On Asynchronous Session Semantics

Dimitrios Kouzapas*, Nobuko Yoshida*, Raymond Hu*, and Kohei Honda†

*Imperial College London †Queen Mary, University of London

Abstract. This paper studies a behavioural theory of the π-calculus with session types under the fundamental principles of the practice of distributed computing — asynchronous communication which is order-preserving inside each connection (session), augmented with asynchronous inspection of events (message arrivals). A new theory of bisimulations is introduced, distinct from either standard asynchronous or synchronous bisimilarity, accurately capturing the semantic nature of session-based asynchronously communicating processes augmented with event primitives. The bisimilarity coincides with the reduction-closed barbed congruence. We examine its properties and compare them with existing semantics. Using the behavioural theory, we verify that the program transformation of multithreaded into event-driven session based processes, using Lauer-Needham duality, is type and semantic preserving. Our benchmark results demonstrate the potential of the session-type based translation as semantically transparent optimisation techniques.

1 Introduction

Modern transports such as TCP in distributed networks provide reliable, ordered delivery of a stream of bytes from a program on one computer to another, once a connection is established. In practical communications programming, two parties start a conversation by establishing a connection over such a transport and exchange semantically meaningful, formatted messages through this connection. The distinction between possibly non order-preserving communications outside of connection and order-preserving ones inside each connection is an essential feature of this practice: order preservation allows proper handling of a sequence of messages following an agreed-upon conversation structure. Further, to enable asynchronous event processing [23], each arriving message needs to be *buffered* at the receiver's end, which the receiver can asynchronously *inspect* (event inspection) and *consume*.

A buffer in the π-calculus serves as an intermediary between a process and its environment. Operational semantics for buffers distill this nature, representing order-preserving, non-blocking communication for sessions; and non order-preserving, non blocking communication for shared names, for both input and output.

This paper investigates semantic foundations of asynchronously communicating processes, capturing these key elements of modern communications programming — distinction between non order-preserving communications outside connections and the order-preserving ones inside each connection, as well as the incorporation of asynchronous inspection of message arrivals. We use the π-calculus augmented with session primitives and a simple event inspection primitive. Sessions offer typed, formally founded abstraction of connections, enabling well-structured description of message passing. The formalism is intended to be a small idealised but expressive communications programming language, offering a basis for a tractable semantic study. Our

study reveals that the combination of the fundamental elements for modern communications programming leads to a rich behavioural theory which differs from both the standard synchronous communications semantics and the asynchronous one [10], manifest through its novel equational laws for asynchrony. These laws can then be used as a semantic justification of a well-known program transformation based on Lauer and Needham's duality principle [18], which translates multithreaded programs to their equivalent single-threaded, event-based programs. This transformation is regularly used in practice, albeit in an ad-hoc manner, playing a key role in e.g. high-performance servers. Our translation is given formally and is backed up by a rigorous semantic justification. It is applied to eventful Session Java [15, 16], a session-typed extension of Java with asynchronous event primitives, using which we demonstrate the practical effectiveness of our formal transformation through benchmark results.

Let us outline some of the key technical ideas informally. In the present theory, the asynchronous order-preserving communications over a connection are modelled as *asynchronous session communication*, extending the standard synchronous session calculus [11, 31] with *message queues* [6, 7, 14]. A message queue is written $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$, which encapsulates both input buffer (`i`) with elements $\vec{h}$ and output buffer (`o`) with $\vec{h}'$. Figure 1 represents the two end points of a session. A message v is first enqueued by a sender $s!\langle v\rangle;P$ at its output queue at s, which intuitively represents a communication pipe extending from the sender's locality to the receiver's. The message will eventually reach the receiver's locality, formalised as its transfer from the sender's output buffer (at s) to the receiver's input buffer (at $\overline{s}$). For a receiver, only when this transfer takes place a visible (and asynchronous) message reception takes place, since only then the receiver can *inspect* and *consume* the message (as shown in **Remote** in Figure 1). Note that dequeuing and enqueing actions inside a location are local to each process and is therefore invisible (τ-actions) (**Local** in Figure 1).

Local (the dashed arrows)

$$s!\langle v\rangle;Q \mid s[\mathtt{o}:\vec{h}] \xrightarrow{\tau} Q \mid s[\mathtt{o}:\vec{h}\cdot v]$$

$$s?(x).P \mid s[\mathtt{i}:w\cdot\vec{h}] \xrightarrow{\tau} P\{w/x\} \mid s[\mathtt{i}:\vec{h}]$$

Remote (the solid arrows)

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

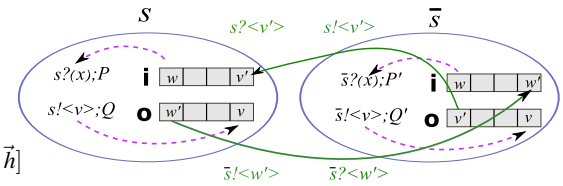

Fig. 1. The transitions in the two locations.

The induced semantics captures the nature of asynchronous observables not investigated before. For example, in weak asynchronous bisimilarity ($\approx_a$ in [10, 12]), the message order is not observable ($s!\langle v_1\rangle \mid s!\langle v_2\rangle \approx_a s!\langle v_2\rangle \mid s!\langle v_1\rangle$) but in our semantics, messages for the same destination do *not* commute ($s!\langle v_1\rangle;s!\langle v_2\rangle \not\approx s!\langle v_2\rangle;s!\langle v_1\rangle$) as in the synchronous semantics [26] ($\approx_s$ in [10, 12]); whereas two inputs for different targets commute ($s_1?(x);s_2?(y);P \approx s_2?(x);s_1?(y);P$) since the dequeue action is un-observable, differing from the synchronous semantics, $s_1?(x);s_2?(y);P \not\approx_s s_2?(x);s_1?(y);P$.

Event-handling primitives [15] lead to further novel equational laws. Asynchronous event-programming is characterised by reactive flows driven by the detection of *events*, that is *message arrivals* at local buffers. In our formalism, this facility is distilled as a simple `arrived` predicate: for example, $Q = \mathtt{if\ arrived}\ s\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ reduces to P_1 if the s input buffer contains one or more message; otherwise Q reduces to P_2. By `arrived`, we can observe the *movement of messages between two locations*. For

(Identifiers) $u ::= a,b \mid x,y \quad k ::= s,\overline{s} \mid x,y \quad n ::= a,b \mid s,\overline{s}$ (Values) $v ::= \mathtt{tt},\mathtt{ff} \mid a,b \mid s,\overline{s}$

(Expressions) $e ::= v \mid x,y,z \mid \mathtt{arrived}\, u \mid \mathtt{arrived}\, k \mid \mathtt{arrived}\, k\, h$

(Processes) $P,Q ::= u(x).P \mid \overline{u}\langle x\rangle;P \mid k!\langle e\rangle;P \mid k?(x).P \mid k \triangleleft l;P \mid k \triangleright \{l_i : P_i\}_{i\in I}$
$\mid \mathtt{if}\, e\, \mathtt{then}\, P\, \mathtt{else}\, Q \mid (\nu\, a)P \mid P \mid Q \mid \mathbf{0} \mid \mu X.P \mid X$
$\mid a[\vec{s}] \mid \overline{a}\langle s\rangle \mid (\nu\, s)P \mid s[\mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}]$ (Messages) $h ::= v \mid l$

Fig. 2. The syntax of processes.

example, $Q \mid s[\mathtt{i} : \emptyset] \mid \overline{s}[\mathtt{o} : v]$ is not equivalent with $Q \mid s[\mathtt{i} : v] \mid \overline{s}[\mathtt{o} : \emptyset]$ because the former can reduce to P_2 (since v has not arrived at the local buffer at s yet) while the latter cannot. To capture `arrived`, we need the IO-buffers at each session, with which one has, for example, $s_1!\langle v_1\rangle;s_2!\langle v_2\rangle \approx s_2!\langle v_2\rangle;s_1!\langle v_1\rangle$ in the presence of the `arrived` (we cannot observe the remote process performing the output), while one would have $s_1!\langle v_1\rangle;s_2!\langle v_2\rangle \not\approx s_2!\langle v_2\rangle;s_1!\langle v_1\rangle$ with the `arrived`, *if* we were without local queues.

This way, our asynchronous session semantics captures asynchrony, locality, event inspection and orders of actions at session channels, all being major elements of practical communications programming. The induced event- and location-aware asynchronous behavioural equivalence is essential for validating a well-known, effective but hitherto unjustified program transformation.

Contributions Section 3 defines a bisimulation for the asynchronous eventful session calculus, examines its properties against the standard bisimulations and proves that it coincides with the barbed reduction-based congruence [12]. Section 4 provides the semantics-preserving Lauer-Needham transformation of multithreaded into event-driven processes and proves its correctness. Section 5 discusses the performance with extensive benchmarks for justifying the transformation in Session-based Java implementation [15]. The paper concludes with the related work with the detailed comparisons with the existing semantics. Appendix lists the full definition of Lauer-Needham transformation, the detailed definitions and full proofs. The benchmark programs and results are found in [29]. Appendix and [29] are provided only for the reviewers' convenience: the paper is self-contained and can be read without them.

2 Asynchronous Network Communications in Sessions

2.1 Syntax

We use a sub-calculus of the eventful session π-calculus [15]. In Figure 2, values $v, v', \ldots$ include the constants, *shared channels* a,b,c, and *session channels* s,s'. A session channel designates one endpoint of a session, where s and $\overline{s}$ denote two ends of a single session, with $\overline{\overline{s}} = s$. Labels for branching and selection range over $l, l', \ldots$, variables over x,y,z, and process variables over X,Y,Z. Shared channel identifiers u,u' include shared channels and variables; session identifiers k,k' are session endpoints and variables. Identifies n denotes either a or s. Expressions e are values, variables and the message arrival predicates (`arrived` u, `arrived` k and `arrived` $k\ h$: the last one checks for the arrival of the specific message h at k). We write $\vec{s}$ and $\vec{h}$ for vectors of session channels and messages respectively, writing ε for the empty vector.

We distinguish two kinds of asynchronous communications, *asynchronous session initiation* and *asynchronous session communication* (over an established session). The

$$
\begin{array}{ll}
\text{[Request1]} & \overline{a}(x);P \longrightarrow (\nu\, s)\,(P\{\overline{s}/x\} \mid \overline{s}\,[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid \overline{a}\langle s\rangle) \quad (s \notin \mathrm{fn}(P)) \\
\text{[Request2]} & a\,[\vec{s}] \mid \overline{a}\,\langle s\rangle \longrightarrow a\,[\vec{s}\cdot s] \\
\text{[Accept]} & a(x).P \mid a\,[s\cdot\vec{s}] \longrightarrow P\{s/x\} \mid s\,[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid a\,[\vec{s}] \\
\text{[Send,Recv]} & s!\langle v\rangle;P \mid s\,[\mathtt{o}:\vec{h}] \longrightarrow P \mid s\,[\mathtt{o}:\vec{h}\cdot v] \qquad s?(x).P \mid s\,[\mathtt{i}:v\cdot\vec{h}] \longrightarrow P\{v/x\} \mid s\,[\mathtt{i}:\vec{h}] \\
\text{[Sel,Bra]} & s\triangleleft l_i;P \mid s\,[\mathtt{o}:\vec{h}] \longrightarrow P \mid s\,[\mathtt{o}:\vec{h}\cdot l_i] \quad s\triangleright\{l_j:P_j\}_{j\in J} \mid s\,[\mathtt{i}:l_i\cdot\vec{h}] \longrightarrow P_i \mid s\,[\mathtt{i}:\vec{h}] \quad (i\in J) \\
\text{[Comm]} & s\,[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}\,[\mathtt{i}:\vec{h}'] \longrightarrow s\,[\mathtt{o}:\vec{h}] \mid \overline{s}\,[\mathtt{i}:\vec{h}'\cdot v] \\
\text{[Areq]} & E[\mathtt{arrived}\ a] \mid a\,[\vec{s}] \longrightarrow E[\mathtt{b}] \mid a\,[\vec{s}] \qquad (|\vec{s}|\geq 1)\searrow \mathtt{b} \\
\text{[Ases]} & E[\mathtt{arrived}\ s] \mid s\,[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s\,[\mathtt{i}:\vec{h}] \qquad (|\vec{h}|\geq 1)\searrow \mathtt{b} \\
\text{[Amsg]} & E[\mathtt{arrived}\ s\ h] \mid s\,[\mathtt{i}:\vec{h}] \longrightarrow E[\mathtt{b}] \mid s\,[\mathtt{i}:\vec{h}] \qquad (\vec{h}=h\cdot\vec{h}')\searrow \mathtt{b}
\end{array}
$$

Fig. 3. Selected reduction rules.

former involves the *unordered* delivery of a *session request message* $\overline{a}\,\langle s\rangle$, where $\overline{a}\,\langle s\rangle$ represents an asynchronous message in transit towards an acceptor at a, carrying a fresh session channel s. As in actual network, a request message will first move through the network and eventually get buffered at a receiver's end. Only then a message arrival can be detected. This aspect is formalised by the introduction of a *shared channel input queue* $a\,[\vec{s}]$, often called *shared input queue* for brevity, which denotes an acceptor's local buffer at a with pending session requests for $\vec{s}$. The intuitive meaning of the endpoint configuration $s\,[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$ is explained in Introduction.

Requester $\overline{u}(x);P$ requests a session initiation, while acceptor $u(x).P$ accepts one. Through an established session, output $k!\langle e\rangle;P$ sends e through channel k asynchronously, input $k?(x).P$ receives through k, selection $k\triangleleft l;P$ chooses the branch with label l, and branching $k\triangleright\{l_i:P_i\}_{i\in I}$ offers branches. The $(\nu\, a)\,P$ binds a channel a, while $(\nu\, s)\,P$ binds the two endpoints, s and $\overline{s}$, making them private within P. The conditional, parallel composition, recursions and inaction are standard. $\mathbf{0}$ is often omitted. For brevity, one or more components may be omitted from a configuration when they are irrelevant, writing e.g. $s\,[\mathtt{i}:\vec{h}]$ which denotes the input part of $s\,[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$. The notions of free variables and channels are standard [27]; we write $\mathrm{fn}(P)$ for the set of free channels in P. The syntax $\overline{a}\langle s\rangle$, $(\nu\, s)\,P$ and $s\,[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$ only appear at runtime. A closed process without runtime syntax is called *program*.

