'n} \mid \textstyle\prod_{1\leq i\leq n} B'_i,\ Q = \mathsf{Sel}_{i+1}^n \mid \prod_{1\leq i\leq n} B_i, \\ & P = \mathsf{IfSel}_{i+1}^n \mid \textstyle\prod_{1\leq i\leq n} B_i,\ Q = \mathsf{Sel}_i^{'n} \prod_{1\leq i\leq n} B'_i\}\end{aligned}$$

where $B_i \succ B'_i$. For visible actions, $\ell \neq \tau$ we can use part 1 of Lemma B.2.1 to obtain $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta_{if} \xrightarrow{\ell} \mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B'_j \mid \ldots \mid B_n \triangleright \Delta'_{if}$ if and only if $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta_{sel} \xrightarrow{\ell} \mathsf{Sel}_i^n \mid B_1 \mid \ldots \mid B'_j \mid \ldots \mid B_n \triangleright \Delta'_{sel}$ and the resulting pair of procesess to be in $\mathscr{R}$ as required.

The result is the same for the other two defining pairs of $\mathscr{R}$.

For $\ell = \tau$ we obtain if $\Gamma \vdash \mathsf{IfSel}_i^{'n} \mid \prod_{1\leq i\leq n} B'_i \triangleright \Delta_{if} \xrightarrow{\tau} \mathsf{IfSel}_i^{''n} \mid \prod_{1\leq i\leq n} B''_i \triangleright \Delta'_{if}$ then $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta_{sel} \Longrightarrow \mathsf{Sel}_{i+1}^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta_{sel}$ and the resulting pair of procesess to be in $\mathscr{R}$ as required.

For the symmetric direction, we obtain if $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta_{sel} \xrightarrow{\tau} \mathsf{Sel}_i^{'n} \mid \prod_{1\leq i\leq n} B_i' \triangleright \Delta_{sel}$ then $\Gamma \vdash \mathsf{IfSel}_i' \prod_{1\leq i\leq n} B_i' \triangleright \Delta_{if} \stackrel{\tau}{\Longrightarrow} \mathsf{IfSel}_{i+1}^n \mid \prod_{1\leq i\leq n} B_i'' \triangleright \Delta_{if}'$ and the resulting pair of procesess to be in $\mathscr{R}$ as required.

The result is the same for the other two defining pairs of $\mathscr{R}$. □

The selectors enjoy the confluence property.

Lemma B.2.3. 1. $\mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B_n$ is confluent.

2. $\mathsf{Sel}_i^n \mid B_1 \mid \ldots \mid B_n$ is confluent.

Proof. We prove the first part. The second part is a direct consequence from Lemma B.2.2 and the fact that bisimulation preserves confluence.

We apply the confluence definition on $\mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B_n$ on all possible pairs of ℓ_1 and ℓ_2. Then we have: $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \xrightarrow{\ell_1} \stackrel{\widehat{\ell_2 \mid \ell_1}}{\Longrightarrow} \mathsf{IfSel}_j^{'n} \mid \prod_{1\leq i\leq n} B_i' \triangleright \Delta'$ and $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \stackrel{\ell_2}{\Longrightarrow} \stackrel{\widehat{\ell_1 \mid \ell_2}}{\Longrightarrow} \mathsf{IfSel}_k^{'n} \mid \prod_{1\leq i\leq n} B_i'' \triangleright \Delta''$.
Hence we need to show that $\mathsf{IfSel}_j^{'n} \mid \prod_{1\leq i\leq n} B_i' \approx \mathsf{IfSel}_k^{'n} \mid \prod_{1\leq i\leq n} B_i''$.

Consider relation $\mathscr{R} = \mathscr{S} \cup \{(\mathsf{IfSel}_j^{'n} \mid \prod_{1\leq i\leq n} B_i', \mathsf{IfSel}_k^{'n} \mid \prod_{1\leq i\leq n} B_i'')\}$, where

$$\mathscr{S} \quad = \quad \{(P,Q),(Q,P) \;\mid\; P = \mathsf{IfSel}_i^{'n} \mid \textstyle\prod_{1\leq i\leq n} B_i', Q = \mathsf{IfSel}_1^n \mid \prod_{1\leq i\leq n} B_i,\ B_i \succ B_i'\}$$

If $\Gamma \vdash \mathsf{IfSel}_i^{'n} \mid \prod_{1\leq i\leq n} B_i' \triangleright \Delta' \xrightarrow{\ell} \mathsf{IfSel}_i^{''n} \mid \prod_{1\leq i\leq n} B_i'' \triangleright \Delta''$ then $\Gamma \vdash \mathsf{IfSel}_1^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \stackrel{\hat{\ell}}{\Longrightarrow} \mathsf{IfSel}_1^n \mid \prod_{1\leq i\leq n} B_i''' \triangleright \Delta'''$ and the resulting process are related by $\mathscr{S}$.

For the symmetric case, $\Gamma \vdash \mathsf{IfSel}_1^n \mid \prod_{1\leq i\leq n} B_i \triangleright \Delta \xrightarrow{\ell} \mathsf{IfSel}_1^{'n} \mid \prod_{1\leq i\leq n} B_i''' \triangleright \Delta'''$ then $\Gamma \vdash \mathsf{IfSel}_i^{'n} \mid \prod_{1\leq i\leq n} B_i' \triangleright \Delta' \stackrel{\ell}{\Longrightarrow} \mathsf{IfSel}_1^n \mid \prod_{1\leq i\leq n} B_i'' \triangleright \Delta''$ and the resulting process are related by $\mathscr{S}$.

□

B.2.3 Proof of Lemma 5.4.1

Proof. By Lemma B.2.2, consider the equivalences,

$\mathsf{IfSel}^n_k \mid \prod_{1\le i\le n} B_i \approx \mathsf{Sel}^n_k \mid \prod_{1\le i\le n} B_i$ and $\mathsf{PermIfSel}^n_k \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermSel}^n_k \mid \prod_{1\le i\le n} B_i$.

We will show that $\mathsf{IfSel}^n_k \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermIfSel}^n_k \mid \prod_{1\le i\le n} B_i$, by exploiting Lemma B.2.3 to build a confluent up-to relation.

Consider the relation $\mathscr{R} = \mathscr{S} \cup \{(\mathsf{IfSel}^n_k \mid \prod_{1\le i\le n} B_i, \mathsf{PermIfSel}^n_k \mid \prod_{1\le i\le n} B_i)\}$ such that $\mathscr{S} = \{(\mathsf{IfSel}^n_1 \mid \prod_{1\le i\le n} B_i, \mathsf{PermIfSel}^n_1 \mid \prod_{1\le i\le n} B_i)\}$.

If $\Gamma \vdash \mathsf{IfSel}^n_1 \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xrightarrow{\ell} \Longrightarrow \mathsf{IfSel}^n_1 \mid \prod_{1\le i\le n} B'_i \triangleright \Delta'$ then $\Gamma \vdash \mathsf{PermIfSel}^n_1 \mid \prod_{1\le i\le n} B_i \triangleright \Delta \overset{\ell}{\Longrightarrow} \mathsf{IfSel}^n_1 \mid \prod_{1\le i\le n} B'_i \triangleright \Delta'$ and both resulting processes are in $\mathscr{S}$.

The symmetric case is similar. Then the proof is complete with Lemma B.2.2. □

B.2.4 Proof of Lemma 5.4.2

First by the similar technique as the static selector, we prove:

Lemma B.2.4. $\mathsf{DSel}^n_i \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i$ is confluent.

Then the rest is proved by constructing the up to relation of

$$\mathscr{R} = \mathscr{S} \cup \{\mathsf{DSel}^n_k \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i,\ \mathsf{PermDSel}^n_k \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i\}$$

where $\mathscr{S} = \{(\mathsf{DSel}^n_1 \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i,\ \mathsf{PermDSel}^n_1 \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i\}$, using a similar confluence property as done in the proof of Lemma 5.4.1.

B.3 Thread Elimination Transform Properties

In this section of the Appendix we prove Theorem 5.5.1. We first use some auxiliary lemmas that i) establish equivalence replicated and recursive definitions, ii) prove the confluence of the Lauer-Needham transform $LN[\![*a(x).P \mid a[\varepsilon]]\!]$ and, iii) prove the selector permutation on the Lauer-Needham transform. We finally prove the main result.

We establish an equivalence result between recursive and the replicated processes.

Lemma B.3.1. $\texttt{def}\ X = C[X]\ \texttt{in}\ X \approx *(c.C[\overline{c}]) \mid \overline{c}$, where C does not contain X.

Proof. $*P$ is defined to be $\mu Y.(P \mid Y)$, so we rewrite $*c.C[\overline{c}]$ to $\mu Y.(c.C[\overline{c}] \mid Y)$. $\mu Y.P$ is defined as $\texttt{def}\ Y \stackrel{\texttt{def}}{=} P\ \texttt{in}\ Y$. So $\mu Y.(c.C[\overline{c}] \mid Y)$ can be written as $\texttt{def}\ Y \stackrel{\texttt{def}}{=} c.C[\overline{c}] \mid Y\ \texttt{in}\ Y$.