2.2 Operational Semantics

Reduction is defined over closed terms. The key rules are given in Figure 3. We use the standard evaluation contexts $E[\_]$ defined as $E ::= \ -\ \mid s!\langle E\rangle;P \mid \mathtt{if}\ E\ \mathtt{then}\ P\ \mathtt{else}\ Q$. The structural congruence $\equiv$ and the rest of the reduction rules are standard.

The first three rules define the initialisation. In [Request1], a client requests a server for a fresh session via shared channel a. A fresh session channel, with two ends s (server-side) and $\overline{s}$ (client-side) as well as the empty configuration at the client side, are generated and the session request message $\overline{a}\langle s\rangle$ is dispatched. Rule [Request2] enqueues the request in the shared input queue at a. A server accepts a session request from the queue using [Accept], instantiating its variable with s in the request message; the new session is now established.

Asynchronous order-preserving session communications are modelled by the next four rules. Rule [Send] enqueues a value in the o-buffer at the *local* configuration; rule [Receive] dequeues the first value from the i-buffer at the local configuration; rules [Sel]

and [Bra] similarly enqueue and dequeue a label. The arrival of a message at a remote site is embodied by [Comm], which removes the first message from the o-buffer of the sender configuration and enqueues it in the i-buffer at the receiving configuration. Note the first four rules manipulate only the local configurations.

Output actions are always non-blocking. An input action can block if no message is available at the corresponding local input buffer. The use of the message arrivals can avoid this blocking: [Areq] evaluates `arrived` a to `tt` iff the queue is non-empty; similarly for `arrived` k in [Areq]. [Amsg] evaluates `arrived` $s\ h$ to `tt` iff the buffer is nonempty and its next message matches h. The notation $e \searrow \mathtt{b}$ means e evaluates to `b`. We write $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$ (see Appendix A for detailed illustration).

2.3 Types and Typing

The type syntax follows the standard session types from [11].

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

The shared types U include booleans bool (and, in examples, naturals nat); shared channel types $\mathtt{i}\langle S\rangle$ (input) and $\mathtt{o}\langle S\rangle$ (output) for shared channels through which a session of type S is established; type variables $(\mathsf{X},\mathsf{Y},\mathsf{Z},..)$; and recursive types. The IO-types (often called server/client types) ensure a unique server and many clients [4, 13]. In the present work they are used for controlling locality (queues are placed only at the server sides) and associated typed transitions, playing a central role in our behavioural theory together with session types.

In session types, output type $!(T);S$ represents outputting values of type T, then performing as S. Dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i\in I}$ describes a selection of one of the labels say l_i then behaves as T_i. Branching type $\&\{l_i : S_i\}_{i\in I}$ waits with I options, and behaves as type T_i if i-th label is chosen. End type `end` represents the session completion and is often omitted. In recursive type $\mu\mathsf{X}.S$, type variables are guarded in the standard sense.

The typing judgement from [15] uses two environments:

$$\Gamma ::= \emptyset \mid \Gamma\cdot u : U \mid \Gamma\cdot X : \vec{T} \qquad \Delta ::= \emptyset \mid \Delta\cdot a \mid \Delta\cdot k : S \mid \Delta\cdot s : (S, [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T}'])$$

Γ is called *shared environment*, which maps shared channels and process variables to, respectively, constant types and value types; Δ is called *linear environment* mapping session channels to session types and recording shared channels for acceptor's input queues. The configuration element $[S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T}']$ records the active type S of the configuration and types of messages enqueued in the buffers where, in this case only, we extend T with labels ($T ::= U \mid S \mid l$). The initial S in $(S, [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T}'])$ denotes the session type of a process. Now the typing judgement for a process is given as:

$$\Gamma \vdash P \triangleright \Delta$$

where program P, under shared environment Γ, uses channels as linear environment Δ, where we stipulate that, if a is in Δ, $\Gamma(a)$ should be an input shared type. The typing system is identical with [15], thus we leave them to Appendix B. The key properties of typed processes stated in [15] are:

– (queues under hiding) The hiding of a shared (resp. session) channel is only possible when a queue (resp. a pair of queues) at that channel is present.
– (subject reduction) if $\Gamma \vdash P \triangleright \Delta$ and $P \longrightarrow Q$ then we have $\Gamma \vdash P \triangleright \Delta'$ for some Δ' (where such Δ' may not be uniquely determined by Γ and Δ).

In the following we study semantic properties of typed processes: however these developments can be understood without knowing the details of the typing rules.[1]

$$\langle \mathsf{Acc} \rangle \quad a[\tilde{s}] \xrightarrow{a\langle s \rangle} a[\tilde{s} \cdot s] \qquad \langle \mathsf{Req} \rangle \quad \overline{a}\langle s \rangle \xrightarrow{\overline{a}\langle s \rangle} \mathbf{0} \qquad \langle \mathsf{In} \rangle \quad s[\mathtt{i} : \vec{h}] \xrightarrow{s?\langle v \rangle} s[\mathtt{i} : \vec{h} \cdot v]$$

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

$$\langle \mathsf{Local} \rangle \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \quad \langle \mathsf{Par} \rangle \frac{P \xrightarrow{\ell} P' \ \mathsf{bn}(\ell) \cap \mathsf{fn}(Q) = \emptyset}{P|Q \xrightarrow{\ell} P'|Q} \quad \langle \mathsf{Tau} \rangle \frac{P \xrightarrow{\ell} P' \ Q \xrightarrow{\ell'} Q' \ \ell \asymp \ell'}{P|Q \xrightarrow{\tau} (\nu \, \mathsf{bn}(\ell, \ell'))(P'|Q')}$$

$$\langle \mathsf{Res} \rangle \frac{P \xrightarrow{\ell} P' \ n \notin \mathsf{fn}(\ell)}{(\nu n)P \xrightarrow{\ell} (\nu n)P'} \quad \langle \mathsf{OpenS} \rangle \frac{P \xrightarrow{\overline{a}\langle s \rangle} P'}{(\nu s)P \xrightarrow{\overline{a}(s)} P'} \quad \langle \mathsf{OpenN} \rangle \frac{P \xrightarrow{\overline{s}\langle a \rangle} P'}{(\nu a)P \xrightarrow{\overline{s}(a)} P'} \quad \langle \mathsf{Alpha} \rangle \frac{P \equiv_\alpha P' \ P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Fig. 4. Labelled transition system (we omit the symmetric rule of Par).

3 Asynchronous Session Bisimulations and its Properties

3.1 Labelled Transitions and Bisimilarity

Untyped and Typed LTS. This section studies the basic properties of behavioural equivalences. We use the following labels ($\ell, \ell', ...$):

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

where the labels denote the session accept, request and bound request, input, output, bound output, branching, selection and the τ-action. We have the labels for both shared and session channels. $\mathsf{sbj}(\ell)$ denotes the set of free subjects in ℓ; and $\mathsf{fn}(\ell)$ (resp. $\mathsf{bn}(\ell)$) denotes the set of free (resp. bound) names in ℓ. The symmetric operator $\ell \asymp \ell'$ on labels that denotes that ℓ is a dual of ℓ', is defined as: $a\langle s \rangle \asymp \overline{a}\langle s \rangle$, $a\langle s \rangle \asymp \overline{a}(s)$, $s?\langle v \rangle \asymp s!\langle v \rangle$, $s?\langle a \rangle \asymp s!(a)$, and $s\&l \asymp s \oplus l$.

Figure 4 gives the untyped labelled transition system (LTS). $\langle \mathsf{Acc} \rangle / \langle \mathsf{Req} \rangle$ are for the session initialisation. The next four rules $\langle \mathsf{In} \rangle / \langle \mathsf{Out} \rangle / \langle \mathsf{Bra} \rangle / \langle \mathsf{Sel} \rangle$ say the action is observable when it moves from its local queue to its remote queue. When the process accesses its local queue, the action is *invisible* from the outside, as formalised by $\langle \mathsf{Local} \rangle$. Other compositional rules are standard. Based on the LTS, we use the standard notations [24] such as $P \stackrel{\ell}{\Longrightarrow} Q$, $P \stackrel{\vec{\ell}}{\Longrightarrow} Q$ and $P \stackrel{\hat{\ell}}{\Longrightarrow} Q$.

We define the typed LTS on the basis of the untyped one. The basic idea is *to use the type information to control the enabling of actions*. This is realised by introducing the definition of the *environment transition*, defined in Figure 5. A transition

[1] This is because the properties of the typing system are captured by typed reductions defined in Figure 5 later.

$(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$ 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 transition is not deterministic, because the τ-action (reduction) of a process can alter its linear environment in several different ways.

$$
\begin{array}{rl}
\Gamma(a) = \mathtt{i}\langle S\rangle, a \in \Delta, s \text{ fresh} & \text{implies } (\Gamma,\Delta) \xrightarrow{a\langle s\rangle} (\Gamma,\Delta\cdot s:\overline{S}) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta & \text{implies } (\Gamma,\Delta) \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta, s \text{ fresh} & \text{implies } (\Gamma,\Delta) \xrightarrow{\overline{a}(s)} (\Gamma,\Delta\cdot s:S) \\
\Gamma \vdash v : U & \text{implies } (\Gamma,\Delta\cdot s:(S,[?(U);S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \xrightarrow{s?\langle v\rangle} (\Gamma,\Delta\cdot s:(S,[S',\mathtt{i}:\vec{T}U,\mathtt{o}:\vec{T}'])) \\
\Gamma \vdash v : U & \text{implies } (\Gamma,\Delta\cdot s:(S,[!\langle U\rangle;S',\mathtt{i}:\vec{T},\mathtt{o}:U\vec{T}'])) \xrightarrow{s!\langle v\rangle} (\Gamma,\Delta\cdot s:(S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \\
a \text{ is fresh} & \text{implies } (\Gamma,\Delta\cdot s:(S,[!\langle\langle S''\rangle\rangle;S',\mathtt{i}:\vec{T},\mathtt{o}:\langle S''\rangle\vec{T}'])) \\
& \qquad \xrightarrow{s!\langle a\rangle} (\Gamma\cdot a:\langle S''\rangle,\Delta\cdot s:(S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}']))
\end{array}
$$

$$
\begin{array}{rl}
(\Gamma,\Delta\cdot s:S) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S,[S,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon])) \\
(\Gamma,\Delta\cdot s:(!\langle U\rangle;S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'U])) \\
(\Gamma,\Delta\cdot s:(?(U);S,[S',\mathtt{i}:U\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s:(!\langle S'\rangle;S,[S'',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S,[S'',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'S'])) \\
(\Gamma,\Delta\cdot s:(?(S');S,[S'',\mathtt{i}:S'\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S,[S'',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s:(\oplus\{l_i:S_i\},[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S_i,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'l_i])) \\
(\Gamma,\Delta\cdot s:(\&\{l_i:S_i\},[S',\mathtt{i}:l_i\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{\tau} (\Gamma,\Delta\cdot s:(S_i,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s':S'\cdot s:(S,[!\langle S'\rangle;S'',\mathtt{i}:\vec{T},\mathtt{o}:S'\vec{T}'])) & \xrightarrow{s!\langle s'\rangle} (\Gamma,\Delta\cdot s:(S,[S'',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s:(S,[?(S');S'',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{s?\langle s'\rangle} (\Gamma,\Delta\cdot s':S'\cdot s:(S,[S'',\mathtt{i}:\vec{T}S',\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s:(S,[\&\{l_i:S_i\},\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}'])) & \xrightarrow{s\&l_k} (\Gamma,\Delta\cdot s:(S,[S_k,\mathtt{i}:\vec{T}l_k,\mathtt{o}:\vec{T}'])) \\
(\Gamma,\Delta\cdot s:(S,[\oplus\{l_i:S_i\},\mathtt{i}:\vec{T},\mathtt{o}:l_k\vec{T}'])) & \xrightarrow{s\oplus l_k} (\Gamma,\Delta\cdot s:(S,[S_k,\mathtt{i}:\vec{T},\mathtt{o}:\vec{T}']))
\end{array}
$$

Fig. 5. Labelled transition rules for environments.

The first rule says that reception of a message via a is possible only when a is input-typed ($\mathtt{i}$-mode) and its queue is present ($a \in \Delta$). The second is dual, saying that an output at a is possible only when a has $\mathtt{o}$-mode and no queue exists. Similarly for a bound output action. For session actions, we note that action types are changed before and after a transition. The first two rules ($\ell = s!\langle v\rangle$ and $\ell = s?\langle v\rangle$) are standard value input and output rules, where an asynchronous input is possible only when a queue is present. The third rule ($\ell = s!(a)$) is a scope opening rule.

The next seven rules ($\ell = \tau$) follow the reduction rules. The final three rules correspond to session channel output, label input and label output. Note that session channel input is subsumed by the case of $\ell = s?\langle v\rangle$.