We can build a bisimulation relation on the transitions of context C.

$$\begin{aligned}\mathscr{R} \;=\; \{&(P,Q),(Q,P) \mid \\ &P = \texttt{def}\ X \stackrel{\texttt{def}}{=} C[X]\ \texttt{in}\ C'[X], Q = \texttt{def}\ Y \stackrel{\texttt{def}}{=} c.C[\overline{c}] \mid Y\ \texttt{in}\ C'[\overline{c}] \mid Y \\ &P = \texttt{def}\ X \stackrel{\texttt{def}}{=} C[X]\ \texttt{in}\ X, Q = \texttt{def}\ Y \stackrel{\texttt{def}}{=} c.C[\overline{c}] \mid Y\ \texttt{in}\ Y \mid \overline{c}\}\end{aligned}$$

If $\texttt{def}\ X = C[X]\ \texttt{in}\ C'[X] \stackrel{\ell}{\longrightarrow} \texttt{def}\ X = C[X]\ \texttt{in}\ C''[X]$ then $\texttt{def}\ Y = c.C[\overline{c}] \mid Y\ \texttt{in}\ C'[\overline{c}] \mid Y \stackrel{\hat{\ell}}{\longrightarrow} \texttt{def}\ Y = c.C[\overline{c}] \mid Y\ \texttt{in}\ C''[\overline{c}] \mid Y$.

For the second pair the transition is obvious. □

A usefull definition is that of the LN-transform in recursive programming style.

Definition B.3.1 (LN transform-recursive programming style)**.**

$$
\begin{aligned}
LNR[\![*a(x).P]\!] &\stackrel{\text{def}}{=} (\nu\, q)(\mathsf{Loop}\langle q\rangle \mid q\langle(\mathtt{sd}, a, \emptyset)\rangle) \\
\mathsf{Loop}\langle q\rangle &\stackrel{\text{def}}{=} \mathtt{select}\ (x_s, x_a, y)\ \mathtt{from}\ q\ \mathtt{in}\ \mathtt{if}\ x_a = a\ \mathtt{then}\ \mathtt{new}\ \mathtt{env}\ y\ \mathtt{in}\ \mathscr{B}[\![*a(x).P]\!]\ \mathtt{else} \\
&\quad\ \mathtt{typecase}\ x\ \mathtt{of}\ \{x_1 : S_1 : \mathscr{B}[\![P_1]\!], \ldots x_{n-m} : S_{n-m} : \mathscr{B}[\![P_{n-m}]\!]\} \\
\mathscr{B}[\![*a(x).P]\!] &\stackrel{\text{def}}{=} a(w).\mathtt{update}\ (y, w, w')\ \mathtt{in}\ \mathtt{register}\ (x_a, a, \emptyset)\ \mathtt{to}\ q\ \mathtt{in}\ [\![P, y]\!] \\
\mathscr{B}[\![x^{(i)}?(z:T); Q]\!] &\stackrel{\text{def}}{=} x'?(z'); \mathtt{update}\ (y, z, z')\ \mathtt{in}\ \mathtt{update}\ (y, x, x')\ \mathtt{in}\ [\![Q, y]\!] \\
\mathscr{B}[\![x^{(i)}\&\{l_j : Q_j\}_j]\!] &\stackrel{\text{def}}{=} x'\&\{l_j : \mathtt{update}\ (y, x, x')\ \mathtt{in}\ [\![Q_j, y]\!]\}_j \\
[\![Q, y]\!] &\stackrel{\text{def}}{=} \mathtt{let}\ x' = [\![x]\!]_y\ \mathtt{in}\ \mathtt{register}\ (x', \mathtt{shd}, y)\ \mathtt{to}\ q\ \mathtt{in}\ \mathsf{Loop}\langle q\rangle \quad (Q \text{ is blocking at } x^{(i)}) \\
[\![\mathbf{0}, y]\!] &\stackrel{\text{def}}{=} \mathsf{Loop}\langle q\rangle
\end{aligned}
$$

Lemma B.3.2. $LN[\![*(a(x).P) \mid a[\varepsilon]]\!] \approx LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$

Proof. The proof is an application of lemma B.3.1. Since definition $LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$ uses process variable using lemma B.3.1 we can substitute process variables with names c_i and their definition with $*c_i...$ to get $LN[\![*(a(x).P) \mid a[\varepsilon]]\!]$. □

The LN-transformed process is essentially a sequential process with session endpoints composed in parallel. Hence we can also establish:

Lemma B.3.3. $LN[\![*(a(x).P) \mid a[\varepsilon]]\!]$ is confluent.

Proof. We use lemma B.2.4 to show that $LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$ is confluent then by lemma B.3.2 and the fact that bisimulation preserves confluence, we get the required result. □

We can know study the behaviour of the LN-transform.

Lemma B.3.4 (Event Server Permutation)**.** Let

$$
\begin{aligned}
P_1 \;=\; & (\nu\, \vec{c}or)(\mathsf{Loop}\langle o, q\rangle \mid \mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle \mid q\langle \ldots, (s_i, a_i, y_i, c_i), (s_j, a_i, y_j, c_j), \ldots\rangle \mid \\
& a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m[\mathtt{i} : \vec{h_{im}}, \mathtt{o} : \vec{h_{om}}])
\end{aligned}
$$

and

$$P_2 = (\nu\ \vec{c}or)(\mathsf{Loop}\langle o,q\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid q\langle\ldots,(s_j,a_iy_j,c_j),(s_i,a_i,y_i,c_i),\ldots\rangle \mid a[\vec{s}] \mid \prod_{m\in I} s_m[\mathtt{i} : \vec{h_{im}}, \mathtt{o} : \vec{h_{om}}])$$

Then $P_1 \approx P_2$.

Proof. The first step is by Definition B.3.1. We then apply Lemmas 5.4.2 and B.3.2. □

We finaly prove our main theorem (Theorem 5.5.1).

Proof. Since both processes are confluent we can develop a confluent up-to relation along with lemma B.3.4 to prove bisimulation closure.

Let relation $\mathscr{R}$ such that

$$\begin{aligned}\mathscr{R} = \{(P_1,P_2),(P_2,P_1) \mid\ & P_1 = *a(x).P \mid \textstyle\prod_{1\le i\le n} R_i \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}] \\ & R_1,\ldots,R_n \text{ blocking subterms of } P \\ & P_2 = \mathsf{Loop}\langle o,q\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid \\ & r\langle s_j,\ldots,s_{j-1}\rangle \mid \textstyle\prod_{1\le i\le n} B_i \mid a[\vec{s}]\}\end{aligned}$$

Then we prove that $\mathscr{R}$ is a bisimulation up-to confluence. For observable actions, the bisimulation holds trivially since if $P_1 \xrightarrow{\ell} P_1'$ then $P_2 \xrightarrow{\ell} P_2'$ and $P_1'\mathscr{R}P_2'$.

Let $P_2 \longrightarrow Q' \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid r\langle s_j,\ldots,s_{j-1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$ then $P_1 \Longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_j' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$, where $R_j \Longrightarrow R_j'$ and R_j' is a blocking server subterm of P and $Q' \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid r\langle s_j,\ldots,s_{j-1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}] \Longrightarrow \mathsf{Loop}\langle o,q\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid r\langle s_{j+1},\ldots,s_j\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$.

For the symmetric case, if $P_1 \longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_i' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$ then we choose a processe $P_2' \approx P_2$ (from Lemma 5.4.2) such that $P_2' = \mathsf{Loop}\langle o,q\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid r\langle s_i,s_j\ldots,s_{j+1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$.

Now we can observe $P_2' \Longrightarrow \mathsf{Loop}\langle o,q\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \mid r\langle s_j,\ldots,s_i\rangle \mid \prod_{1\leq i\leq n} B_i \mid a[\vec{s}]$

and

$*a(x).P \mid R_1 \mid \ldots \mid R_i' \mid \ldots \mid R_n \mid \prod_{1\leq i\leq n} B_i \mid a[\vec{s}] \Longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_i'' \mid \ldots \mid R_n \mid \prod_{1\leq i\leq n} B_i \mid a[\vec{s}]$

where R_i'' is a blocking subterm of P.

This completes the proof.

□

Appendix C

Apendix for the MSP

C.1 Global Types

C.1.1 Proof for Lemma 6.2.1

Proof. The proof uses induction on the syntax of the global type G. For $G = \mathtt{end}, G = t$ the proof is trivial.

For the induction step we do a case analysis on the definition of the syntax of G. Let $G = \mathtt{p} \to \mathtt{q} : \langle U \rangle . G'$. Then:

$$\begin{aligned} (\mathtt{p} \to \mathtt{q} : \langle U \rangle . G' \lceil \mathtt{p}) \lceil \mathtt{q} &= (G' \lceil \mathtt{p}) \lceil \mathtt{q} \\ (\mathtt{p} \to \mathtt{q} : \langle U \rangle . G' \lceil \mathtt{q}) \lceil \mathtt{p} &= (G' \lceil \mathtt{q}) \lceil \mathtt{p} \end{aligned}$$

From the induction hypothesis we know that

$$(G' \lceil \mathtt{p}) \lceil \mathtt{q} = (G' \lceil \mathtt{q}) \lceil \mathtt{p}$$

This is enough to complete the proof.

The rest of the cases are similar to the above. □

C.2 Subject Reduction

C.2.1 Proof for Theorem 6.2.1

Proof. We apply induction on the length of the reduction $\twoheadrightarrow$. Induction is done by a case analysis on the reduction rules. We present some cases, since the methodology is similar for the rest.

Case: [Link]

Let

$$P = a[\mathtt{p}](x_1).P_1 \mid \ldots \mid \overline{a}[n](x).P_n$$

We apply the typing rules [Accept], [Request], [Conc] to get $\Gamma \vdash P \triangleright \Delta$ with $\mathtt{co}(\Delta)$. Assume that

$$P \longrightarrow P' = (\nu\ s)(P_1\{s[1]/x_1\} \mid \ldots \mid P_n\{s[n]/x_n\})$$

From rule [Conc] we get that

$$\Gamma \vdash (\nu\ s)(P_1\{s[1]/x_1\} \mid \ldots \mid P_n\{s[n]/x_n\}) \triangleright \Delta \cdot s[1] : T_1 \ldots s[n] : T_n$$

We apply rule [SRes] to get $\Gamma \vdash P' \triangleright \Delta$ as required.