Equivalence $\bowtie$ and Typed Relation. Before the introduction of typed bisimilarity, we define several preliminary notions, including a notion of typed relations over processes. First, the *composition of two environments*, say Δ_1 and Δ_2, is denoted by $\Delta_1 \odot \Delta_2$ [27], and is defined as:

$$\Delta_1 \odot \Delta_2 = \{s:(\Delta_1(s) \odot \Delta_2(s)) \mid s \in \mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2)\} \cup \Delta_1 \setminus \mathrm{dom}(\Delta_2) \cup \Delta_2 \setminus \mathrm{dom}(\Delta_1)$$

where $S \odot [S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'] = [S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'] \odot S = [S', \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}']$ if $S \leq S'$; otherwise undefined. We first introduce Δ-ordering, which represents how session environments are updated as typable processes are reduced. We define:

1.	$s:!(T);S$	$\odot\ s:[!(T);S, \mathtt{o}:\vec{T}]$	$\sqsubset$	$s:S$	$\odot\ s:[S, \mathtt{o}:\vec{T}\cdot T]$
2.	$s:?(T);S$	$\odot\ s:[?(T);S, \mathtt{i}:T\cdot\vec{T}]$	$\sqsubset$	$s:S$	$\odot\ s:[S, \mathtt{i}:\vec{T}]$
3.	$s:[\mathtt{o}:T\cdot\vec{T}]$	$\odot\ \overline{s}:[\mathtt{i}:\vec{T}']$	$\sqsubset$	$s:[\mathtt{o}:\vec{T}]$	$\odot\ \overline{s}:[\mathtt{i}:\vec{T}'\cdot T]$
4.	$s:\oplus\{l_i:S_i\}_{i\in I}$	$\odot\ s:[\oplus\{l_i:S_i\}_{i\in I}, \mathtt{o}:\vec{T}]$	$\sqsubset$	$s:S_i$	$\odot\ s:[S_i, \mathtt{o}:\vec{T}\cdot l_i]$
5.	$s:\&\{l_i:S_i\}_{i\in I}$	$\odot\ s:[\&\{l_i:S_i\}_{i\in I}, \mathtt{i}:l_i\cdot\vec{T}]$	$\sqsubset$	$s:S_i$	$\odot\ s:[S_i, \mathtt{i}:\vec{T}]$
6.	$s:\{S_i\}_{i\in I}$	$\odot\ s:[S_i]$	$\sqsubset$	$s:S_i$	$\odot\ s:[S_i]$
7.	$s:\mu\mathsf{X}.S$	$\odot\ s':[\mu\mathsf{X}.S, \mathtt{i}:\vec{T}_1, \mathtt{o}:\vec{T}_2]$	$\sqsubset$	$s:S'$	$\odot\ s':[S', \mathtt{i}:\vec{T}_1', \mathtt{o}:\vec{T}_2']$
	if $s:S\{\mu\mathsf{X}.S/\mathsf{X}\}$	$\odot\ s':[S\{\mu\mathsf{X}.S/\mathsf{X}\}, \mathtt{i}:\vec{T}_1, \mathtt{o}:\vec{T}_2]$	$\sqsubset$	$s:S'$	$\odot\ s':[S', \mathtt{i}:\vec{T}_1', \mathtt{o}:\vec{T}_2']$

In (7), s' is s or $\overline{s}$. When $\Delta_1 \sqsubset \Delta_2$ and $\Delta \odot \Delta_1$ defined, we define $\Delta \odot \Delta_1 \sqsubset \Delta \odot \Delta_2$.

Now write $\sqsubseteq$ for the reflexive and transitive closure of $\sqsubset$ and write $\Delta \bowtie \Delta'$ if $\Delta \sqsubseteq \Delta'$ or $\Delta' \sqsubseteq \Delta$. Then we say a relation on typed processes is a *typed relation* if, whenever it relates two typed processes, we have $\Gamma \vdash P_1 \triangleright \Delta_1$ and $\Gamma \vdash P_2 \triangleright \Delta_2$ and $\Delta_1 \bowtie \Delta_2$ under Γ. We write $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ if $(\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2)$ are in a typed relation $\mathscr{R}$. Further we often leave the environments implicit, writing simply $P_1 \mathscr{R} P_2$.

Localisation and Bisimulation. A bisimulation we shall introduce is a typed relation over processes which are *localised*, in the sense that each process is equipped with its local queues. Formally, let P be closed and $\Gamma \vdash P \triangleright \Delta$. Then we say P is $\Gamma \vdash P \triangleright \Delta$ is *localised* if the following two conditions hold.

- For each $s \in \mathsf{dom}(\Delta)$, we have $\Delta(s) = (S, [S, \mathtt{i}:\vec{T}, \mathtt{o}:\vec{T}'])$ (i.e. P contains its buffer).
- Similarly for shared channels, i.e. if $\Gamma(a) = \mathtt{i}\langle S\rangle$, then $a \in \mathsf{dom}(\Delta)$.

Substantively, being localised means that a process owns all necessary queues as specified in environments: for this reason, it is natural to say P is *localised w.r.t. Γ and Δ* if $\Gamma \vdash P \triangleright \Delta$ is localised. Further we often simply say *P is localised* if it is so for a suitable pair of environments.

For example, $s?(x);s!\langle x+1\rangle;\mathbf{0}$ is not localised: we need a queue at s. However $s?(x);s!\langle x+1\rangle;\mathbf{0} \mid [s, \mathtt{i}:\vec{h}_1, \mathtt{o}:\vec{h}_2]$ is localised. Similarly, $a(x).P$ is not localised, but $a(x).P \mid a[\vec{s}]$ is. By composing buffers at appropriate channels, any typable closed process can become localised. If P is localised w.r.t. (Γ, Δ) then $P \longrightarrow Q$ implies Q is localised w.r.t. (Γ, Δ), since queues never go away. We can now introduce the asynchronous bisimilarity.

Definition 3.1 (Asynchronous Session Bisimulation). A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, the following two conditions holds: (1) $\Gamma \vdash P_1 \triangleright \Delta_1 \stackrel{\ell}{\longrightarrow} \Gamma' \vdash P_1' \triangleright \Delta_1'$ 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 $\Delta_1' \bowtie \Delta_2'$ holds and (2) the symmetric case of (1). 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 which are without queues). Further $\approx$ is extended to open terms in the standard way [12].

3.2 Properties of Asynchronous Session Bisimilarity

Characterisation of Reduction Congruence. This subsection studies central properties of asynchronous session semantics. We first show that the bisimilarity coincides with the naturally defined reduction-closed congruence [12], given below.

Definition 3.2 (Reduction Congruence). We write $P \downarrow a$ if $P \equiv (\nu\, \vec{n})(\overline{a}\langle s\rangle \mid R)$ with $a \notin \vec{n}$. Similarly we write $P \downarrow s$ if $P \equiv (\nu\, \vec{n})(s[\mathsf{o} : h\cdot\vec{h}] \mid R)$ with $s \notin \vec{n}$. $P \Downarrow n$ means $\exists P'.P \twoheadrightarrow P' \downarrow n$. A typed relation $\mathscr{R}$ is *reduction congruence* if it is a congruent and satisfies the following conditions for each $P_1 \mathscr{R} P_2$ whenever they are localised w.r.t. their given environments.

1. $P_1 \Downarrow n$ iff $P_2 \Downarrow n$.
2. Whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \cong P_2 \triangleright \Delta_2$ 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 $\Delta_1' \bowtie \Delta_2'$ and the symmetric case.

The maximum reduction congruence which is not a universal relation exists [12] which we call *reduction congruency*, denoted by $\cong$.

Theorem 3.3 (Soundness and Completeness). $\approx = \cong$.

The soundness ($\approx \subset \cong$) is by showing $\approx$ is congruent. The most difficult case is a closure under parallel composition, which requires to check the side condition $\Delta_1' \bowtie \Delta_2'$ for each case. The completeness ($\cong \subset \approx$) follows [9, § 2.6] where we prove that every external action is definable by a testing process, see Appendix D.1.

Asynchrony and Session Determinacy. Let us call ℓ an *output action* if ℓ is one of $\overline{a}\langle s\rangle, \overline{a}(s), s!\langle v\rangle, s!(a), s \oplus l$; and an *input action* if ℓ is one of $a\langle s\rangle, s?\langle v\rangle, s\&l$. 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 3.4 (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* $\Gamma \vdash P \triangleright \Delta \Longrightarrow\!\!\stackrel{\ell}{\longrightarrow} P' \triangleright \Delta'$.

The asynchronous environment interaction on the session buffers enables input actions to happen before multi-internal steps and output actions to happen after multi-internal steps.

Following [28], we define determinacy and confluence. Below and henceforth we often omit the environments in typed transitions.

Definition 3.5 (Determinacy). We say $\Gamma' \vdash Q \triangleright \Delta'$ is *derivative* of $\Gamma \vdash P \triangleright \Delta$ if there exists $\vec{\ell}$ such that $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} \Gamma' \vdash Q \triangleright \Delta'$. We say $\Gamma \vdash P \triangleright \Delta$ is *determinate* if for each derivative Q of P and action ℓ, if $Q \stackrel{\ell}{\longrightarrow} Q'$ and $Q \stackrel{\ell}{\Longrightarrow} Q''$ then $Q' \approx Q''$.

We then extend the above notions to session communications.

Definition 3.6 (Session Determinacy). Let us write $P \xrightarrow{\ell}_s Q$ if $P \xrightarrow{\ell} Q$ where $\mathsf{sbj}(\ell)$ is a session channel; or $\ell = \tau$ and it is generated without using [Request1], [Request2], [Accept], [Areq] nor [Amsg] in Figure 3 (i.e. a communication is performed without arrival predicates or shared name interactions). We extend the definition to $\stackrel{\vec{\ell}}{\Longrightarrow}_s$ and $\stackrel{\hat{\ell}}{\Longrightarrow}_s$ 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 define $\ell_1 \lfloor \ell_2$ as (1) $\overline{a}\langle s \rangle$ if $\ell_1 = \overline{a}(s')$ and $s' \in \mathsf{bn}(\ell_2)$; (2) $s!\langle s' \rangle$ if $\ell_1 = s!\langle s' \rangle$ and $s' \in \mathsf{bn}(\ell_2)$; (3) $s!\langle a \rangle$ if $\ell_1 = s!(a)$ and $a \in \mathsf{bn}(\ell_2)$; and otherwise ℓ_1. We write that $l_1 \bowtie l_2$ when $l_1 \neq l_2$ and if l_1, l_2 are input actions then $\mathsf{sbj}(l_1) \neq \mathsf{sbj}(l_2)$.

Definition 3.7 (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 $\ell_1 \bowtie \ell_2$, (i) if $Q \xrightarrow{\ell} Q_1$ and $Q \stackrel{\ell}{\Longrightarrow} Q_2$, then $Q_1 \Longrightarrow Q_1'$ and $Q_2 \Longrightarrow Q_2' \approx Q_1'$; and (ii) if $Q \xrightarrow{\ell_1} Q_1$ and $Q \stackrel{\ell_2}{\Longrightarrow} Q_2$, then $Q_1 \stackrel{\ell_2 \lfloor \ell_1}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\ell_1 \lfloor \ell_2}{\Longrightarrow} Q_2' \approx Q_1'$.

Lemma 3.8. *Let P be session determinate and Q be a session derivative of P. Then $P \approx Q$.*

Theorem 3.9 (Session Determinacy). *Let P be session determinate. Then P is determinate and confluent.*

The following relation is used to prove the thread elimination optimisation.

Definition 3.10 (Determinate Upto-expansion Relation). Let $\mathscr{R}$ be a typed relation such that if $\Gamma \vdash P \triangleright \Delta \ \mathscr{R} \ Q \triangleright \Delta'$ and (1) P, Q are determinate; (2) $\Delta = \Delta'$; (3) If $\Gamma \vdash P \triangleright \Delta \xrightarrow{l} \Gamma' \vdash P'' \triangleright \Delta''$ then $\Gamma \vdash Q \triangleright \Delta \stackrel{l}{\Longrightarrow} \Gamma' \vdash Q' \triangleright \Delta'$ and $\Gamma' \vdash P'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash P' \triangleright \Delta'$ with $\Gamma' \vdash P' \triangleright \Delta' \mathscr{R} \ Q' \triangleright \Delta'$; and (4) If $\Gamma \vdash Q \triangleright \Delta \xrightarrow{l} \Gamma'' \vdash Q'' \triangleright \Delta''$ then $\Gamma \vdash P \triangleright \Delta \stackrel{l}{\Longrightarrow} \Gamma' \vdash P' \triangleright \Delta'$ and $\Gamma' \vdash P' \triangleright \Delta' \ \mathscr{R} \ Q' \triangleright \Delta'$ with $\Gamma' \vdash Q'' \triangleright \Delta'' \Longrightarrow \Gamma' \vdash Q' \triangleright \Delta'$. Then we call $\mathscr{R}$ a *determinate upto-expansion relation*, or often simply *upto-expansion relation*.

Lemma 3.11. *Let $\mathscr{R}$ be an upto-expansion relation. Then $\mathscr{R} \subset \approx$.*

The proof is by showing $\Longrightarrow \mathscr{R} \Longleftarrow$ is a bisimulation, using determinacy.

3.3 Examples of Induced Equations: Permutation and Events

This subsection shows example equations (algebra of processes) pertaining to properties of permutations of actions studied in § 3.2 and the semantic effects of `arrival` predicates. Significant equations induced by a behavioural equivalence offer insights on the nature of process behaviours we are concerned with, and play a key role in reasoning about processes.

We use the examples from § 1. Let $R_i = s_i\,[\mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_i']$ for some $\vec{h}_i, \vec{h}_i'$ and assume $s_1 \neq s_2$.

(1) Input and Output Permutations. The two actions at different channels are permutable up to $\approx$, i.e. $s_1?(x); s_2?(y); P \mid R_1 \mid R_2 \approx s_2?(y); s_1?(x); P \mid R_1 \mid R_2$. Similarly for two outputs: $s_1!\langle v_1 \rangle; s_2!\langle v_2 \rangle; P \mid R_1 \mid R_2 \approx s_2!\langle v_2 \rangle; s_1!\langle v_1 \rangle; P \mid R_1 \mid R_2$. However, an input and an output are not permutable: $s_1?(x); s_2!\langle v_2 \rangle; P \mid R_1 \mid R_2 \not\approx s_2!\langle v_2 \rangle; s_1?(x); P \mid R_1 \mid R_2$.

(2) Input and Output Ordering. Two actions at the same session have the ordering, hence: $s_1?(x);s_1?(y);P \mid R_1 \not\approx s_1?(y);s_1?(x);P \mid R_1$ for inputs, $s_1!\langle v_1\rangle;s_1!\langle v_2\rangle;P \mid R_1 \not\approx s_1!\langle v_2\rangle;s_1!\langle v_1\rangle;P \mid R_1$ for outputs and $s_1!\langle v_1\rangle;s_2?(x_2);P \not\approx s_2?(x_2);s_1!\langle v_1\rangle;P$.

(3) Non-Local Semantics. In the literature [6, 7, 27], the asynchronous session types are modelled by the *two end-point queues* without distinction between input and output entries in queues. We call this semantics *non-local* since the output process directly puts the value into the input queue. The transition relation for non-local semantics is defined by replacing the output and selection rules in Figure 4 (see Appendix E.3 for full details) to:

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

We write the induced bisimilarity $\approx_2$. The same (in)equalities hold except permutation of outputs, e.g. $s_1!\langle v_1\rangle;s_2!\langle v_2\rangle;P \mid R_1 \mid R_2 \not\approx_2 s_2!\langle v_2\rangle;s_1!\langle v_1\rangle;P \mid R_1 \mid R_2$.