Case: [Comm]

Let

$$P = s[\mathtt{p}][\mathtt{q}]!\langle v\rangle; P_1 \mid s[\mathtt{q}][\mathtt{p}]?(x); P_2$$

We apply typing rules [Send], [Rcv], [Conc] to get that $\Gamma \vdash P \triangleright \Delta$ with

$$\Delta = \Delta_1 \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}} \cdot s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}$$

and $\mathtt{co}(\Delta)$. From the coherency of Δ we get that $\mathtt{co}(\Delta_1)$ and $T_{\mathtt{p}}\lceil \mathtt{q} = \overline{T_{\mathtt{q}}\lceil \mathtt{p}}$.

Let $P \longrightarrow P' = P_1 \mid P_2\{v/x\}$ and $\Gamma \vdash P' \triangleright \Delta'$ with

$$\Delta' = \Delta_1 \cdot s[\mathtt{p}] : T_{\mathtt{p}} \cdot s[\mathtt{q}] : T_{\mathtt{q}}$$

From the coherency of Δ we get that Δ' is also coherent. □

C.2.2 Proof for Theorem 6.3.1

We use the next Lemma to prove Theorem 6.3.1:

Lemma C.2.1. If $\mathtt{co}(\Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle;T)$ and $s[\mathtt{q}] \notin \mathtt{dom}(\Delta)$ then $\mathtt{co}(\Delta \cdot s[\mathtt{p}] : T)$.

Proof. Let $\mathtt{q}' \neq \mathtt{q}$ and $s[\mathtt{q}'] : T_{\mathtt{q}'} \in \Delta$. Then $[\mathtt{q}]!\langle U\rangle;T\lceil \mathtt{q}' = T\lceil \mathtt{q}'$. Since $s[\mathtt{q}] \notin \mathtt{dom}(\Delta)$ then for all $s[\mathtt{q}'] \in \mathtt{dom}(\Delta)$ $[\mathtt{q}]!\langle U\rangle;T\lceil \mathtt{q}' = T\lceil \mathtt{q}'$, so $\mathtt{co}(\Delta \cdot s[\mathtt{p}] : T)$. □

Proof for Theorem 6.3.1:

Proof. For each of the asynchronous MSP calculi, we use induction on the length of $\twoheadrightarrow$.

Output Asynchronous MSP:

Case: [Link]

Let

$$P = a[\mathtt{p}](x_1).P_1 \mid \ldots \mid \overline{a}[n](x).P_n$$

We apply typing rules [Accept], [Request], [Conc] to get $\Gamma \vdash P \triangleright \Delta$ with $\mathtt{co}(*(\Delta))$.

Assume

$$P \longrightarrow P' = (\nu\ s)(P_1\{s[1]/x_1\} \mid \ldots \mid P_n\{s[n]/x_n\} \mid s[1][\mathtt{o} : \varepsilon] \mid \ldots \mid s[n][\mathtt{o} : \varepsilon])$$

We apply typing rules [Accept], [Request], (QEmp), [Conc] to get

$$\Gamma \vdash P' \triangleright \Delta \cdot s[1] : T_1 \ldots s[n] : T_n \cdot s^{\circ}[1] : \emptyset \cdot \ldots \cdot s^{\circ}[n] : \emptyset$$

We apply (SRes) to get $\Gamma \vdash P' \triangleright \Delta$ as required.

Case: [Send]

Let

$$P = s[\mathrm{p}][\mathrm{q}]!\langle v\rangle; P_1 \mid s[\mathrm{p}][\mathrm{o} : h]$$

We type to get $\Gamma \vdash P \triangleright \Delta$ with

$$\begin{aligned} \Delta &= \Delta_1 \cdot s[\mathrm{p}] : [\mathrm{q}]!\langle U\rangle; T \cdot s^{\circ}[\mathrm{p}] : M \\ *(\Delta) &= *(\Delta_1) \cup \{s[\mathrm{p}] : M * [\mathrm{q}]!\langle U\rangle; T\} \end{aligned}$$

Assume that

$$P \longrightarrow P' = P_1 \mid s[\mathrm{p}][\mathrm{o} : [\mathrm{q}](v) \cdot h]$$

with $\Gamma \vdash P' \triangleright \Delta'$ with

$$\begin{aligned} \Delta' &= \Delta_1 \cdot s[\mathrm{p}] : T \cdot s^{\circ}[\mathrm{p}] : [\mathrm{q}]!U; M \\ *(\Delta') &= *(\Delta_1) \cup \{s[\mathrm{p}] : M * [\mathrm{q}]!\langle U\rangle; T\} = *(\Delta) \end{aligned}$$

as required.

Input Asynchronous MSP:

Case: [Link]

Similar justification with **Case:** [Link] for Output Asynchronous MSP.

Case: [Send]

Let

$$P = s[\mathtt{p}][\mathtt{q}]!\langle v\rangle; P_1 \mid s[\mathtt{q}][\mathtt{i} : h] \mid P_2$$

We type to get $\Gamma \vdash P \triangleright \Delta$ with

$$\Delta = \Delta_1 \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T \cdot s^{\mathtt{i}}[\mathtt{q}] : M$$

and $\mathtt{co}(*(\Delta))$.

Assume

$$P \longrightarrow P' = P_1 \mid s[\mathtt{q}][\mathtt{i} : [\mathtt{q}](v) \cdot h]$$

with $\Gamma \vdash P' \triangleright \Delta'$ with

$$\Delta' = \Delta_1 \cdot s[\mathtt{p}] : T \cdot s^{\mathtt{i}}[\mathtt{q}] : [\mathtt{q}]!U; M$$

and $*(\Delta')$ coherent from lemma C.2.1.

I/O Asynchrony MSP:

Case: [Link]

Similar justification with **Case:** [Link] for Output Asynchronous MSP.

Case: [Send]

Similar argumentation with **Case:** [Send] for Output Asynchronous MSP.

Case: [Comm]

Let

$$P = s[\mathtt{p}][\mathtt{o} : \vec{h} \cdot h] \mid s[\mathtt{q}][\mathtt{i} : \vec{h'}]$$

We type to get $\Gamma \vdash P \triangleright \Delta$ with

$$\Delta = s^{\circ}[\mathtt{p}] : M; [\mathtt{q}]!U \cdot s^{\mathtt{i}}[\mathtt{q}] : M'$$

with trivialy $\mathtt{co}(*(\Delta))$.

Assume

$$P \longrightarrow P' = s[\mathtt{p}][\mathtt{o} : \vec{h}] \mid s[\mathtt{q}][\mathtt{i} : h \cdot \vec{h'}]$$

and $\Gamma \vdash P' \triangleright \Delta'$ with

$$\Delta = s^{\circ}[\mathtt{p}] : M \cdot s^{\mathtt{i}}[\mathtt{q}] : [\mathtt{p}]?U; M'$$

with trivialy $\mathtt{co}(*(\Delta'))$. □

C.3 Proofs for Bisimulation Properties

C.3.1 Parallel Observer Property

Lemma C.3.1. If $\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2$ and $E, \Gamma \vdash P_1 \mid P_2 \triangleright \Delta$ then

1. $\Delta = \Delta_1 \cup \Delta_2$, $\Delta_1 \cap \Delta_2 = \emptyset$
2. $E, \Gamma \vdash P_1 \triangleright \Delta_1$ and $E, \Gamma \vdash P_2 \triangleright \Delta_2$

Proof. Part 1 is obtain from typing rule [Conc]. Part 2 is immediate from part 1, since $\Delta \subseteq \Delta_1$ (resp. $\Delta \subseteq \Delta_2$). □

C.3.2 Proof for Lemma 6.5.1

Proof. We use a coinduction method which is implied by the bisimilarity definition.

Assume that for $\Gamma \vdash P_1 \triangleright \Delta_1 \approx P_2 \triangleright \Delta_2$, we have $\Delta_1 \rightleftharpoons \Delta_2$. Then by the definition of $\rightleftharpoons$, there exists Δ such that

$$\Delta_1 \twoheadrightarrow \Delta \text{ and } \Delta_2 \twoheadrightarrow \Delta \tag{C.1}$$

Now assume that $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} P_1' \triangleright \Delta_1'$ then, $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} P_2' \triangleright \Delta_2'$ and by the typed transition definition we get $(\Gamma, \Delta_1) \xrightarrow{\ell} (\Gamma, \Delta_1')$, $(\Gamma, \Delta_2) \stackrel{\ell}{\Longrightarrow} (\Gamma, \Delta_2')$. We need to show that $\Delta_1' \rightleftharpoons \Delta_2'$.

We prove by a case analysis on the transition $\xrightarrow{\ell}$ on (Γ, Δ_1) and (Γ, Δ_2).

- **Case $\ell = \tau$:** The proof is trivial.

  Case $\ell = a[\mathtt{p}](s)$ or $\ell = \overline{a}[\mathtt{p}](s)$: Then

  $$(\Gamma, \Delta_1) \xrightarrow{\ell} (\Gamma, \Delta_1 \cdot s[\mathtt{p}] : T_{\mathtt{p}} \cdot \ldots \cdot s[\mathtt{q}] : T_{\mathtt{q}})$$

  and

  $$(\Gamma, \Delta_2) \Longrightarrow \xrightarrow{\ell} \Longrightarrow (\Gamma, \Delta_2'' \cdot s[\mathtt{p}] : T_{\mathtt{p}} \cdot \ldots \cdot s[\mathtt{q}] : T_{\mathtt{q}})$$

  We set

  $$\Delta' = \Delta \cdot s[\mathtt{p}] : T_{\mathtt{p}} \cdot \ldots \cdot s[\mathtt{q}] : T_{\mathtt{q}}$$

  to obtain $\Delta_1' \twoheadrightarrow \Delta'$ and $\Delta_2' \twoheadrightarrow \Delta'$, by the coinduction hypothesis (C.1).

- **Case $\ell = s[\mathtt{p}][\mathtt{q}]!\langle v \rangle$:**

  For *synchronous and input asynchronous multiparty session π-calculus*, we know from the definition of environment transition, that $s[\mathtt{q}] \notin \mathtt{dom}(\Delta_1)$ and $s[\mathtt{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s[\mathtt{q}] \notin \mathtt{dom}(\Delta)$ for the synchronous case and $s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta_1)$ and $s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s^{\mathtt{i}}[\mathtt{q}] \notin \mathtt{dom}(\Delta)$ for the input asynchronous case. We set

  $$\Delta_1 = s[\mathtt{p}] : [\mathtt{q}]!\langle v \rangle; T \cdot \Delta_1''$$

  and

  $$\Delta_2 = s[\mathtt{p}] : [\mathtt{q}]!\langle v \rangle; T \cdot \Delta_2''$$

so

$$\Delta = s[\mathrm{p}] : [\mathrm{q}]!\langle v\rangle; T \cdot \Delta''$$

by (C.1). We set $\Delta' = s[\mathrm{p}] : T \cdot \Delta''$ to obtain $\Delta_1' \twoheadrightarrow \Delta'$ and $\Delta_2' \twoheadrightarrow \Delta'$.

For *output and input/output asynchronous multiparty session π-calculus*, we know from the definition of environment transition, that $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s[\mathrm{q}] \notin \mathtt{dom}(\Delta)$ for the output asynchrony case and $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta)$ for the input/output asynchronous case. Then

$$\Delta_1 = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!v \cdot \Delta_1''$$

and

$$\Delta_2 = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!v \cdot \Delta_2''$$

so

$$\Delta = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!v \cdot \Delta''$$

by (C.1). We set $\Delta' = s^{\circ}[\mathrm{p}] : M \cdot \Delta''$ to obtain $\Delta_1' \twoheadrightarrow \Delta'$ and $\Delta_2' \twoheadrightarrow \Delta'$.

- **Case** $\ell = s[\mathrm{p}][\mathrm{q}]!(s'[\mathrm{p}'])$**:**

For *synchronous and input asynchronous multiparty session π-calculus*, we know from the definition of environment transition, that for the synchronous case, $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s[\mathrm{q}] \notin \mathtt{dom}(\Delta)$. And for the input asynchronous case $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta)$. We set

$$\Delta_1 = s[\mathrm{p}] : [\mathrm{q}]!\langle T'\rangle; T \cdot \Delta_1''$$

and

$$\Delta_2 = s[\mathrm{p}] : [\mathrm{q}]!\langle T'\rangle; T \cdot \Delta_2''$$

so

$$\Delta = s[\mathrm{p}] : [\mathrm{q}]!\langle v\rangle; T \cdot \Delta''$$

by (C.1). We set $\Delta' = s[\mathrm{p}] : T \cdot \Delta'' \cdot \{s[\mathrm{p}_i] : T_i\}$ to obtain $\Delta_1' \twoheadrightarrow \Delta'$ and $\Delta_2' \twoheadrightarrow \Delta'$.

For *output and input/output asynchronous multiparty session* π*-calculus*, we know from the definition of environment transition, that $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s[\mathrm{q}] \notin \mathtt{dom}(\Delta)$ for the output asynchrony case and $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s^{\mathtt{i}}[\mathrm{q}] \notin \mathtt{dom}(\Delta)$ for the input/output asynchronous case. Then $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_1)$ and $s[\mathrm{q}] \notin \mathtt{dom}(\Delta_2)$, thus $s[\mathrm{q}] \notin \mathtt{dom}(\Delta)$ and

$$\Delta_1 = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!T' \cdot \Delta_1''$$

and

$$\Delta_2 = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!T' \cdot \Delta_2''$$

so

$$\Delta = s^{\circ}[\mathrm{p}] : M; [\mathrm{q}]!v \cdot \Delta''$$

by (C.1). We set $\Delta' = s^{\circ}[\mathrm{p}] : M \cdot \Delta'' \cdot \{s[\mathrm{p}_i] : T_i\}$ to obtain $\Delta_1' \twoheadrightarrow \Delta'$ and $\Delta_2' \twoheadrightarrow \Delta'$.

- The remaining cases on session channel actions are similar.

□

C.3.3 Configuration Transition Properties

Lemma C.3.2.

- If $E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E'$ then $\{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}}, s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}\} \subseteq \mathtt{proj}(E)$ and $\{s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}\} \subseteq \mathtt{proj}(E')$.

- If $E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:l} E'$ then $\{s[\mathtt{p}] : [\mathtt{q}] \oplus \{l_i : T_{i\mathtt{p}}\}, s[\mathtt{q}] : [\mathtt{p}]\&\{l_i : T_{i\mathtt{q}}\}\} \subseteq \mathtt{proj}(E)$ and $\{s[\mathtt{p}] : T_{k\mathtt{p}}, s[\mathtt{q}] : T_{k\mathtt{q}}\} \subseteq \mathtt{proj}(E')$

Proof. Part 1: We apply induction on the definition structure of $s : \mathtt{p} \to \mathtt{q} : U$. The base case

$$\{s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s : G\}$$

is easy since

$$\{s[\mathtt{p}] : (\mathtt{p} \to \mathtt{q} : \langle U\rangle.G)\lceil\mathtt{p}, s[\mathtt{q}] : (\mathtt{p} \to \mathtt{q} : \langle U\rangle.G)\lceil\mathtt{q}\} =$$
$$\{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}}, s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}\} \subseteq \mathtt{proj}(s : \mathtt{p} \to \mathtt{q} : \langle U\rangle.G)$$

and

$$\{s[\mathtt{p}] : G\lceil\mathtt{p}, s[\mathtt{q}] : G\lceil\mathtt{q}\} = \{s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}\} \subseteq \mathtt{proj}(s : G)$$

The main induction rule concludes that:

$$\{s : \mathtt{p}' \to \mathtt{q}' : \langle U\rangle.G\} \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} \{s : G'\}$$

if $\mathtt{p} \neq \mathtt{p}'$ and $\mathtt{q} \neq \mathtt{q}'$ and $\{s : G\} \overset{s:\mathtt{p}\to\mathtt{q}:U}{\longrightarrow} \{s : G'\}$. From the induction hypothesis we know that:

$$\begin{aligned} \{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}}, s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}\} &\subseteq \mathtt{proj}(s : G) \\ \{s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}\} &\subseteq \mathtt{proj}(s : G') \end{aligned}$$

to conclude that:

$$\begin{aligned} \{s[\mathtt{p}] : (\mathtt{p}' \to \mathtt{q}' : \langle U\rangle.G)\lceil\mathtt{p}, s[\mathtt{q}] : (\mathtt{p}' \to \mathtt{q}' : \langle U\rangle.G)\lceil\mathtt{q}\} &= \\ \{s[\mathtt{p}] : G\lceil\mathtt{p}, s[\mathtt{q}] : G\lceil\mathtt{q}\} &= \\ \{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}}, s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}}\} \subseteq \mathtt{proj}(s : G) & \end{aligned}$$

and

$$\begin{aligned} \{s[\mathtt{p}] : (\mathtt{p}' \to \mathtt{q}' : \langle U\rangle.G')\lceil\mathtt{p}, s[\mathtt{q}] : (\mathtt{p}' \to \mathtt{q}' : \langle U\rangle.G')\lceil\mathtt{q}\} &= \\ \{s[\mathtt{p}] : G'\lceil\mathtt{p}, s[\mathtt{q}] : G'\lceil\mathtt{q}\} &= \\ \{s[\mathtt{p}] : T_{\mathtt{p}}, s[\mathtt{q}] : T_{\mathtt{q}}\} \subseteq \mathtt{proj}(s : G) & \end{aligned}$$

as required.

Part 2: Similar. □

Proof for Proposition 6.4.1

Proof. **(1)** We apply induction on the definition structure of $\overset{\ell}{\longrightarrow}$.

Basic Step:

Case: $\ell = \overline{a}[s](A)$.

From rule [Acc] we get

$$(E_1, \Gamma_1, \Delta_1) \xrightarrow{\ell} (E_1 \cdot s : G, \Gamma_1, \Delta_1 \cdot \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\}_{\mathtt{p}_i \in A})$$

From the environment configuration definition we get that

$$\exists E_1' \cdot E_1 \longrightarrow^* E_1', \mathtt{proj}(E_1') \supseteq *(\Delta_1)$$

We also get that $\mathtt{proj}(s : G) \supseteq \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\}_{i \in A}$. So we can safely conclude that

$$E_1 \cdot s : G \longrightarrow^* E_1' \cdot s : G, \mathtt{proj}(E_1 \cdot s : G) \supseteq \Delta_1 \cdot \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\}_{\mathtt{p}_i \in A}$$

Case: $\ell = \overline{a}[s](A)$. Similar as above.