(4) Arrival Predicates. Let $Q = \mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ with $P_1 \not\approx P_2$. If the syntax does not include arrival predicates, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$. In the presence of the arrival predicate, we have $Q \mid s[\mathtt{i}:\emptyset] \mid \overline{s}[\mathtt{o}:v] \not\approx Q \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\emptyset]$ with $e = \mathtt{arrived}\ s$. However we have: $\mathtt{if}\ \mathtt{arrived}\ s\ \mathtt{then}\ P\ \mathtt{else}\ P \approx P$.

(5) Arrive Inspection Ordering. Let

$$P = \mu X.\mathtt{if}\ \mathtt{arrived}\ s_1\ \mathtt{then}\ s_1?(x).X\ \mathtt{else}\ \mathtt{if}\ \mathtt{arrived}\ s_2\ \mathtt{then}\ s_2?(x).X\ \mathtt{else}\ X$$
$$Q = \mu X.\mathtt{if}\ \mathtt{arrived}\ s_2\ \mathtt{then}\ s_2?(x).X\ \mathtt{else}\ \mathtt{if}\ \mathtt{arrived}\ s_1\ \mathtt{then}\ s_1?(x).X\ \mathtt{else}\ X$$

Then $P \mid R_1 \mid R_2 \approx Q \mid R_1 \mid R_2$. An inspection of arrival of messages using recursion as above makes no distinction between different orderings by which session endpoints are inspected. Later this equation plays a central role in the thread elimination later.

For more comparisons, see § 6.

4 Lauer-Needham Transform

In an early work [18], Lauer and Needham observed that a concurrent program may be written equivalently either in a thread-based programming style (with shared memory primitives) or in an event-based style (where a single-threaded event loop processes messages sequentially). Following this framework and using high-level asynchronous event primitives such as *selectors* [15, 19] for the event-based style, many studies compare these two programming styles, often focusing on performance of server architectures, e.g. [2, 15, 20, 32]. These studies and 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 thread-based web servers), to its *equivalent* event-based program, which treats concurrent services using a single threaded event-loop (as in event-based web servers). However the precise semantic effects of such a transformation nor the exact meaning of the associated "equivalence" has not been clarified.

We study the semantic effects of such a transformation using the asynchronous session bisimulation. Not only is this transform hence its semantics has practical significance, but also the semantic analysis of this transform demands the use of the key

elements of our asynchronous semantics, including asynchrony and events. We first introduce a formal mapping from a thread-based process to their event-based one, based on Lauer and Needham and suggested by recent studies on event-based programming [2, 20, 32]. Assuming a server process whose code creates fresh threads at each service invocation, the key idea of the transformation is to decompose this whole code into distinct smaller code segments each handling the part of the original code starting from a blocking action. As in the standard event-based programs, such a blocking action is represented as reception of a message (input or branching). Then a single global event-loop can treat each message arrival by processing the corresponding code segment combined with an environment, returning to inspect the content of event/message buffers.

We first stipulate a class of processes which we consider for our translation. Below $*a(x);P$ denotes an *input replication* abbreviating $\mu X.a(x).(P|X)$.

Definition 4.1 (Simple Server). A *server at a* is a replicated process $*a(x);P$ with a typing of form $a:\langle S\rangle$. A server is *simple* if it has the form $*a(w:S);P$ such that P contains neither parallel compositions ($|$) nor name hiding (ν), and each bound process variable in P is guarded. We usually consider a simple server in a localised form, with the 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. But it may not spawn further threads nor export a new shared channel, and, by guardedness, it always goes through a blocking action before recurring. Note a simple server does *not* involve accesses to internal shared state among threads: a simple server is stateless in this sense (as we shall see later, it is confluent), as found in static web-servers — those which only serve static pages.

Example 4.2 (A Simple Server). As an example of a simple server, consider:

$$P \;=\; *a(x);x?(y).x!\langle y+1\rangle;x?(z).x!\langle y+z\rangle;\mathbf{0} \mid a[\varepsilon]$$

This process has the session type $?(\mathsf{nat});!(\mathsf{nat})?(\mathsf{nat});!(\mathsf{nat})$ at a which can be read: *a process should first expect to receive* ? *a message of type* nat *and send* ! *it, then to receive (again* ?*) a* nat*, and finish by sending* ! *a result.*

A simple server creates an unbounded number of threads as it processes services concurrently. The thread elimination transforms this process into an equivalent one which uses a *single* thread for processing these unbounded number of services. Given a simple server $*a(w:S);P \mid a[\varepsilon]$, its translation, which we call *Lauer-Needham transform* or *LN-transform* for short, is written $\mathscr{LN}[\![*a(w:S);P \mid a[\varepsilon]]\!]$. The formal mapping is given in Appendix G: here we outline the key constructions of the LN-transformed $\mathscr{LN}[\![*a(w:S);P \mid a[\varepsilon]]\!]$ and illustrate the key ideas through examples.

First, $\mathscr{LN}[\![*a(w:S);P]\!]$ consists of the following key elements:

1. An *event loop* $\mathsf{Loop}\langle o,q\rangle$ repeats the standard *event-loop*, inspecting its *selector queue* named q (see below), *selects* a message from a message arrived and processes it by sending it to the corresponding *code block* (see below).
2. A *selector queue* $q\langle a,c_0\rangle$ whose initial element is $\langle a,c_0\rangle$. This data says: "if a message comes at a, jump to the code block (CPS procedure) whose subject is c_0".

3. A collection of *code blocks* $\mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle$, CPS procedures handling incoming messages. A code block originates from a *blocking subterm* of $a(w);P$, i.e. a subterm starting from an input or a branching, see Example 4.3 below.

The process uses a standard "select" primitive represented as a process, called *selector* [15]. It stores *a collection of session channels*, with each channel associated with an environment, binding variables to values. It then picks up one of them at which a message arrives, receives that message via that channel and has it be processed by the corresponding code block (which may alter the environment). Finally it stores the session and the associated environment back in the collection, and moves to the next iteration. Since a selector should handle channels of different types, it uses the `typecase` construct from [15]. $\texttt{typecase}\ k\ \texttt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I}$ takes a session endpoint k and a list of cases $(x_i:T_i)$, each binding the free variable x_i of type pattern T_i in P_i. The reduction of `typecase` is defined as:

$$\texttt{typecase}\ s\ \texttt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I} \mid s\,[S] \longrightarrow P_i\{s/x_i\} \mid s\,[S]$$

where there exists $i \in I$ such that $(\forall j < i.T_j \not\leq S \wedge T_i \leq S)$. We extend the calculus with the selector operations, with the following reduction semantics.

$$\begin{array}{l}
\texttt{new selector}\ r\ \texttt{in}\ P \longrightarrow (\nu\, r)(P \mid \mathsf{sel}\langle r, \varepsilon\rangle) \qquad \texttt{register}\langle s', r\rangle; P \mid \mathsf{sel}\langle r, \vec{s}\rangle \longrightarrow P \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \\
\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \\
\qquad\qquad \longrightarrow P_i\{s'/x_i\} \mid \mathsf{sel}\langle r, \vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \quad (\vec{h} \neq \varepsilon) \\
\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[\texttt{i}:\varepsilon] \\
\qquad\qquad \longrightarrow \texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i:T_i):P_i\}_{i\in I} \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \mid s'\,[\texttt{i}:\varepsilon]
\end{array}$$

where in the third line S and T_i satisfies the condition for `typecase` in the reduction rule. We also include the structural rules with garbage collection rules for queues as $(\nu\, r)\mathsf{sel}\langle r, \varepsilon\rangle \equiv \mathbf{0}$. Operator $\texttt{new selector}\ r\ \texttt{in}\ P$ (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r and with the empty queue ε. $\texttt{register}\langle s', r\rangle; P$ registers a session channel s

Operator $\texttt{register}\langle s', r\rangle; P$ registers a session channel s to r, adding s' to the original queue $\vec{s}$. The next `let` retrieves a registered session and checks the availability to test if an event has been triggered. If so, find the match of the type of s' among $\{T_i\}$ and select P_i; if not, the next session is tested. As shown in [15], these primitives are encodable in the original calculus augmented with `typecase`. For a detailed encoding, see Appendix F. The bisimulations and their properties remain unchanged.

Example 4.3 (Lauer-Needham Transform). Recall the simple server in Example 4.2. We extract the blocking subterms from this process as follows.

Blocking Process	Type at Blocking Prefix
$a(x).x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle;\mathbf{0}$	$\langle ?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat})\rangle$
$x?(y).x!\langle y+1\rangle x?(z).x!\langle y+z\rangle;\mathbf{0}$	$?(\mathsf{nat}); !(\mathsf{nat}); ?(\mathsf{nat}); !(\mathsf{nat})$
$x?(z).x!\langle y+z\rangle;\mathbf{0}$	$?(\mathsf{nat}); !(\mathsf{nat})$

These blocking processes are translated into *code block processes* or *code blocks*, denoted CodeBlocks, given as:

$$\begin{array}{l}
*c_0(y); a(x).\texttt{update}(y,x,x); \texttt{register}\ \langle sel, x, y, c_1\rangle; \overline{o}\ \mid \\
*c_1(x,y); x?(z); \texttt{update}(y,z,z); x!\langle [\![z]\!]_y + 1\rangle; \texttt{register}\ \langle sel, x, y, c_2\rangle; \overline{o}\ \mid \\
*c_2(x,y); x?(z'); \texttt{update}(y,z',z'); x!\langle [\![z]\!]_y + [\![z']\!]_y\rangle; \overline{o}
\end{array}$$

which is used for processing each message. Above, the operation $\texttt{update}(y,x,x)$; updates an environment, while `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 centre of the event processing, the main event-loop denoted Loop, is given as:

$$\begin{array}{l}\mathsf{Loop} = *o.\texttt{let}\ (x,y,z) = \texttt{select from}\ \mathit{sel}\ \texttt{in typecase}\ x\ \texttt{of}\ \{\\ \qquad\qquad \langle ?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat})\rangle\ :\ \texttt{new}\ y : \texttt{env}\ \texttt{in}\ \overline{z}(y)\\ \qquad\qquad ?(\mathsf{nat});!(\mathsf{nat});?(\mathsf{nat});!(\mathsf{nat})\ :\ \overline{z}(x,y)\\ \qquad\qquad ?(\mathsf{nat});!(\mathsf{nat})\ :\ \overline{z}(x,y)\\ \qquad \}\end{array}$$

Above `select from` *sel* `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 dispatch the task to each code block.

For the formal details, see Appendix G. Because a simple server does not allow, by construction, its internal shares state to be accessed by the threads it spawns, it is currently stateless.[2], we observe:

Lemma 4.4. *$*a(w:S);R \mid a\,[\varepsilon]$ is confluent.*

On the other hand, the transformed process is essentially a sequential process. Hence we can also establish:

Lemma 4.5. *$\mathscr{LN}[\![*a(w:S);P \mid a\,[\varepsilon]]\!]$ is confluent.*

Using these results, we can establish the fundamental result, which crucially relies on asynchronous nature of our bisimulations, noting that the original simple server and its translation generally have completely different timings at which the session input/branching subjects (prefixes) become active: in the former they are made active in parallel for all threads, while in the latter they are made active only sequentially.

Lemma 4.6. *Let $R = \mathsf{Loop} \mid \mathsf{CodeBlocks} \mid a[\vec{s}] \mid \prod_{m\in I} s_m\,[\mathtt{i}:\vec{h_{im}}, \mathtt{o}:\vec{h_{om}}]$. Assume*

$$\begin{array}{l}P_1 = r\langle (s_1,\xi_1,c_1),\ldots,(s_i,\xi_i,c_i),\ldots,(s_j,\xi_j,c_j),\ldots,(s_n,\xi_n,c_n)\rangle\\ P_2 = r\langle (s_1,\xi_1,c_1),\ldots,(s_j,\xi_j,c_j),\ldots,(s_i,\xi_i,c_i),\ldots,(s_n,\xi_n,c_n)\rangle\end{array}$$

Then $(\nu\ \vec{c}or)(P_1 \mid R) \approx (\nu\ \vec{c}or)(P_2 \mid R)$.

The above lemma substantiates a generalisation of the observations behind Example 3.3(5), which distills a nature of event-driven programming. With this lemma, we can permute the sessions in a selector queue, still having the same behaviours (this permutation is possible w.r.t. bisimilarity because of the stateless nature of a simple server). Selector's behaviour ties together threaded and event programs because there is no difference between which event (resp. thread) is selected to be executed first.

[2] The transform easily extends to the situation where threads share state, though its behavioural justification takes a different form.

Theorem 4.7 (Semantic Preservation). *Let* $*a(w:S);R \mid a[\varepsilon]$ *be a simple server. Then* $*a(w:S);P \mid a[\varepsilon] \approx \mathscr{LN}[\![a(w:S);P \mid a[\varepsilon]]\!]$.

The proof of the above theorem uses a determinate upto-expansion relation, cf. Definition 3.10 and Lemma 3.11. We construct the target relation so that it contains each process pair that has all the parallel processes on a blocking prefix for the threaded server and starts from the Loop process for the thread eliminated process. We show the conditions needed Definition 3.10 by using Lemmas 4.4 and 4.5. We conclude the proof through Lemma 3.11. For details of the proof, see Appendix G.

5 Performance Evaluation of the LN-Transform

This section presents a benchmark evaluation of the type-directed transformation from multithreaded to thread-eliminated (i.e. event-driven) programs by the LN-transform (§ 4). The benchmark compares the performance of implementations of the Server processes from Examples 4.2 and 4.3 under load from a varying number of concurrent clients, ranging up to 1000. As discussed earlier, event-driven concurrency is typically regarded as trading performance for more complex implementation; the present benchmark examines these aspects from each side of the trade-off. First, the results show that the LN-transformed Server exhibits the performance and scalability improvements expected of thread-elimination. The second and key point, however, is that the LN-transformation, with formal justification by asynchronous session bisimilarity (Theorem 4.7), ensures the thread-eliminated Server preserves equivalent behaviour to the original. This benchmark thus demonstrates how the session-oriented behavioural equivalence theory presented in this paper can be directly applied to the development of practical tools for real-world concurrent programming.