Case: $\ell = s[\mathtt{p}][\mathtt{q}]!\langle v \rangle$.

From rule [Out] we get

$$(E_1, \Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T) \xrightarrow{\ell} (E_2, \Gamma, \Delta \cdot s[\mathtt{p}] : T) \tag{C.2}$$

$$\mathtt{proj}(E_1) \supseteq \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T \tag{C.3}$$

$$E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E_2 \tag{C.4}$$

From C.3 we get $\mathtt{proj}(E_1) \supseteq \Delta \cdot \{s[\mathtt{p}] : [\mathtt{q}]!\langle U \rangle; T \cdot s[\mathtt{q}] : [\mathtt{p}]?(U); T'\}$ and from C.4 and lemma C.3.2 we get that $\mathtt{proj}(E_2) \supseteq \Delta \cdot \{s[\mathtt{p}] : T \cdot s[\mathtt{q}] : T'\}$. The result is then implied.

Case: $\ell = s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])$.

$$(E_1, \Gamma, \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T\mathtt{p}'\rangle; T) \xrightarrow{\ell} (E_2 \cdot s : G, \Gamma, \Delta \cdot s[\mathtt{p}] : T \cdot \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\}) \quad \text{(C.5)}$$

$$\mathtt{proj}(E_1) \supseteq \Delta \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T'_{\mathtt{p}}\rangle; T \quad \text{(C.6)}$$

$$E_1 \xrightarrow{s:\mathtt{p}\to\mathtt{q}:T'_{\mathtt{p}}} E_2 \quad \text{(C.7)}$$

$$\mathtt{proj}(s : G) \supseteq \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\} \quad \text{(C.8)}$$

From C.6 we get $\mathtt{proj}(E_1) \supseteq \Delta \cdot \{s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T \cdot s[\mathtt{q}] : [\mathtt{p}]?(U); T'\}$ and from C.7 and lemma C.3.2 we get that $\mathtt{proj}(E_2) \supseteq \Delta \cdot \{s[\mathtt{p}] : T \cdot s[\mathtt{q}] : T'\} \supset \Delta \cdot s[\mathtt{p}] : T$. From C.8 we get that $\mathtt{proj}(E_2 \cdot s : G) \supseteq \Delta \cdot s[\mathtt{p}] : T \cdot \{s[\mathtt{p}_i] : s\lceil \mathtt{p}_i\}$ as required.

The rest of the base cases are similar.

Inductive Step:

The inductive rule for environment configuration is [Inv]. $(E_1, \Gamma_1, \Delta_1) \xrightarrow{\ell} (E_2, \Gamma_2, \Delta_2)$. From rule [Inv] we get

$$E_1 \longrightarrow^* E'_1 \quad \text{(C.9)}$$

$$(E'_1, \Gamma_1, \Delta_1) \xrightarrow{\ell} (E'_2, \Gamma_2, \Delta_2) \quad \text{(C.10)}$$

$$E_2 \longrightarrow^* E'_2 \quad \text{(C.11)}$$

From the inductive hypothesis we know that for C.10 $\exists E_3 \cdot E'_2 \longrightarrow^* E_3$. By C.11 we get that $E_2 \longrightarrow^* E'_2 \longrightarrow^* E_3$ as required. □

Lemma C.3.3.

1. If $(E, \Gamma, \Delta_1) \xrightarrow{\ell} (E', \Gamma', \Delta_2)$ then $(\Gamma, \Delta_1) \xrightarrow{\ell} (\Gamma', \Delta_2)$
2. If $(E, \Gamma, \Delta_1) \xrightarrow{\ell} (E', \Gamma', \Delta'_1)$ and $\Delta_1 \rightleftharpoons \Delta_2$ then $(E, \Gamma, \Delta_2) \xRightarrow{\ell} (E', \Gamma', \Delta'_2)$

3. If $(\Gamma,\Delta_1) \xrightarrow{\ell} (\Gamma',\Delta_2)$ then there exists E such that $(E,\Gamma,\Delta) \xrightarrow{\ell} (E',\Gamma',\Delta_2)$
4. If $(E,\Gamma,\Delta\cdot s[\mathtt{p}] : T_{\mathtt{p}}) \xrightarrow{\ell} (E',\Gamma,\Delta'\cdot s[\mathtt{p}] : T_{\mathtt{p}})$ then $(E,\Gamma,\Delta) \xrightarrow{\ell} (E',\Gamma,\Delta')$
5. If $(E,\Gamma,\Delta_1) \xrightarrow{\ell} (E',\Gamma,\Delta_2)$ then $(E,\Gamma,\Delta_1\cdot\Delta) \xrightarrow{\ell} (E',\Gamma,\Delta_2\cdot\Delta)$
   provided that if $(E,\Gamma,\Delta) \xrightarrow{\ell'} (E,\Gamma,\Delta')$ then $\ell \not\asymp \ell'$

Proof. Part 1:

The proof for part 1 is easy to be implied by a case analysis on the configuration transition definition with respect to environment transition definition.

Part 2:

By the case analysis on ℓ.

Case $\ell = \tau$**:** The result is trivial.

Case $\ell = \overline{a}[\mathtt{p}](s)$ **or** $\ell = a[\mathtt{p}](s)$**:** The result comes from a simple transition.

Case $\ell = s[\mathtt{p}][\mathtt{q}]!\langle v\rangle$**:** $\Delta_1 \rightleftharpoons \Delta_2$ implies $\Delta_1 \twoheadrightarrow \Delta$ and $\Delta_2 \twoheadrightarrow \Delta$ for some Δ and $\Delta = \Delta'\cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle;T$ for synchronous and input asynchronous MSP and $\Delta = \Delta'\cdot s[\mathtt{p}] : M;[\mathtt{q}]!U.$ for output and input/output asynchronous MSP.

$(E,\Gamma,\Delta_2) \Longrightarrow (E,\Gamma,\Delta) \xrightarrow{\ell}$ as required.

Case $\ell = s[\mathtt{p}][\mathtt{q}]!(s'[\mathtt{p}'])$**:** $\Delta_1 \rightleftharpoons \Delta_2$ implies $\Delta_1 \twoheadrightarrow \Delta$ and $\Delta_2 \twoheadrightarrow \Delta$ for some Δ and $\Delta = \Delta'\cdot s[\mathtt{p}] : [\mathtt{q}]!\langle T'\rangle;T$ for synchronous and input asynchronous MSP and $\Delta = \Delta'\cdot s[\mathtt{p}] : M;[\mathtt{q}]!T'.$ for output and input/output asynchronous MSP.

$(E,\Gamma,\Delta_2) \Longrightarrow (E,\Gamma,\Delta) \xrightarrow{\ell}$ as required.

The remaining cases are similar.

Part 3:

We do a case analysis on ℓ.

Cases $\ell = \tau, \ell = \overline{a}[\mathtt{p}](s), \ell = a[\mathtt{p}](s)$**:** The result holds for any E.

Case $\ell = s[\mathtt{p}][\mathtt{q}]!\langle v\rangle : \Delta_1 = \Delta_1' \cdot \Delta_1''$ with $\Delta_1'' = s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T_{\mathtt{p}} \cdot \ldots \cdot s[\mathtt{r}] : T_{\mathtt{r}}$ for synchronous and input asynchronous MSP. Choose $E = E' \cdot s : G$ with $*(\Delta_1'') \subseteq \mathtt{proj}(s : G)$ and $s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}} \in \mathtt{proj}(s : G)$ and $*(\Delta)_1 \subseteq \mathtt{proj}(E)$ By the definition of configuration transition relation, we obtain $(E, \Gamma, \Delta) \xrightarrow{\ell} (E, \Gamma', \Delta_2)$, as required.

$\Delta_1 = \Delta_1' \cdot \Delta_1''$ with $\Delta_1'' = s[\mathtt{p}] : T_{\mathtt{p}} \cdot s^{\circ}[\mathtt{p}] : M; [\mathtt{q}]!U \cdot \ldots \cdot s[\mathtt{r}] : T_{\mathtt{r}}$ for output and input/output asynchronous MSP. Choose $E = E' \cdot s : G$ with $*(De_1'') \subseteq \mathtt{proj}(s : G)$ and $s[\mathtt{q}] : [\mathtt{p}]?(U); T_{\mathtt{q}} \in \mathtt{proj}(s : G)$ and $*(\Delta)_1 \subseteq \mathtt{proj}(E)$ By the definition of configuration transition relation, we obtain $(E, \Gamma, \Delta) \xrightarrow{\ell} (E, \Gamma', \Delta_2)$, as required.

Remaining cases are similar.

Part 4:

$(E, \Gamma, \Delta \cdot s[\mathtt{p}] : T_{\mathtt{p}}) \xrightarrow{\ell} (E', \Gamma, \Delta' \cdot s[\mathtt{p}] : T_{\mathtt{p}})$ implies that $s[\mathtt{p}] \notin \mathtt{subj}(\ell)$. The result then follows from the definition of configuration transition.

Part 5:

Case $\ell = \tau, \ell = \overline{a}[\mathtt{p}](s), \ell = a[\mathtt{p}](s)$**:** The result holds by definition of the configuration transition.