Benchmark Programs. The implementation and execution of the session-typed benchmark applications uses SJ (Session Java) [16] with extensions for event-driven session programming [15]. The session type implemented by the multithreaded Server (MT) is a slightly modified version of the type from Example 4.2 (to allow variation of the message size), which written in SJ syntax is:

```
?(byte[]).!<byte[]>.?(byte[]).!<byte[]>
```

The MT Server spawns a new thread to execute a session of this type for each client that connects. The LN-transformed Server (TE) uses a single-threaded central event-loop to dispatch the input events of the above type across concurrent sessions one-by-one, following Example 4.3. The same Client program, which implements the dual of the above type, is used to interact with both the MT and TE Servers.

Methodology and Environment. The benchmark measures Server throughput in terms of the number of Client sessions completed per millisecond. The benchmark parameters are the message size (100 Bytes and 1 KB), and the number (10, 100, 300, 500, 700 and 900) of concurrent Clients, which simply request and execute repeated sessions with the Server. After the execution of the benchmark configuration has stabilised, a measurement is taken by recording the number of sessions completed by the Server within a 30 s measurement window. For each parameter combination, we take measurements from three windows to each Server instance, and repeat the whole benchmark 20 times.

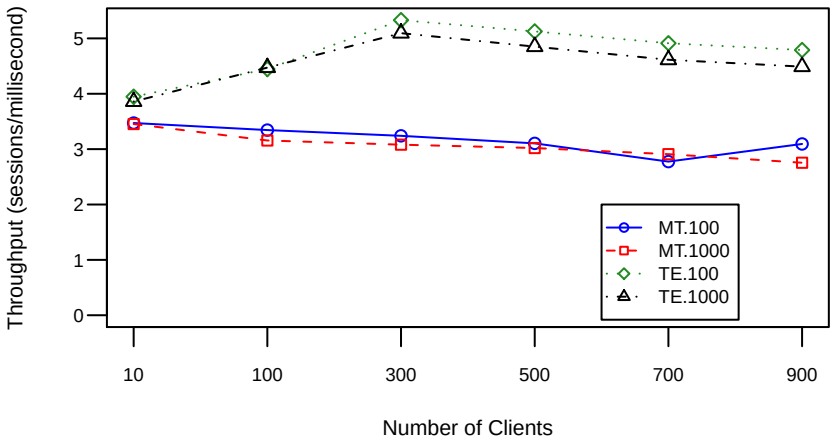

Fig. 6. Throughput of multithreaded (MT) and LN-transformed (TE) SJ Servers under increasing client loads.

The benchmark is conducted in the following cluster environment: each node is a Sun Fire x4100 with two 64-bit Opteron 275 processors at 2 GHz and 8 GB of RAM, running 64-bit Mandrakelinux 10.2 (kernel 2.6.11) and connected via gigabit Ethernet. Latency between each node is measured to be 0.5 ms on average (ping 64 bytes). The benchmark applications are compiled and executed using the Sun Java SE compiler and Runtime versions 1.6.0.

Results. Figure 6 gives the mean throughput for the multithreaded (MT) and thread-eliminated (TE) Server implementations for the increasing number of Clients. In all cases, as expected, the TE Server exhibits higher throughput than the MT server. We observe that the throughput of the MT Server decreases steadily due to the thread scheduling overhead. For the TE Server, there is first an increase in throughput, until the saturation point for the Server performance is reached, and then a decline as the number of Clients increases, due to the cost of the event selector handling more Client connections. We also observe higher throughput for the smaller message size: this is because more messages of the smaller size can be transmitted, and thus more sessions can be completed, in the same period of time. Further micro and macrobenchmarks comparing session-typed multithreaded and event-driven SJ applications, and also their untyped counterparts (standard Java threads and Java NIO) as base cases, can be found at [30].

6 Discussions

Comparisons with Asynchronous/Synchronous Calculi. We briefly compared the present asynchronous bisimulation to related ones in Example 3.3 in § 3. Below we report more comprehensive comparisons, clarifying the relationship between (1) the session-typed asynchronous π-calculus [10] without queues ($\approx_a$, the asynchronous version of the labelled transition relation for the asynchronous π-calculus), (2) the session-typed synchronous π-calculus [11, 31] without queues ($\approx_s$), (3) the asynchronous session π-calculus with two end-point queues without IO queues [6, 7, 27] ($\approx_2$), see Ex-

ample 3.3; and (4) the asynchronous session π-calculus with two end-point IO-queues ($\approx$), i.e. the one developed in this paper. See Appendix E for the full definitions.

The following figure summarises distinguishing examples. Non-Blocking Input/Output means inputs/outputs on different channels, while the Input/Output Order-Preserving means that the messages will be received/delivered preserving the order. The final table explains whether Lemma 3.4 (1) (input advance) or (2) (output delay) is satisfied or not. If not, we place a counterexample.

	Non-Blocking Input	Non-Blocking Output
(1)	$s_1?(x);s_2?(y);P \approx_a s_2?(y);s_1?(x);P$	$\overline{s_1}\langle v\rangle \mid \overline{s_2}\langle w\rangle \mid P \approx_a \overline{s_1}\langle w\rangle \mid \overline{s_2}\langle v\rangle \mid P$
(2)	$s_1?(x);s_2?(y);P \not\approx_s s_2?(y);s_1?(x);P$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \not\approx_s s_2!\langle w\rangle;s_1!\langle v\rangle;P$
(3)	$s_1?(x);s_2?(y);P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \approx_2$ $s_2?(y);s_1?(x);P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon] \not\approx_2$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid s_1[\varepsilon] \mid s_2[\varepsilon]$
(4)	$s_1?(x);s_2?(y);P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \approx$ $s_2?(y);s_1?(x);P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \approx$ $s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

	Input Order-Preserving	Output Order-Preserving
(1)	$s?(x);s?(y);P \approx_a s?(y);s?(x);P$	$\overline{s}\langle v\rangle \mid \overline{s}\langle w\rangle \mid P \approx_a \overline{s}\langle w\rangle \mid \overline{s}\langle v\rangle \mid P$
(2)	$s?(x);s?(y);P \not\approx_s s?(y);s?(x);P$	$s!\langle v\rangle;s!\langle w\rangle;P \not\approx_s s!\langle w\rangle;s!\langle v\rangle;P$
(3)	$s?(x);s?(y);P \mid s[\varepsilon] \not\approx_2$ $s?(x);s?(y);s?(x);P \mid s[\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid s[\varepsilon] \not\approx_b$ $s!\langle w\rangle;s!\langle v\rangle;P \mid s[\varepsilon]$
(4)	$s?(x);s?(y);P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx$ $s?(x);s?(y);P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$	$s!\langle v\rangle;s!\langle w\rangle;P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \not\approx$ $s!\langle w\rangle;s!\langle v\rangle;P \mid [s_1,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid [s_2,\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$

	Lemma 3.4 (1)	Lemma 3.4 (2)
(1)	yes	yes
(2)	$(\nu\ s)(s!\langle v\rangle;s'?(x);\mathbf{0} \mid s?(x);\mathbf{0})$	$(\nu\ s)(s!\langle v\rangle;s'!\langle v'\rangle;\mathbf{0} \mid s'?(x);\mathbf{0})$
(3)	yes	$s!\langle v\rangle;s'?(x);\mathbf{0} \mid s'[v']$
(4)	yes	yes

Another technical interest is the effects of the `arrived` predicate on these combinations, which we carry out below. For precise comparisons, we define the synchronous and asynchronous π-calculi augmented with the `arrived` predicate and local buffers. For the asynchronous π-calculus, we add $a[\vec{h}]$ and `arrived` a in the syntax, and define the following rules for input and outputs.

$$\overline{a}\langle v\rangle \xrightarrow{\overline{a}\langle v\rangle} \mathbf{0} \quad a[\vec{h}] \xrightarrow{a\langle h\rangle} a[\vec{h}\cdot h] \quad a?(x).P \mid a[\vec{h_1}\cdot h_i\cdot\vec{h_2}] \longrightarrow P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}]$$

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

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

The above definition precludes the order preservation as the property of transport, but still keeps the non-blocking property as in the asynchronous π-calculus. The synchronous version is similarly defined by setting the buffer size to be one. The non-local version is defined just by adding `arrived` predicate.

Figure 7 summarises the results which incorporate the `arrived` predicate. Interestingly in all of the calculi (1–4), the same examples as in Example 3.3(4), which separate semantics with/without the `arrived`, are effective. The IO queues provide non-blocking inputs and outputs, while preserving the input/output ordering, which distinguishes the present framework from other known semantics.

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

Fig. 7. Arrive inspection behaviour in synchronous/asynchronous calculi.

The formal definitions and proofs for the above results can be found in Appendix E. As a whole, 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 `arrived` predicates, which enables processes to observe the effects of fine-grained synchronisations.

Related Work. Some of the key proof methods of our work draw their ideas from [28], which study an extension of the confluence theory on the π-calculus. They apply the theory to reason about the correctness of the distributed protocol which can be represented by constructing a collection of confluent processes. Our work differs in that we investigate the effect of asynchronous IO queues and its relationship to confluence.

The work [3] examines expressiveness of various messaging mediums by adding message bags (no ordering), stacks (LIFO policy) and message queues (FIFO policy) in the asynchronous π-calculus [10]. They show that the calculus with the message bags is encodable into the asynchronous π-calculus, but it is impossible to encode the message queues and stacks. Neither the effects of locality, queues, and typed transitions are studied. Further neither of [3, 28] treats event-based programming, including such examples as thread elimination nor performance analysis.

Programming constructs that can test the presence of actions or events are studied in the context of the Linda language [5] and CSP [21, 22]. The work [5] measures expressive powers between three variants of asynchronous Linda-like calculi, with a construct for inspecting the output in the tuple space, which is reminiscent of the *inp* predicate of Linda. The first calculus (called *instantaneous*) corresponds to (1) [10], the second one (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. It shows that the instantaneous and ordered calculi are Turing powerful, while the unordered is not. The work [21] studies CSP with a construct that checks if a parallel process is able to perform an output action on a given channel. It studies operational and denotational semantics, demonstrating a significance to investigate event primitives using process calculi. A subsequent work [22] studies the expressiveness of its variants focussing on the full abstraction theorem of the trace equivalence. Due to the difference of the base calculi and the aims of the primitives, direct comparisons are difficult: for example, our calculi (1,2,3,4) are Turing powerful and we aim to examine properties and applications of the typed bisimilarity characterised by buffered sessions: on the other hand, the focus of [5] is a tuple space where our input/output order pre-

serving examples (which treat different objects with the same session channel) cannot be naturally (and efficiently) defined. The same point applies for [21, 22]. As another difference, the nature of localities has not been considered either in [5, 21, 22] since no notion of a local or remote tuple or environment is defined. Further, neither large applications which include these constructs (§ 4), the equivalences as we treated, nor the performance analysis of the proposed primitives had been discussed in [5, 21, 22].

Using the confluence and determinacy guaranteed by session types, and through observations on the semantics of the arrive predicate, we have demonstrated that the theory is applicable, through the verification of the correctness of the Lauer-Needham transform. This well-known transform claims that threads and events are dual to each other. Our LN-transform and the asynchronous, buffered bisimulation provide a formal framework and reasoning mechanisms for the conversion from the former to latter and for establishing their equivalence. The asynchronous nature realised through IO message queues provides a precise analysis of local and eventful behaviours, found in major distributed transports such as TCP. The benchmark results from high-performance clusters justify the effect of the type and semantic preserving LN-transformation.

Some preceding works approach the Lauer-Needham duality by combining multithreaded and event-driven programming to obtain benefits from both categories. The work [20] implements both multithreaded and event-driven components at the application level. A trace over blocking system calls is inferred from the threaded code and the scheduler invokes the user-supplied event handler when event points are reached in the trace. Scala [8] offers both thread-blocking receive operations and actor-based event handlers as a library. Heuristics are used to determine whether a thread has become blocked on some external operation to prevent blocking other actors. Although these studies offer improved support for user-level threads, event-based systems currently retain the edge in terms of performance and scalability for highly concurrent applications such as Web servers [17]. These preceding studies do not investigate either semantic foundations or the formal correctness of their approaches.

In contrast, the present paper offers a semantic basis for asynchronous session communications with event primitives, which can justify the correct implementation of practical communications-based applications, as we have seen in §4. It also applies the behavioural theory to demonstrate a benefit of structured and type-safe transformation based on session types. It is an interesting future work to use our bisimulation theory in the higher-order setting [27] as well as for analysing more complex programs where threads and events co-exist.

References

1. M. Abadi, L. Cardelli, B. C. Pierce, and G. D. Plotkin. Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268, 1991.
2. A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming. In *In Proceedings of the 2002 Usenix ATC*, 2002.
3. R. Beauxis, C. Palamidessi, and F. D. Valencia. On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 473–492. Springer, 2008.
4. M. Berger, K. Honda, and N. Yoshida. Sequentiality and the π-calculus. In *Proc. TLCA'01*, volume 2044 of *LNCS*, pages 29–45, 2001.

5. N. Busi, R. Gorrieri, and G. Zavattaro. Comparing three semantics for linda-like languages. *Theor. Comput. Sci.*, 240(1):49–90, 2000.
6. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.
7. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 2009.
8. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.
9. M. Hennessy. *A Distributed Pi-Calculus*. CUP, 2007.
10. K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.
11. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
12. K. Honda and N. Yoshida. On reduction-based process semantics. *TCS*, 151(2):437–486, 1995.
13. K. Honda and N. Yoshida. A uniform type structure for secure information flow. *TOPLAS*, 29(6), 2007.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
15. R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
16. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer, 2008.
17. M. Krohn. Building secure high-performance web services with OKWS. In *ATEC'04*, pages 15–15. USENIX Association, 2004.
18. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.
19. D. Lea. *Scalable IO in Java*. `http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf`, November 2003.
20. P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199, 2007.
21. G. Lowe. Extending csp with tests for availability. *Procedings of Communicating Process Architectures (CPA 2009)*, 2009.
22. G. Lowe. Models for CSP with availability information. *Pre-proceeding for Express'10*, 2010.
23. S. Microsystems Inc. New IO APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`.
24. R. Milner. *Communication and Concurrency*. Prentics Hall, 1989.
25. R. Milner. The polyadic π-calculus: A tutorial. In *Proceedings of the International Summer School on Logic Algebra of Specification*. Marktoberdorf, 1992.
26. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1989.
27. D. Mostrous and N. Yoshida. Session-based communication optimisation for higher-order mobile processes. In *TLCA'09*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.
28. A. Philippou and D. Walker. On confluence in the pi-calculus. In *ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 314–324. Springer, 1997.
29. On-line Appendix of this paper. `http://www.doc.ic.ac.uk/~dk208/semantics.html`.
30. SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

31. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
32. M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP'01*, pages 230–243. ACM Press, 2001.