Case $\ell = s[\mathtt{p}][\mathtt{q}]!\langle U\rangle$**:** For synchronous and input asynchronous MSP we have that $\Delta_1 = \Delta_1' \cdot s[\mathtt{p}] : [\mathtt{q}]!\langle U\rangle; T$ and $E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E'$. For synchronous MSP assume $s[\mathtt{q}] \in \Delta$, then by definition of weak configuration pair we have $\Delta = \Delta'' \cdot s[\mathtt{q}] : \mathtt{q}[U][T]?()$; and $(E, \Gamma, \Delta) \xrightarrow{s[\mathtt{q}][\mathtt{p}]?\langle U\rangle}$. But this contradicts with the assumption $\ell \not\asymp \ell'$, so $s[\mathtt{q}] \notin \Delta$. By the definition of configuration pair transition we get that $(E, \Gamma, \Delta_1 \cdot \Delta) \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle U\rangle} (E, \Gamma, \Delta_2 \cdot \Delta)$. For input asynchronous MSP assume that $s^{\mathtt{i}}[\mathtt{q}] \in \Delta$, then by definition of weak configuration pair we have $\Delta = \Delta'' \cdot s^{\mathtt{i}}[\mathtt{q}] : M$

and $(E,\Gamma,\Delta) \xrightarrow{s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle U\rangle}$. But this contradicts with the assumption $\ell \not\asymp \ell'$, so $s^{\mathtt{i}}[\mathtt{q}] \notin \Delta$. By the definition of configuration pair transition we get the result.

For output and input/output asynchronous MSP we have $\Delta_1 = \Delta_1' \cdot s^{\mathtt{o}}[\mathtt{p}] : M; [\mathtt{q}]!\langle U\rangle$; and $E \xrightarrow{s:\mathtt{p}\to\mathtt{q}:U} E'$. For output asynchronous MSP assume $s[\mathtt{q}] \in \Delta$, then we would have $(E,\Gamma,\Delta) \xrightarrow{s[\mathtt{q}][\mathtt{p}]?\langle U\rangle}$ which contradicts with $\ell \not\asymp \ell'$ and $s[\mathtt{q}] \notin \Delta$ to get the required result by the configuration pair definition. For input/output asynchronous MSP assume $s^{\mathtt{i}}[\mathtt{q}] \in \Delta$, then we would have $(E,\Gamma,\Delta) \xrightarrow{s^{\mathtt{i}}[\mathtt{q}][\mathtt{p}]?\langle U\rangle}$ which contradicts with $\ell \not\asymp \ell'$ and $s^{\mathtt{i}}[\mathtt{q}] \notin \Delta$ to get the required result by the configuration pair definition.

Remaining cases are similar.

□

C.3.4 Proof for Lemma A.3.1

Proof. Since we are dealing with closed processes, the interesting case is parallel composition. We need to show that if $E,\Gamma \vdash P \triangleright \Delta_1 \approx_g Q \triangleright \Delta_2$ then for all R such that $E,\Gamma \vdash P \mid R \triangleright \Delta_3, E,\Gamma \vdash Q \mid R \triangleright \Delta_4$ then $E,\Gamma \vdash P \mid R \triangleright \Delta_3 \approx_g Q \mid R \triangleright \Delta_4$.

Let

$$\begin{aligned} S = \; & \{(E,\Gamma \vdash P \mid R \triangleright \Delta_3,\; E,\Gamma \vdash Q \mid R \triangleright \Delta_4) \;\mid \\ & E,\Gamma \vdash P \triangleright \Delta_1 \approx_g Q \triangleright \Delta_2, \\ & \forall R \cdot E,\Gamma \vdash P \mid R \triangleright \Delta_3, E,\Gamma \vdash Q \mid R \triangleright \Delta_4\} \end{aligned}$$

Before we proceed to a case analysis, we extract general results. Let $\Gamma \vdash P \triangleright \Delta_1, \Gamma \vdash Q \triangleright \Delta_2, \Gamma \vdash$

$R \triangleright \Delta_5, \Gamma \vdash P \mid R \triangleright \Delta_3, \Gamma \vdash Q \mid R \triangleright \Delta_4$ then from typing rule [Conc] we get

$$\Delta_3 = \Delta_1 \cup \Delta_5 \tag{C.12}$$

$$\Delta_4 = \Delta_2 \cup \Delta_5 \tag{C.13}$$

$$\Delta_1 \cap \Delta_5 = \emptyset \tag{C.14}$$

$$\Delta_2 \cap \Delta_5 = \emptyset \tag{C.15}$$

We prove that S is a bisimulation. There are three cases:

Case: $E, \Gamma \vdash P \mid R \triangleright \Delta_3 \xrightarrow{\ell} E', \Gamma \vdash P' \mid R \triangleright \Delta_3'$

From typed transition definition we have that:

$$P \mid R \xrightarrow{\ell} P' \mid R \tag{C.16}$$

$$(E, \Gamma, \Delta_3) \xrightarrow{\ell} (E', \Gamma, \Delta_3') \tag{C.17}$$

Transition (C.16) and rule $\langle \text{Par} \rangle$ (LTS for MSP calculi in Figure 6.9) imply:

$$P \xrightarrow{\ell} P' \tag{C.18}$$

From (C.12), transition (C.17) can be written as $(E, \Gamma, \Delta_1 \cup \Delta_5) \xrightarrow{\ell} (E', \Gamma, \Delta_1' \cup \Delta_5)$, to conclude from Lemma C.3.3 part 4, that:

$$(E, \Gamma, \Delta_1) \xrightarrow{\ell} (E', \Gamma, \Delta_1') \tag{C.19}$$

$$\mathtt{subj}(\ell) \notin \mathtt{dom}(\Delta_5) \tag{C.20}$$

Transitions C.18 and C.19 imply $E,\Gamma \vdash P \triangleright \Delta_1 \xrightarrow{\ell} E',\Gamma \vdash P' \triangleright \Delta_1'$. From the definition of set S we get that $E,\Gamma \vdash Q \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} E',\Gamma \vdash Q' \triangleright \Delta_2'$.

From the typed transition definition we have that:

$$Q \stackrel{\ell}{\Longrightarrow} Q' \tag{C.21}$$

$$(E,\Gamma,\Delta_2) \stackrel{\ell}{\Longrightarrow} (E',\Gamma,\Delta_2') \tag{C.22}$$

From C.20 and part 5 of Lemma C.3.3 we can write: $(E,\Gamma,\Delta_2 \cup \Delta_5) \stackrel{\ell}{\Longrightarrow} (E',\Gamma,\Delta_2' \cup \Delta_5)$, to imply from C.21 that $E,\Gamma \vdash P \mid R \triangleright \Delta_4 \stackrel{\ell}{\Longrightarrow} E',\Gamma \vdash P' \mid R \triangleright \Delta_4'$ as required.

Case: 2

$$E,\Gamma \vdash P \mid R \triangleright \Delta_3 \xrightarrow{\tau} E' \vdash P' \mid R' \triangleright \Delta_3'$$

From the typed transition definition we have that:

$$P \mid R \xrightarrow{\tau} P' \mid R' \tag{C.23}$$

$$(E,\Gamma,\Delta_3) \xrightarrow{\tau} (E,\Gamma,\Delta_3') \tag{C.24}$$

From C.23 and rule $\langle$Tau$\rangle$ we get

$$P \xrightarrow{\ell} P' \tag{C.25}$$

$$R \xrightarrow{\ell'} R' \tag{C.26}$$

From C.12 transition C.24 can be written $(E,\Gamma,\Delta_1 \cup \Delta_5) \xrightarrow{\tau} (E,\Gamma,\Delta_1' \cup \Delta_5')$, to conclude that

$$(E,\Gamma,\Delta_1) \xrightarrow{\ell} (E,\Gamma,\Delta_1') \tag{C.27}$$

$$(E,\Gamma,\Delta_5) \xrightarrow{\ell'} (E,\Gamma,\Delta_5') \tag{C.28}$$

From C.25 and C.27 we conclude that $E,\Gamma \vdash P \triangleright \Delta_1 \xrightarrow{\ell} E,\Gamma \vdash P' \triangleright \Delta_1'$ and from C.26 and C.28 $E,\Gamma \vdash R \triangleright \Delta_5 \xrightarrow{\ell} E,\Gamma \vdash R' \triangleright \Delta_5'$.

From the definition of set S we get that $E,\Gamma \vdash Q \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} E,\Gamma \vdash Q' \triangleright \Delta_2'$, implies

$$Q \stackrel{\ell}{\Longrightarrow} Q' \tag{C.29}$$

$$(E,\Gamma,\Delta_2) \stackrel{\ell}{\Longrightarrow} (E,\Gamma,\Delta_2') \tag{C.30}$$

From C.26 we get that $Q \mid R \stackrel{\tau}{\Longrightarrow} Q' \mid R'$ and $(E,\Gamma,\Delta_2 \cup \Delta_5) \stackrel{\tau}{\Longrightarrow} (E,\Gamma,\Delta_2' \cup \Delta_5')$, implies

$$E,\Gamma \vdash Q \mid R \triangleright \Delta_4 \stackrel{\tau}{\Longrightarrow} E' \vdash Q' \mid R' \triangleright \Delta_4'$$

Case: 3

$$E,\Gamma \vdash P \mid R \triangleright \Delta_3 \xrightarrow{\ell} E' \vdash P \mid R' \triangleright \Delta_3'$$

□

C.3.5 Proof for Lemma 6.5.5

Proof. We take into advantage the fact that bisimulation has a stratifying definition.

- $\approx_{g0}$ is the union of all configuration relations, $E,\Gamma \vdash P \triangleright \Delta_1 \; R \; Q \triangleright \Delta_2$.
- $E,\Gamma \vdash P \triangleright \Delta_1 \approx_{g_n} Q \triangleright \Delta_2$ if
  - $E,\Gamma \vdash P \triangleright \Delta_1 \xrightarrow{\ell} E',\Gamma \vdash P' \triangleright \Delta_1'$ then $E,\Gamma \vdash Q \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} E',\Gamma \vdash Q \triangleright \Delta_2'$ and $E',\Gamma \vdash P' \triangleright \Delta_1' \approx_{g_{n-1}} Q' \triangleright \Delta_2'$
  - The symmetric case.
- $\approx_{g_n}^{\omega} = \bigcap_{0 \leq i \leq n} \approx_{g_i}$

From coinduction theory, we know that $(\bigcap_{\forall n} \approx_{g_n}) = \approx_g$.

To this purpose we define a set of tests $T\langle N, \vec{\ell_n}\rangle$ to inductively show that:

$$
\begin{array}{ll}
\text{If} & E,\Gamma \vdash P_1 \triangleright \Delta_1 \cong_g P_2 \triangleright \Delta_2 \text{ implies} \\
 & E,\Gamma \vdash P_1 \mid T\langle N, \vec{\ell_n}\rangle \triangleright \Delta_1 \cong_g P_2 \mid T\langle N, \vec{\ell_n}\rangle \triangleright \Delta_2 \text{ implies} \\
\forall n, & E,\Gamma \vdash P_1 \triangleright \Delta_1 \approx_{g_n} P_2 \triangleright \Delta_2 \text{ implies} \\
 & E,\Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2
\end{array}
$$

We give the definition for $T\langle N, \vec{\ell_n}\rangle$:

$T\langle N, succ, \vec{\ell_n}\rangle = Q\langle N, n, \vec{\ell_i}\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell_i}\rangle$

where

1. $i \in I$
2. $\bigcup_{i \in I} \vec{\ell_i} = \vec{\ell_n}$

3. $n \;::=\; s[\mathtt{p}] \;\mid\; a.$

and let

- $B^s\langle s[\mathtt{p}]\rangle = \mathbf{0}$
- $B^i\langle s[\mathtt{p}]\rangle = s[\mathtt{p}][\mathtt{i} : \emptyset]$
- $B^o\langle s[\mathtt{p}]\rangle = s[\mathtt{p}][\mathtt{o} : \emptyset]$
- $B^{io}\langle s[\mathtt{p}]\rangle = s[\mathtt{p}][\mathtt{i} : \emptyset] \mid s[\mathtt{p}][\mathtt{o} : \emptyset]$

to define

- $Q\langle N, a, a[A](s) \cdot \vec{\ell}_n\rangle = \overline{a}[n](x).Q\langle N, s[n], \vec{\ell}_i\rangle \mid \ldots \mid a[\mathtt{p}](x).Q\langle N, s[\mathtt{p}], \vec{\ell}_i\rangle, i \in I.$
- $Q\langle N, s[\mathtt{q}], s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \cdot \vec{\ell}_n\rangle = s[\mathtt{q}][\mathtt{p}]!\langle v\rangle; Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle \mid B\langle s[\mathtt{q}]\rangle.$
- $Q\langle N, s[\mathtt{q}], s[\mathtt{p}][\mathtt{q}]\&l \cdot \vec{\ell}_n\rangle = s[\mathtt{q}][\mathtt{p}] \oplus l; Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle \mid B\langle s[\mathtt{q}]\rangle.$
- $Q\langle N, a, \overline{a}[A](s) \cdot \vec{\ell}_n\rangle = a[q](x).Q\langle N, s[q], \vec{\ell}_i\rangle \mid \ldots \mid a[\mathtt{p}](x).Q\langle N, s[\mathtt{p}], \vec{\ell}_i\rangle, i \in I.$
- $Q\langle N, s[\mathtt{q}], s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \cdot \vec{\ell}_n\rangle =$
  $s[\mathtt{q}][\mathtt{p}]?(x); \mathtt{if}\ x \in N\ \mathtt{then}\ Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle\ \mathtt{else}\ (\nu\ b)(b[1](x).Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle) \mid B\langle s[\mathtt{q}]\rangle.$
- $Q\langle N, s[\mathtt{q}], s[\mathtt{p}][\mathtt{q}]!\langle s'[\mathtt{p}']\rangle \cdot \vec{\ell}_n\rangle =$
  $s[\mathtt{q}][\mathtt{p}]?(x); \mathtt{if}\ x \in N\ \mathtt{then}\ Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle\ \mathtt{else}\ (\nu\ b)(b[1](x).Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle) \mid B\langle s[\mathtt{q}]\rangle.$
- $Q\langle N, s[\mathtt{q}], s[\mathtt{p}][\mathtt{q}] \oplus l_k \cdot \vec{\ell}_n\rangle =$
  $s[\mathtt{q}][\mathtt{p}]\&\{l_k : Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle,\ l_i : (\nu\ b)(b[1](x).Q\langle N, s[\mathtt{q}], \vec{\ell}_n\rangle)\} \mid B\langle s[\mathtt{q}]\rangle.$
- $Q\langle N, n, \emptyset\rangle = R.$

where $R = (\nu\ b)(b[1](x).R')$ or $R = \mathbf{0}$. R completes the session type on session channel n and is used to keep processes typed.

From the definition of $T\langle N, \vec{\ell}_n\rangle$ we can show that $\forall T\langle N, \ell\cdot\vec{\ell}_n\rangle, T\langle N, \ell\cdot\vec{\ell}_n\rangle \stackrel{\ell'}{\Longrightarrow} T'\langle N, \vec{\ell}_n\rangle, \ell \asymp \ell'$.

We prove the required result inductively:

$E, \Gamma \vdash P_1 \triangleright \Delta_3 \cong_g P_2 \triangleright \Delta_4$ implies
$\forall \ell\cdot\vec{\ell}_n$ choose $T\langle N, \ell\cdot\vec{\ell}_n\rangle, E, \Gamma \vdash P_1 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_1 \cong_g P_2 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_2$ implies
$E, \Gamma \vdash P_1 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_1 \twoheadrightarrow P_1' \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_1'$,
$E, \Gamma \vdash P_2 \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2 \twoheadrightarrow P_2' \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'$ then by induction hypothesis
$P_1' \approx_{g_n} P_2'$ implies
$\forall n, E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_{g_n} P_2 \triangleright \Delta_2$ implies
$E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$

We need to show that if

$$E, \Gamma \vdash P_1 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_1 \cong_g P_2 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_2$$

then

$$E, \Gamma \vdash P_1 \mid T\langle N, \ell\cdot\vec{\ell}_n\rangle \triangleright \Delta_1 \twoheadrightarrow P_1' \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_1', E, \Gamma \vdash P_2 \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2 \twoheadrightarrow P_2' \mid T\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'$$

We perform a case analysis on $E, \Gamma \vdash P_1 \triangleright \Delta_3 \xrightarrow{\ell} P_1 \triangleright \Delta_3'$:

- $E, \Gamma \vdash P_1 \triangleright \Delta_3 \xrightarrow{s[\mathtt{p}][\mathtt{q}]?\langle v\rangle} P_1 \triangleright \Delta_3'$ implies, $E, \Gamma \vdash P_1 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]?\langle v\rangle\cdot\vec{\ell}_n\rangle \triangleright \Delta_1 = E, \Gamma \vdash P_1 \mid Q\langle N, s[\mathtt{p}], s[\mathtt{p}][\mathtt{q}]?\langle v\rangle\cdot\vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \triangleright \Delta_1 \longrightarrow P_1' \mid Q\langle N, s[\mathtt{p}], \vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \triangleright$

Δ_1.

$E, \Gamma \vdash P_2 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2$ needs to match the reduction, $E, \Gamma \vdash P_2 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2 \twoheadrightarrow E, \Gamma \vdash P_2''' \mid T\langle N, s[\mathtt{p}][\mathtt{q}]?\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2''' \longrightarrow E, \Gamma \vdash P_2'' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'' \twoheadrightarrow E, \Gamma \vdash P_2' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'$

- $E, \Gamma \vdash P_1 \triangleright \Delta_3 \xrightarrow{a[A](s)} P_1 \vdash \Delta_3' \triangleright$ implies, $E, \Gamma \vdash P_1 \mid T\langle N, a[A](s) \cdot \vec{\ell}_n\rangle \triangleright \Delta_1 = E, \Gamma \vdash P_1 \mid Q\langle N, a, a[A](s) \cdot \vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \triangleright \Delta_1 \longrightarrow P_1' \mid Q\langle N, a, \vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \vdash \Delta_1 \triangleright$.

  $E, \Gamma \vdash P_2 \mid T\langle N, a[A](s) \cdot \vec{\ell}_n\rangle \triangleright \Delta_2$ needs to match the reduction $E, \Gamma \vdash P_2 \mid T\langle N, a[A](s) \cdot \vec{\ell}_n\rangle \triangleright \Delta_2 \twoheadrightarrow E, \Gamma \vdash P_2''' \mid T\langle N, a[A](s) \cdot \vec{\ell}_n\rangle \triangleright \Delta_2''' \longrightarrow E, \Gamma \vdash P_2'' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'' \twoheadrightarrow E, \Gamma \vdash P_2' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'$

- $E, \Gamma \vdash P_1 \triangleright \Delta_3 \xrightarrow{s[\mathtt{p}][\mathtt{q}]!\langle v\rangle} P_1 \vdash \Delta_3' \triangleright$ implies, $E, \Gamma \vdash P_1 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_1 = E, \Gamma \vdash P_1 \mid Q\langle N, s[\mathtt{p}], s[p][q]!\langle v\rangle \cdot \vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \triangleright \Delta_1 \twoheadrightarrow P_1' \mid Q\langle N, s[\mathtt{p}], \vec{\ell}_i\rangle \mid \ldots \mid Q\langle N, n, \vec{\ell}_i\rangle \vdash \Delta_1 \triangleright$.