Table of Contents

$$
\begin{array}{rll}
P \equiv Q \quad \text{if } P =_\alpha Q & & (\alpha\text{-renaming}) \\
P \mid \mathbf{0} \equiv P & & (\text{Idempotence}) \\
P \mid Q \equiv Q \mid P & & (\text{Commutativity}) \\
(P \mid P') \mid P'' \equiv P \mid (P' \mid P'') & & (\text{Associativity}) \\
(\nu\, a{:}\langle S\rangle)\, a[\varepsilon] \equiv \mathbf{0} & & (\text{Shared channels}) \\
(\nu\, a{:}\langle S\rangle)\, P \mid Q \equiv (\nu\, a{:}\langle S\rangle)\,(P \mid Q) & (a \notin \mathrm{fn}(Q)) & \\
(\nu\, s)\, \mathbf{0} \equiv \mathbf{0} & & (\text{Session channels}) \\
(\nu\, s)\,(s : [\varepsilon] \mid \overline{s} : [\varepsilon]) \equiv \mathbf{0} & & (\text{Session queues}) \\
(\nu\, s)\, P \mid Q \equiv (\nu\, s)\,(P \mid Q) & (s \notin \mathrm{fn}(Q)) & \\
\mu X.P \equiv P\{\mu X.P/X\} & &
\end{array}
$$

Fig. 8. Structural congruence.

A Appendix for Section 2

We give the definitions that were omitted from § 2 including the type case construct.

A.1 Structural Congruence

The notion of bound and free identifiers is extended to cover the subject and objects of `arrived` u, `arrived` k, `arrived` k h, `typecase` k `of` $\{(x_i : T_i)\, P_i\}_{i\in I}$, $\overline{a}\,\langle s\rangle$, $a\,[\vec{s}]$, and $s\,[S, \mathtt{i}:\vec{h}, \mathtt{o}:\vec{h}']$ in the expected way. We write $\mathrm{fn}(P)$ for the set of names that have a free occurrence in P. Then structural congruence is the smallest congruence on processes generated by the following rules in Figure 8.[3]

A.2 Reduction

Reduction Relation The binary single-step reduction relation, $\longrightarrow$ is the smallest relation on closed terms generated by the rules in Figure 3 together with those in Figure 9.

B Appendix to Section 2.3

B.1 Subtyping

If P has a session channel s typed by S, P can interact at *s at most* as S (e.g. if S has shape $\oplus\{l_1 : S_1, l_2 : S_2, l_3 : S_3\}$ then P may send l_1 or l_3, but not a label different from $\{l_1, l_2, l_3\}$). Hence, $S \leq S'$ means that a process with a session typed by S is more composable with a peer process than one by S'. Composability also characterises the subtyping on shared channel types. Formally the subtyping relation is defined for the set $\mathscr{T}$ of all closed and contractive types as follows: T is a subtype of T', written $T \leq T'$,

[3] For the recursion, it would be more natural to use a demand-driven input-guarded reduction for the LN-transformation. The choice does not affect the developments of this paper.

$$
\begin{array}{lrcll}
\text{[Request1]} & \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) & (s \notin \mathrm{fn}(P)) \\
\text{[Request2]} & a\,[\vec{s}] \mid \overline{a}\,\langle s\rangle & \longrightarrow & a\,[\vec{s}\cdot s] & \\
\text{[Accept]} & 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}] & \\
\text{[Send]} & s!\langle v\rangle;P \mid s\,[!(T);S, \mathtt{o}:\vec{h}] & \longrightarrow & P \mid s\,[S, \mathtt{o}:\vec{h}\cdot v] & \\
\text{[Receive]} & s?(x).P \mid s\,[?(T);S, \mathtt{i}:v\cdot\vec{h}] & \longrightarrow & P\{v/x\} \mid s\,[S, \mathtt{i}:\vec{h}] & \\
\text{[Sel], [Bra]} & s \triangleleft l_i;P \mid s\,[\oplus\{l_i:S_i\}_{i\in I}, \mathtt{o}:\vec{h}] & \longrightarrow & P \mid s\,[S_i, \mathtt{o}:\vec{h}\cdot l_i] & (i \in I) \\
& s \triangleright \{l_j:P_j\}_{j\in J} \mid s\,[\&\{l_i:S_i\}_{i\in I}, \mathtt{i}:l_i\cdot\vec{h}] & \longrightarrow & P_i \mid s\,[S_i, \mathtt{i}:\vec{h}] & (i \in I \subseteq J) \\
\text{[Comm]} & 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] & \\
\text{[Arriv-req]} & E[\mathtt{arrived}\ a] \mid a\,[\vec{s}] & \longrightarrow & E[\mathtt{b}] \mid a\,[\vec{s}] & (|\vec{s}| \geq 1) \searrow_{\mathtt{b}} \\
\text{[Arriv-ses]} & E[\mathtt{arrived}\ s] \mid s\,[\mathtt{i}:\vec{h}] & \longrightarrow & E[\mathtt{b}] \mid s\,[\mathtt{i}:\vec{h}] & (|\vec{h}| \geq 1) \searrow_{\mathtt{b}} \\
\text{[Arriv-msg]} & E[\mathtt{arrived}\ s\,h] \mid s\,[\mathtt{i}:\vec{h}] & \longrightarrow & E[\mathtt{b}] \mid s\,[\mathtt{i}:\vec{h}] & (\vec{h} = h\cdot\vec{h}') \searrow_{\mathtt{b}} \\
\text{[Typecase]} & & & & \\
& \mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:T_i)\,P_i\}_{i\in I} \mid s\,[S] & \longrightarrow & P_i\{s/x_i\} \mid s\,[S] & \exists i \in I.(\forall j < i.T_j \not\leq S \wedge T_i \leq S)
\end{array}
$$

$$
\text{[Unfold]}\ P \mid s\,[S\{\mu\mathsf{X}.S/\mathsf{X}\}, \mathtt{i}:\vec{h}_1, \mathtt{o}:\vec{h}'_1] \longrightarrow P' \mid s\,[S', \mathtt{i}:\vec{h}_2, \mathtt{o}:\vec{h}'_2] \implies P \mid s\,[\mu\mathsf{X}.S, \mathtt{i}:\vec{h}_1, \mathtt{o}:\vec{h}'_1] \longrightarrow P' \mid s\,[S', \mathtt{i}:\vec{h}_2, \mathtt{o}:\vec{h}'_2]
$$

$$
\begin{array}{lrcl}
\text{[Eval]} & e \longrightarrow e' & \implies & E[e] \longrightarrow E[e'] \\
\text{[Chan]} & P \longrightarrow P' & \implies & (\nu\, a{:}\langle S\rangle)\,P \longrightarrow (\nu\, a{:}\langle S\rangle)\,P' \\
\text{[Sess]} & P \longrightarrow P' & \implies & (\nu\, s)\,P \longrightarrow (\nu\, s)\,P' \\
\text{[If-true]} & & & \mathtt{if}\ \mathtt{tt}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow P \\
\text{[If-false]} & & & \mathtt{if}\ \mathtt{ff}\ \mathtt{then}\ P\ \mathtt{else}\ Q \longrightarrow Q \\
\text{[Par]} & P \longrightarrow P' & \implies & P \mid Q \longrightarrow P' \mid Q \\
\text{[Struct]} & P \equiv P' \longrightarrow Q' \equiv Q & \implies & P \longrightarrow Q
\end{array}
$$

Fig. 9. The reduction rules § 2.2.

if (T, T') is in the largest fixed point of the monotone function $\mathscr{F} : \mathscr{P}(\mathscr{T} \times \mathscr{T}) \to \mathscr{P}(\mathscr{T} \times \mathscr{T})$, such that $\mathscr{F}(\mathscr{R})$ for each $\mathscr{R} \subset \mathscr{T} \times \mathscr{T}$ is given as follows.

$$
\begin{array}{l}
\{(\mathsf{bool}, \mathsf{bool}), (\mathsf{nat}, \mathsf{nat})\} \cup \{(\mathtt{o}\langle S\rangle, \mathtt{o}\langle S'\rangle) \mid (S, S'), (S', S) \in \mathscr{R}\} \\
\cup\ \{(\mathtt{i}\langle S\rangle, \mathtt{i}\langle S'\rangle) \mid (S, S'), (S', S) \in \mathscr{R}\} \\
\cup\ \{(\mu\mathsf{X}.U, U') \mid (U\{\mu\mathsf{X}.U/\mathsf{X}\}, U') \in \mathscr{R}\} \cup \{(U, \mu\mathsf{X}.U') \mid (U, U'\{\mu\mathsf{X}.U'/\mathsf{X}\}) \in \mathscr{R}\} \\
\cup\ \{(!(T_1);S'_1, !(T_2);S'_2) \mid (T_2, T_1), (S'_1, S'_2) \in \mathscr{R}\} \cup \{(?(T_1);S'_1, ?(T_2);S'_2) \mid (T_1, T_2), (S'_1, S'_2) \in \mathscr{R}\} \\
\cup\ \{(\oplus\{l_i:S_i\}_{i\in I}, \oplus\{l_j:S'_j\}_{j\in J}) \mid \forall i \in I \subseteq J.(S_i, S'_i) \in \mathscr{R}\} \\
\cup\ \{(\&\{l_i:S_i\}_{i\in I}, \&\{l_j:S'_j\}_{j\in J}) \mid \forall j \in J \subseteq I.(S_j, S'_j) \in \mathscr{R}\} \\
\cup\ \{(\mu\mathsf{X}.S, S') \mid (S\{\mu\mathsf{X}.S/\mathsf{X}\}, S') \in \mathscr{R}\} \cup \{(S, \mu\mathsf{X}.S') \mid (S, S'\{\mu\mathsf{X}.S'/\mathsf{X}\}) \in \mathscr{R}\} \\
\cup\ \{(\{S_i\}_{i\in I}, \{S'_j\}_{j\in J}) \mid \neg(|I| = |J| = 1), \forall j \in J, \exists i \in I.(S_i, S'_j) \in \mathscr{R}\}
\end{array}
$$

Line 1 is standard ($\langle S\rangle$ is invariant at S since it logically contains both S and $\overline{S}$). Line 2/6 are the standard rule for recursion. In Line 3, the linear output (resp. input) is contravariant (resp. covariant) on its carried types following [27]. In Line 4, the selection is co-variant since if a process may send more labels, it is less composable with its peer. Dually for branching in Line 5. Finally, the ordering of the set types says that if each element of the set type $\{S'_j\}_{j\in J}$ has its subtype in $\{S_i\}_{i\in I}$, the former is less composable by the latter. The condition $\neg(|I| = |J| = 1)$ avoids the case $\{S_i\}_{i\in I} = S, \{S'_j\}_{j\in J} = S'$, which makes the relation universal.

We now clarify the semantics of $\leq$ using *duality*. The dual of S, denoted $\overline{S}$, is defined in the standard way: $\overline{!(T);S} = ?(T);\overline{S}$, $\overline{?(T);S} = !(T);\overline{S}$, $\overline{\mu \mathsf{X}.S} = \mu \mathsf{X}.\overline{S}$, $\overline{\mathsf{X}} = \mathsf{X}$, $\overline{\oplus\{l_i : S_i\}_{i\in I}} = \&\{l_i : \overline{S_i}\}_{i\in I}$ $\overline{\&\{l_i : S_i\}_{i\in I}} = \oplus\{l_i : \overline{S_i}\}_{i\in I}$ and $\overline{\mathtt{end}} = \mathtt{end}$.

B.2 Program Typing

We first define a typing system for programs (§ 2.1). Program typing can be considered a static typing phase performed by a compiler, on user-level code before execution. Program typing uses two environments:

$$\Gamma ::= \emptyset \mid \Gamma \cdot u : U \mid \Gamma \cdot X : \Delta \qquad \Sigma ::= \emptyset \mid \Sigma \cdot a \mid \Sigma \cdot k : \{S_i\}_{i\in I}$$

Γ is called *shared environment*, which maps shared channels and process variables to shared channel types and linear environment respectively; Σ is called *linear environment* mapping session channels to set types (writing $k : S$ for $k : \{S\}$) and recording shared channels for acceptor's input queues. $\Sigma \cdot a$ means $a \notin \mathrm{dom}(\Sigma)$ and similarly for others. Shared channel a is recorded in Σ to ensure that one and only one queue for a exists. Subtyping is extended to environments by $\Sigma \leq \Sigma'$ iff $\mathrm{dom}(\Sigma) = \mathrm{dom}(\Sigma')$ and $\forall k \in \mathrm{dom}(\Sigma).\Sigma(k) \leq \Sigma'(k)$. The typing judgements for a process and an expression are given as:

$$\Gamma \vdash P \triangleright \Sigma \qquad \Gamma, \Sigma \vdash e : T$$

where the program P, under shared environment Γ, features the channel usage specified by linear environment Σ; and similarly for the expression. The judgement can be shortened to, e.g. $\Gamma \vdash e : T$ if Σ is not required.

Figure 10 presents the typing rules for programs. The first four rules are the standard expression typing (note $u : \mathtt{i}\langle S\rangle$ and $u : \mathtt{o}\langle S\rangle$ allows both accept and request respectively via u by (Shared), while $u : \mathtt{i}\langle S\rangle$ allows request via u by (Shared')). The next four rules are for the arrivals. Rule (Areq) is for a session request arrival. Rule (Asess) is for an in-session message arrival. Rule (Amsg) checks the message arrival and its value. Similarly (Alab) checks a branch label.

The remaining rules are standard. Rule (Subs) is the standard subsumption. Rule (Acc) (resp. Rule (Req)) says that the session following an accept (resp. request) should conform to a declared shared channel type. (Send) and (Recv) are for sending and receiving values. (Send) is standard apart from using the extended expression judgement to handle occurrences of the arrival predicates within e: the relevant type prefix $!(S)$ for the output is composed with T in the conclusion's session environment. (Recv) is a dual input rule. (SSend) and (SRecv) are the standard rules for delegation of a session and its dual [27], observing (SSend) says that $k' : T$ does not appear in P, symmetrically to (SRecv) which uses the channels in P. (Select) and (Branching), identical with [27], are the rules for selection and branching. (If) rule handles the conditional expression.

(Chan) records the $a : \langle S\rangle$ from the shared environment after checking the presence of a (in effect unique) queue at a. (Par) prevents multiple queues for the same shared channel and processes with the same session channels from being composed. (Nil), (Rec) and (PVar) are standard from [11]. (Queue) records the presence of an empty shared queue in a linear environment. "Σ $\mathtt{end}$ only" means Σ has the form $s_1 : \mathtt{end}, \ldots, s_n : \mathtt{end}$.

$$\frac{}{\Gamma \vdash \mathtt{tt},\mathtt{ff}:\mathsf{bool}}(\mathsf{Bool}) \quad \frac{}{\Gamma\cdot x:\mathsf{bool}\vdash x:\mathsf{bool}}(\mathsf{Var}) \quad \frac{}{\Gamma\cdot u:\mathtt{i}\langle S\rangle \vdash u:\mathtt{o}\langle S\rangle}(\mathsf{Shared'})$$

$$\frac{}{\Gamma\cdot u:T\vdash u:T}(\mathsf{Shared}) \quad \frac{\Gamma\vdash u:\langle S\rangle}{\Gamma\vdash \mathtt{arrived}\,u:\mathsf{bool}}(\mathsf{Areq}) \quad \frac{\Gamma,\Sigma\vdash \mathtt{arrived}\,k\,h:\mathsf{bool}}{\Gamma,\Sigma\vdash \mathtt{arrived}\,k\,:\mathsf{bool}}(\mathsf{Asess})$$

$$\frac{\Gamma\vdash v:U}{\Gamma,\Sigma\cdot k:?(U);S\vdash \mathtt{arrived}\,k\,v:\mathsf{bool}}(\mathsf{Amsg}) \quad \frac{l\in\{l_i\}_{i\in I}}{\Gamma,\Sigma\cdot k:\&\{l_i:S_i\}_{i\in I}\vdash \mathtt{arrived}\,k\,l:\mathsf{bool}}(\mathsf{Alab})$$

$$\frac{\Gamma\vdash P\triangleright\Sigma' \quad \Sigma'\le\Sigma}{\Gamma\vdash P\triangleright\Sigma}(\mathsf{Subs})$$

$$\frac{\Gamma\vdash u:\mathtt{i}\langle S\rangle \quad \Gamma\vdash P\triangleright\Sigma\cdot x:S}{\Gamma\vdash u(x:S).P\triangleright\Sigma}(\mathsf{Acc}) \quad \frac{\Gamma\vdash u:\mathtt{o}\langle S\rangle \quad \Gamma\vdash P\triangleright\Sigma\cdot x:\overline{S}}{\Gamma\vdash \overline{u}(x:\overline{S});P\triangleright\Sigma}(\mathsf{Req})$$

$$\frac{\Gamma,\Sigma\vdash e:U \quad \Gamma\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k!\langle e\rangle;P\triangleright\Sigma\cdot k:!(U);S}(\mathsf{Send}) \quad \frac{\Gamma\cdot x:U\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k?(x).P\triangleright\Sigma\cdot k:?(U);S}(\mathsf{Recv})$$

$$\frac{\Gamma\vdash P\triangleright\Sigma\cdot k:S}{\Gamma\vdash k!\langle k'\rangle;P\triangleright\Sigma\cdot k:!(T);S\cdot k':T}(\mathsf{SSend}) \quad \frac{\Gamma\vdash P\triangleright\Sigma\cdot k:S\cdot x:T}{\Gamma\vdash k?(x).P\triangleright\Sigma\cdot k:?(T);S}(\mathsf{SRecv})$$

$$\frac{1\le i\le n \quad \Gamma\vdash P\triangleright\Sigma\cdot k:S_i}{\Gamma\vdash k\triangleleft l_i;P\triangleright\Sigma\cdot k:\oplus\{l_i:S_i\}_{i\in I}}(\mathsf{Select}) \quad \frac{\forall i.1\le i\le n \quad \Gamma\vdash P_i\triangleright\Sigma\cdot k:S_i}{\Gamma\vdash k\triangleright\{l_i:P_i\}_n\triangleright\Sigma\cdot k:\&\{l_i:S_i\}_{i\in I}}(\mathsf{Branch})$$

$$\frac{\Gamma,\Sigma\vdash e:\mathsf{bool}\ \Gamma\vdash P\triangleright\Sigma\ \Gamma\vdash Q\triangleright\Sigma}{\Gamma\vdash \mathtt{if}\,e\,\mathtt{then}\,P\,\mathtt{else}\,Q\triangleright\Sigma}(\mathsf{If}) \quad \frac{\Gamma\cdot a:\langle S\rangle\vdash P\triangleright\Sigma\cdot a}{\Gamma\vdash(\nu\,a:\langle S\rangle)P\triangleright\Sigma}(\mathsf{Chan}) \quad \frac{\Gamma\vdash P\triangleright\Sigma \quad \Gamma\vdash Q\triangleright\Sigma'}{\Gamma\vdash P\mid Q\triangleright\Sigma\cdot\Sigma'}(\mathsf{Par})$$

$$\frac{\Sigma\ \mathtt{end\ only}}{\Gamma\vdash\mathbf{0}\triangleright\Sigma}(\mathsf{Nil}) \quad \frac{\Gamma\cdot X:\Delta\vdash P\triangleright\Delta}{\Gamma\vdash\mu X.P\triangleright\Delta}(\mathsf{Rec}) \quad \frac{}{\Gamma\cdot X:\Delta\vdash X\triangleright\Delta}(\mathsf{Pvar})$$

$$\frac{\forall i\in I \quad \Gamma\vdash P_i\triangleright\Sigma\cdot x_i:T_i \quad \cup_{i\in I}T_i\le T}{\Gamma\vdash \mathtt{typecase}\,k\,\mathtt{of}\,\{(x_i:T_i)\,P_i\}_{i\in I}\triangleright\Sigma\cdot k:T}(\mathsf{Typecase}) \quad \frac{\Sigma\ \mathtt{end\ only}}{\Gamma\vdash a[\varepsilon]\triangleright\Sigma\cdot a}(\mathsf{Queue})$$

Fig. 10. Typing rules for programs.

Rule (Typecase) is an extension from the dynamic typing system of the λ-calculus [1] to session types. It checks the body P_i is typed under Σ with x_i assigned to T_i. Then the whole process is typable with x assigned to T which is a supertype of all T_i.

C Typing Runtime Processes

This subsection gives typing for runtime processes following the presentation in [27]. The judgement is extended as: $\Gamma\vdash P\triangleright\Delta$ with $\Delta ::= \Sigma \mid \Delta\cdot s:[S,\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]$ where the configuration element $[S,\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]$ records the active type S of the configuration and types of values enqueued in the buffers. We extend T with labels ($T ::= U \mid S \mid l$). $(S,[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}])$ pairs a type S for the session s and the type information $[S,\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]$ for the associated configuration at s. We identify $(S,[\ldots])$ with $([\ldots],S)$.

The composition of Δ_1 and Δ_2, denoted by $\Delta_1\odot\Delta_2$ [27], is defined as:

$$\Delta_1\odot\Delta_2=\{s:(\Delta_1(s)\odot\Delta_2(s))\mid s\in\mathrm{dom}(\Delta_1)\cap\mathrm{dom}(\Delta_2)\}\cup\Delta_1\setminus\mathrm{dom}(\Delta_2)\cup\Delta_2\setminus\mathrm{dom}(\Delta_1)$$

where $S\odot[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]=[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]\odot S=[S',\mathtt{i}:\vec{T},\mathtt{o}:\vec{T'}]$ if $S\le S'$; otherwise undefined. Next, we define *the session type remainder* S' obtained by subtracting a vector of message types $\vec{T}$ from a session type S, denoted by $S-\vec{T}=S'$, by: (1) $S-\varepsilon=$

$$\frac{\Gamma \vdash P \triangleright \Delta \;\; \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \odot \Delta'}(\mathsf{Par'}) \qquad \frac{\Gamma \vdash a : \mathtt{i}\langle S \rangle}{\Gamma \vdash a\,[\langle S \rangle : \vec{s}] \triangleright \cup_{s \in \vec{s}} \{s : (S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])\}, a}(\mathsf{Queue'})$$

$$\frac{}{\Gamma, \{s : S\} \vdash s : S}(\mathsf{ASreq'})$$

$$\frac{\Gamma \vdash a : \mathtt{o}\langle S \rangle}{\Gamma \vdash \overline{a}\,\langle s \rangle \triangleright s : (S, [S, \mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon])}(\mathsf{Endpt}) \qquad \frac{\Gamma \vdash P \triangleright \Delta \quad s, \overline{s} \in \mathsf{dom}(\Delta) \quad \mathrm{wc}(\Delta, s)}{\Gamma \vdash (\nu\, s) P \triangleright \Delta \setminus \{s, \overline{s}\}}(\mathsf{SRes})$$

$$\frac{\forall i \leq m. \Gamma, \Sigma_i \vdash h_i : T_i \quad \forall j \leq n. \Gamma, \Sigma'_j \vdash h'_j : T'_j \quad \Sigma = \Sigma_0 \cdot \Sigma_1 \cdots \Sigma_m \cdot \Sigma'_1 \cdots \Sigma'_n \quad \mathrm{ct}(\Sigma_0)}{\Gamma \vdash s\,[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}] \triangleright \Sigma \odot s : [S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}]}(\mathsf{Config})$$

Fig. 11. Selected typing rules for runtime processes.

S; (2) $?(T);S - T \cdot \vec{T}$ if $S - \vec{T} = S'$; and (3) $\&\{l_i : S_i\}_{i \in I} - l_i \cdot \vec{T} = S'$ if for all $i \in I$, $S_i - \vec{T} = S'$. Our session type remainder differs from that in [27] because we require the stronger condition that a well-formed configuration with an output prefixed active type cannot have a non-empty local `i`-buffer or be composed with an configuration with a non-empty `o`-buffer. We say a Δ-environment is *well-configured with respect to a session s*, written $\mathrm{wc}(\Delta, s)$, if the following is satisfied:

$$\Delta(s) = [S_1, \mathtt{i} : \vec{T}_1, \mathtt{o} : \vec{T}'_2],\ \Delta(\overline{s}) = [S_2, \mathtt{i} : \vec{T}_2, \mathtt{o} : \vec{T}'_1] \ \text{ implies } \ S'_i = S_i - (\vec{T}_i \cdot \vec{T}'_i)\ (i \in \{1,2\}),\ S'_1 \leq \overline{S'_2}$$

These conditions say that, at any stage of execution of an established session, the types of the remaining session implementations on each endpoint, modulo any messages buffered (i.e. to be consumed by pending input actions) at the local `i`-buffer and opposing `o`-buffer, should be subtypes of the active types in their respective configurations; and that the dual of the active type at one endpoint should be a supertype of the active type at the other. We say that Δ is *well-configured*, $\mathrm{wc}(\Delta)$, iff $\forall s \in \mathrm{dom}(\Delta), \mathrm{wc}(\Delta, s)$.

Figure 11 lists the rules for typing runtime processes. The composition by (Par') pairs up the types of each session endpoint with the associated configurations to form $(S, [S', \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}])$. Rule (Queue') creates a server-side type for each of the buffered session request messages, as well as recording the presence of the a-queue in the Δ-environment like (Queue). Buffered session endpoints (i.e. endpoints being delegated) are typed using (Endpt), which creates a Σ-context holding the type of the endpoint; by typing each buffered message under a separate Σ-context, the concatenation of these contexts ensures the linearity of buffered endpoints. (SRes) checks that the body of the session restriction is well-configured with respect to the session, i.e. that the session implementations on each endpoint and the two session configurations together constitute a valid runtime state in the consistent execution of the session. Rule (Config) types all messages enqueued within the `i`- and `o`-buffers of the endpoint configuration to construct the $[S, \mathtt{i} : \vec{T}, \mathtt{o} : \vec{T'}]$ representation of the configuration. Using the properties of the ordering between environments, $\Delta \sqsubset \Delta'$, we have:

Theorem 3.1 (Type Soundness) *If $\Gamma \vdash P \triangleright \Delta$, $wc(\Delta)$ and $P \longrightarrow Q$, then $\Gamma \vdash Q \triangleright \Delta'$ and either $\Delta = \Delta'$ or $\Delta \sqsubset \Delta'$.*

Proof. The proof is done by using the induction on the structure of the typing rule, following each operational transition case.

Case [Request1]**.**
Let $\overline{a}(x).P \longrightarrow (\nu\, s)(P\{s/x\} \mid \overline{s}\,[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid \overline{a}\langle\overline{s}\rangle)$. By typing both sides of the transition, we have that $\Gamma \vdash \overline{a}(x).P \triangleright \Delta$ and $\Gamma \vdash (\nu\, s)(P\{s/x\} \mid \overline{s}\,[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid \overline{a}\langle\overline{s}\rangle) \triangleright \Delta$ as required. Note that $\Gamma = \Gamma' \cdot a : \mathtt{o}\langle S\rangle$.

Case [Request2]**.**
Let $\overline{a}\langle s\rangle \mid a[\vec{s}] \longrightarrow a[\vec{s}\cdot s]$. If we type both sides of the transition, we get $\Gamma \vdash \overline{a}\langle s\rangle \mid a[\vec{s}] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S$ and $\Gamma \vdash a[\vec{s}\cdot s] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S$ as required.

Case [Accept]**.**
Let $a(x).P \mid a[s\cdot\vec{s}] \longrightarrow a(x).P\{s/x\} \mid s\,[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon] \mid a[\vec{s}]$. If we type both sides of the transition we have $\Gamma \vdash a(x).P \mid a[s\cdot\vec{s}] \triangleright \Delta \cdot a \cdot s : S$ and $\Gamma \vdash P\{a/x\} \mid a[\vec{s}] \triangleright \Delta \cdot a \cdot s : S$ as required.