  $E, \Gamma \vdash P_2 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2$ needs to match the reduction, $E, \Gamma \vdash P_2 \mid T\langle N, s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2 \twoheadrightarrow E, \Gamma \vdash P_2''' \mid T\langle N, s[\mathtt{p}][\mathtt{q}]!\langle v\rangle \cdot \vec{\ell}_n\rangle \triangleright \Delta_2''' \twoheadrightarrow E, \Gamma \vdash P_2'' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'' \twoheadrightarrow E, \Gamma \vdash P_2' \mid T'\langle N, \vec{\ell}_n\rangle \triangleright \Delta_2'$

□

C.3.6 Proof for Theorem 6.5.2

We first show that session actions are closed inside the bisimularity relation.

Lemma C.3.4 (Session endpoint linearity)**.** If $\Gamma \vdash P_1 \triangleright \Delta_1 \Longrightarrow_b \Gamma \vdash P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx \Gamma \vdash P_2 \triangleright \Delta_2$.

Proof. We are based on the fact that $\longrightarrow_s$ is a linear transition. We define the relation $S = \{(\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2)\} \cup R$ where R is the reflexive relation on the derivatives of $\Gamma \vdash P_2 \triangleright \Delta_2$.

Because $\Longrightarrow_s$ is linear we can easily show that S is a bisimulation. □

Proof for Theorem 6.5.2

Proof. We use lemma C.3.4 to achieve the required result.

Case: $\alpha = s, \alpha' = i$

Let

$$S = \{(P_1, P_2) \mid P_1 \approx_s P_2\}$$

We can show that if $P_1 \; S \; P_2$

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma \vdash P_1'' \triangleright \Delta_1''$ then $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} \Gamma \vdash P_2' \triangleright \Delta_2'$ and $\Gamma \vdash P_1'' \triangleright \Delta_1'' \Longrightarrow_b \Gamma \vdash P_1' \triangleright \Delta_1' \; S \; \Gamma \vdash P_2' \triangleright \Delta_2'$.

2. The symmetric case.

From lemma C.3.4 we get that relation S is a bisimulation up-to $\Longrightarrow_b$.

Case: $\alpha = s, \alpha' = o$

Let

$$S = \{(P_1, P_2) \mid P_1 \approx_s P_2\}$$

We can show that if $P_1 \; S \; P_2$

1. $\Gamma \vdash P_1 \triangleright \Delta_1 \Longrightarrow_s \xrightarrow{\ell} \Gamma \vdash P_1' \triangleright \Delta_1'$ then $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} \Gamma \vdash P_2' \triangleright \Delta_2'$ and $\Gamma \vdash P_1' \triangleright \Delta_1' \; S \; \Gamma \vdash P_2' \triangleright \Delta_2'$.

2. The symmetric case.

From lemma C.3.4 we get that relation S a bisimulation up-to $\Longrightarrow_b$.

Case: $\alpha' = io$

We rely on similar arguments to prove that $\Gamma \vdash P_1 \triangleright \Delta_1 \Longrightarrow_b \xrightarrow{\ell} \Longrightarrow_b \Gamma \vdash P_1' \triangleright \Delta_1'$ results in a closed up-to $\Longrightarrow_b$ bisimulation relation.

For the second part of the theorem we provide the proper counter-examples.

Let processes

$$\begin{aligned} P_1 &= s_1[\mathtt{p}][\mathtt{q}]!\langle v\rangle; s_2[\mathtt{p}][\mathtt{q}]!\langle w\rangle; \mathbf{0} \mid s_1[\mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \\ P_2 &= s_2[\mathtt{p}][\mathtt{q}]!\langle w\rangle; s_1[\mathtt{p}][\mathtt{q}]!\langle v\rangle; \mathbf{0} \mid s_1[\mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \end{aligned}$$

and

$$E, \Gamma \vdash P_1 \triangleright \Delta$$
$$E, \Gamma \vdash P_2 \triangleright \Delta$$

with

$$E = s_1 : \mathtt{p} \to \mathtt{q} : \langle V \rangle.\mathtt{end} \cdot s_2 : \mathtt{p} \to \mathtt{q} : \langle W \rangle.\mathtt{end}$$

If we consider the input asynchronous MSP semantics we can observe the action

$$E, \Gamma \vdash P_1 \triangleright \Delta \xrightarrow{s_1[\mathtt{p}][\mathtt{q}]!\langle v\rangle}$$

The same action cannot be observed for the typed process:

$$E, \Gamma \vdash P_2 \triangleright \Delta \not\xrightarrow{s_1[\mathtt{p}][\mathtt{q}]!\langle v\rangle}$$

If we consider the output asynchronous MSP semantics we can build a bisimulation closure $\mathscr{R}$ on $E,\Gamma \vdash P_1 \triangleright \Delta$ and $E,\Gamma \vdash P_1 \triangleright \Delta$ as follows:

$$
\begin{aligned}
\mathscr{R} \;=\; & \{(E,\Gamma \vdash P_1 \triangleright \Delta, E,\Gamma \vdash P_2 \triangleright \Delta) \\
& (s_2[\mathtt{p}][\mathtt{q}]!\langle w\rangle;\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon],\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w]) \\
& (\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w], s_1[\mathtt{p}][\mathtt{q}]!\langle v\rangle;\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w]) \\
& (\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w],\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w]) \\
& (\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w],\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:w]) \\
& (\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:v] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]) \\
& (\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]\mathbf{0} \mid s_1[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]) \\
& \}
\end{aligned}
$$

This concludes that $\approx_g^i$ and $\approx_g^o$ are incompatible. □

C.3.7 Proof for Lemma 6.5.4

Proof. We prove direction if $\forall E, E,\Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta_1 \approx \Gamma \vdash P_2 \triangleright \Delta_2$.

If $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} P_1' \triangleright \Delta_1'$ then $P_1 \xrightarrow{\ell} P_1'$ and $(\Gamma,\Delta_1) \xrightarrow{\ell} (\Gamma',\Delta_1')$.

From part 3 of Lemma C.3.3 we choose E such that $(E,\Gamma,\Delta_1) \xrightarrow{\ell} (E',\Gamma',\Delta_1')$. Since $\forall E, E,\Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$ it can now be implied that, $E,\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E',\Gamma \vdash P_1' \triangleright \Delta_1'$ implies, $E,\Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\ell} E',\Gamma \vdash P_2' \triangleright \Delta_2'$ implies, $P_2 \xRightarrow{\ell} P_2'$ and $(E,\Gamma,\Delta_2) \xRightarrow{\ell} (E',\Gamma',\Delta_2')$.

From part 1 of Lemma C.3.3 we get $(\Gamma,\Delta_2) \xRightarrow{\ell} (\Gamma',\Delta_2')$ implies $\Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\ell} P_2' \triangleright \Delta_2'$ as required.

We prove direction if $\Gamma \vdash P_1 \triangleright \Delta_1 \approx \Gamma \vdash P_2 \triangleright \Delta_2$ then $\forall E, E,\Gamma \vdash P_1 \triangleright \Delta_1 \approx_g P_2 \triangleright \Delta_2$.

Let $E,\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} P_1' \triangleright \Delta_1'$ then

$$P_1 \xrightarrow{\ell} P_1' \tag{C.31}$$

$$(E,\Gamma,\Delta_1) \xrightarrow{\ell} (E',\Gamma',\Delta_1') \tag{C.32}$$

If $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} P_1' \triangleright \Delta_1'$ then $P_1 \xrightarrow{\ell} P_1', (\Gamma,\Delta_1) \xrightarrow{\ell} (\Gamma',\Delta_1'), \Gamma \vdash P_2 \triangleright \Delta_2 \xrightarrow{\ell} P_2' \triangleright \Delta_2'$

From the last implication we get

$$P_2 \stackrel{\ell}{\Longrightarrow} P_2' \tag{C.33}$$

$$(\Gamma,\Delta_2) \stackrel{\ell}{\Longrightarrow} (\Gamma',\Delta_2') \tag{C.34}$$

$$\Delta_1 \rightleftharpoons \Delta_2 \tag{C.35}$$

We apply part 2 of Lemma C.3.3 to C.32 and C.35 to get $(E,\Gamma,\Delta_2) \stackrel{\ell}{\Longrightarrow} (E',\Gamma',\Delta_2')$. From the last result and C.33 we get $E,\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} E',\Gamma \vdash P_2' \triangleright \Delta_2'$. □

C.3.8 Proof for theorem 6.5.5

Proof. If $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma \vdash P_1' \triangleright \Delta_1'$ and P_1 is simple then

$$(\Gamma,\Delta_1) \xrightarrow{\ell} (\Gamma,\Delta_1')$$

$$P_1 \xrightarrow{\ell} P_1'$$

We follow the requirement of part 3 of Lemma C.3.3 to get that there $\exists E_1 \cdot E_1,\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} E_1,\Gamma \vdash P_1' \triangleright \Delta_1'$. From here we can get that $E_1 \xrightarrow{\ell} E_2$. But since P_1 is simple then $E \xrightarrow{\ell} E', \forall E \cdot E,\Gamma \vdash P_1 \triangleright \Delta_1$.

From that point on we apply part 2 of Lemma C.3.3 to get that If P_1 and P_2 are simple and

$\exists E \cdot E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx^s_g P_2 \triangleright \Delta_2$ then $\forall E, E, \Gamma \vdash P_1 \triangleright \Delta_1 \approx^s_g P_2 \triangleright \Delta_2$. By applying Lemma 6.5.4 we are done. □