Case [Send]**.**
Let $\langle s\rangle v; P \mid s\,[\mathtt{o}:\vec{v}] \longrightarrow P \mid s\,[\mathtt{o}:\vec{v}\cdot v]$. If we type both sides of the transition, we have $\Gamma \vdash \langle s\rangle v; P \mid s\,[\mathtt{o}:\vec{v}] \triangleright \Delta \cdot s :!\langle T\rangle; S$ and $\Gamma \vdash P \mid s\,[\mathtt{o}:\vec{v}\cdot v] \triangleright \Delta \cdot s : S$ as required.

Case [Send]**.**
Let $s!\langle s'\rangle; P \mid s\,[\mathtt{o}:\vec{v}] \mid s'\,[\mathtt{i}:\vec{v},\mathtt{o}:\vec{v'}] \longrightarrow P \mid s\,[\mathtt{o}:\vec{v}\cdot s'] \mid s'\,[\mathtt{i}:\vec{v},\mathtt{o}:\vec{v'}]$. If we type both sides of the transition, we have $\Gamma \vdash s!\langle s'\rangle; P \mid s\,[\mathtt{o}:\vec{v}] \mid s'\,[\mathtt{i}:\vec{v},\mathtt{o}:\vec{v'}] \triangleright \Delta \cdot s' : S' \cdot s :!\langle S'\rangle; S$ and $\Gamma \vdash P \mid s\,[\mathtt{o}:\vec{v}\cdot v] \mid s'\,[\mathtt{i}:\vec{v},\mathtt{o}:\vec{v'}] \triangleright \Delta \cdot s' : S' \cdot s : S$ as required.

Below we assume the following as the inductive hypothesis (IH).
Let $P \longrightarrow P'$ and $\Gamma \vdash P \triangleright \Delta$ and Δ *well configured*, then $\Gamma \vdash P' \triangleright \Delta'$ and $\Delta = \Delta'$ or $\Delta \sqsubset \Delta'$.

Case [Par]**.**
Let $P \mid Q \longrightarrow P \mid Q'$. If we type both sides of the transition we have that $\Gamma \vdash P \mid Q \triangleright \Delta_P \odot \Delta_Q$ and $\Gamma \vdash P \mid Q' \triangleright \Delta_P \odot \Delta'_Q$. From the (IH), we have that $\Delta_Q \sqsubset \Delta'_Q$ and from the definition of $\odot$ operator the result follows.

Case [Chan]**.**
Let $(\nu\, a)(P) \longrightarrow (\nu\, a)(P')$. From the (IH), we have that $\Gamma \cdot a : \mathtt{i}\langle S\rangle \vdash P \triangleright \Delta \cdot a$ and $\Gamma \cdot a : \mathtt{i}\langle S\rangle \vdash P' \triangleright \Delta' \cdot a$ and $\Delta \cdot a \sqsubset \Delta' \cdot a$. From here we can see that $\Gamma \cdot a : \mathtt{i}\langle S\rangle \vdash (\nu\, a)(P) \triangleright \Delta$ and $\Gamma \cdot a : \mathtt{i}\langle S\rangle \vdash (\nu\, a)(P') \triangleright \Delta'$ and $\Delta \sqsubset \Delta'$ as required.

Case [Sess]**.**
Let $(\nu\, s)(P) \longrightarrow (\nu\, s)(P')$. From the (IH), we have that $\Gamma \vdash P \triangleright \Delta \cdot s : S$ and $\Gamma \vdash P' \triangleright \Delta' \cdot s : S'$ and $\Delta \cdot s : S \sqsubset \Delta' \cdot s : S'$. By runtime typing rule (SRes), we have that $\Gamma \vdash (\nu\, s)(P) \triangleright \Delta$ and $\Gamma \vdash (\nu\, s)(P') \triangleright \Delta'$ and $\Delta \sqsubset \Delta'$ as required.

D Appendix for Section 3

Transitions: We write $\Longrightarrow$ 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.

Subject and Object: We define the subject ($\mathsf{subj}(\ell)$) and object ($\mathsf{obj}(\ell)$) for actions ℓ. For session transitions, we have $\mathsf{subj}(a\langle s\rangle) = \mathsf{subj}(\overline{a}\langle s\rangle) = \mathsf{subj}(\overline{a}(s)) = \{a\}$ and $\mathsf{obj}(a\langle s\rangle) = \mathsf{obj}(\overline{a}\langle s\rangle) = \mathsf{obj}(\overline{a}(s)) = \{s\}$. For sesion transitions, we have $\mathsf{subj}(s!\langle v\rangle) = \mathsf{subj}(s?\langle x\rangle) = \mathsf{subj}(s!(a)) = \mathsf{subj}(s\oplus l) = \mathsf{subj}(s\&l) = \{a\}$ and $\mathsf{obj}(s!\langle v\rangle) = \{v\}$, $\mathsf{obj}(s?\langle x\rangle) = \{x\}$, $\mathsf{subj}(s!(a)) = \{a\}$ $\mathsf{subj}(s\oplus l) = \mathsf{subj}(s\&l) = \{l\}$.

Free and Bound Names: Free and bound names follow the definition, $\mathsf{fn}(a\langle s\rangle) = \mathsf{fn}(\overline{a}\langle s\rangle) = \{a,s\}$, $\mathsf{fn}(\overline{a}(s)) = \{a\}$, $\mathsf{bn}(a\langle s\rangle) = \mathsf{bn}(\overline{a}\langle s\rangle) = \emptyset$, $\mathsf{bn}(\overline{a}(s)) = \{s\}$ for shared name actions. For session name actions, we have $\mathsf{fn}(s!\langle v\rangle) = \mathsf{fn}(s?\langle x\rangle) = \mathsf{fn}(s!(a)) = \mathsf{fn}(s \oplus l) = \mathsf{fn}(s\&l) = \{s\}$, where v is not a name, $\mathsf{fn}(s!\langle a\rangle) = \{s,a\}$, $\mathsf{fn}(s!\langle s'\rangle) = \{s,s'\}$ and $\mathsf{bn}(s!\langle v\rangle) = \mathsf{fn}(s?\langle x\rangle) = \mathsf{subj}(s!\langle a\rangle) = \mathsf{bn}(s \oplus l) = \mathsf{bn}(s\&l) = \emptyset$, $\mathsf{bn}(s!(a)) = \{a\}$, $\mathsf{bn}(s!(s')) = \{s'\}$.

The names of an action ($\mathsf{n}(\ell)$) is the union of $\mathsf{fn}(\ell)$ and $\mathsf{bn}(\ell)$, $\mathsf{n}(\ell) = \mathsf{fn}(\ell) \cup \mathsf{bn}(\ell)$.

Contexts: We define the context of a process. Assume that α is a prefix:

- $\alpha \in \{a(s).,\overline{a}(s).,s!\langle v\rangle;,s?(x).,s!\lhd l;\}$.
- $C ::= - \mid C \mid C \mid C \mid D \mid (\nu\ \tilde{n})(C) \mid \texttt{if}\ e\ \texttt{then}\ C\ \texttt{else}\ C \mid \alpha.C$
  $\mid \mu X.C \mid s \rhd \{l_i : C\}_{i\in I} \mid X \mid \mathbf{0}$
- $D ::= s[\texttt{i} : \vec{h}_i, \texttt{o} : \vec{h}_o] \mid a[\vec{s}]$.

D.1 Proof for Theorem 3.3

Theorem (Coincidence) $\approx$ and $\cong$ coincide.

The above theorem requires to show the equality into two directions.

Lemma D.1 (Soundness). $\approx$ *implies* $\cong$

Proof. The prove of the soundness direction follows $P \approx Q$ implies $\forall\ C,\ C[P] \approx C[Q]$ and $\forall\ C,\ C[P] \approx C[Q]$ implies $\forall\ C,\ C[P] \cong C[Q]$

To prove the above it is enough to show the first implication, that bisimulation is a congruence, meaning that bisimulation is preserved through parallel composition, prefixed processes and the restriction operator. The second implication is trivial.

To show that input prefixed processes are congruent, consider:

$$P \approx Q \text{ implies } \alpha.P \approx \alpha.Q \text{ where } \alpha \text{ is an input prefix.}$$

Assume that

$$S = \{(\alpha.P, \alpha.Q) | P \approx Q, \alpha.P, \alpha.Q \text{ typable}, \alpha \text{ is an input prefix}\}$$

then S is a bisimulation.

As a note here, consider the case where:

$$s!\langle v\rangle; \mathbf{0} \mid s[!\langle T\rangle; \texttt{end}, \texttt{i} : \varepsilon, \texttt{o} : \varepsilon] \approx \mathbf{0} \mid s[!\langle T\rangle; \texttt{end}, \texttt{i} : \varepsilon, \texttt{o} : v]$$

If we prefix the above pair with $s?(v')$. Then we have

$$s?(v').s!\langle v\rangle; \mathbf{0} \mid s[?(T'); !\langle T\rangle; \texttt{end}, \texttt{i} : \varepsilon, \texttt{o} : \varepsilon] \text{ and } s?(v').\mathbf{0} \mid s[?(T'); !\langle T\rangle; \texttt{end}, \texttt{i} : \varepsilon, \texttt{o} : v]$$

Obviously the second process is not typable. The runtime type of the s buffer does not agree with the process typing.

The proof that S is a bisimulation considers a simple reduction on the processes of S to reach to their bisimilar point.

Output prefix and restriction are trivial.

The interesting case is the parallel composition of processes:

$$P \approx Q \text{ implies } P \mid R \approx Q \mid R \tag{1}$$

We assume

$$S = \{(P \mid R, Q \mid R) | P \approx Q, \forall R \text{ such that } P|R, Q|R \text{ are localised and typable}\}$$

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 S:

$$\Gamma \vdash Q \triangleright \Delta_Q \stackrel{\ell}{\Longrightarrow} Q' \triangleright \Delta_Q' \text{ and} \tag{2}$$

$$\Delta_P' \bowtie \Delta_Q' \tag{3}$$

By (2) we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} Q' \mid R \triangleright \Delta_2'$. It remains to show that $\Delta_1' \bowtie \Delta_2'$.

By composing parallel processes, we have that $\Delta_1' = \Delta_P' \odot \Delta_R$ and $\Delta_2' = \Delta_Q' \odot \Delta_R$.

From the definition of bisimulation (Definition 3.1), we have that P, Q, P' and Q' are localised, hence: $\forall\, s \in \mathsf{dom}(\Delta_P')$, $\Delta_P'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$, $\forall\, a \in \mathsf{dom}(\Delta_P')$, $\Delta_P'(a) = a$, $\forall\, s \in \mathsf{dom}(\Delta_Q')$, $\Delta_Q'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$ and $\forall\, a \in \mathsf{dom}(\Delta_Q')$, $\Delta_Q'(a) = a$.

From the above, the definition of $\odot$ operator and localisation of P and Q we can conclude that $\mathsf{dom}(\Delta_P') \cap \mathsf{dom}(\Delta_R) = \emptyset$ and $\mathsf{dom}(\Delta_Q') \cap \mathsf{dom}(\Delta_R) = \emptyset$. The last condition, along with (3) gives that $P \mid R \approx Q \mid R$ as required.

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 $\Gamma \vdash Q \mid R \triangleright \Delta_2 \xrightarrow{\ell} Q \mid R' \triangleright \Delta_2'$. It remains to show that $\Delta_1' \bowtie \Delta_2'$.

By composition of parallel processes, we have that $\Delta_1' = \Delta_P \odot \Delta_R'$ and $\Delta_2' = \Delta_Q \odot \Delta_R'$.

From the the definition of bisimulation (Definition 3.1),we have that P and Q' are localised, so: $\forall\, s \in \mathsf{dom}(\Delta_P)$, $\Delta_P'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$, $\forall\, a \in \mathsf{dom}(\Delta_P)$, $\Delta_P'(a) = a$, $\forall\, s \in \mathsf{dom}(\Delta_Q)$, $\Delta_Q'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$ and $\forall\, a \in \mathsf{dom}(\Delta_Q)$, $\Delta_Q'(a) = a$.

From the above, the definition of $\odot$ and localisation we can conclude that: $\mathsf{dom}(\Delta_P) \cap \mathsf{dom}(\Delta_R') = \emptyset$ and $\mathsf{dom}(\Delta_Q') \cap \mathsf{dom}(\Delta_R') = \emptyset$.

From here, we conclude $P \mid R \approx Q \mid R$ as required.

Case (3) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \longrightarrow P' \mid R' \triangleright \Delta_1'$. Then we have

$$\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta_P' \tag{4}$$

$$\tag{5}$$

By the definition of S, we have:

$$\Gamma \vdash Q \triangleright \Delta_Q \Longrightarrow \xrightarrow{\ell} \Longrightarrow Q' \triangleright \Delta_Q' \tag{6}$$

$$\Delta_P' \bowtie \Delta_Q' \tag{7}$$

By (6), we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \Longrightarrow Q' \mid R' \triangleright \Delta_2'$. Then it remains to show that $\Delta_1' \bowtie \Delta_2'$. By composing parallel processes, we have that $\Delta_1' = \Delta_P' \odot \Delta_R'$ and $\Delta_2' = \Delta_Q' \odot \Delta_R'$

From the definition of the bisimulation we have that P, Q, P' and Q' are localised, so: $\forall\ s \in \mathsf{dom}(\Delta_P'),\ \Delta_P'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$, $\forall\ a \in \mathsf{dom}(\Delta_P'),\ \Delta_P'(a) = a$, $\forall\ s \in \mathsf{dom}(\Delta_Q'),\ \Delta_Q'(s) = (S, [S', \mathtt{i} : \vec{h}_i, \mathtt{o} : \vec{h}_o])$ and $\forall\ a \in \mathsf{dom}(\Delta_Q'),\ \Delta_Q'(a) = a$.

From the above and the definition of $\odot$ and the localisation we can conclude: $\mathsf{dom}(\Delta_P') \cap \mathsf{dom}(\Delta_R') = \emptyset$ and $\mathsf{dom}(\Delta_Q') \cap \mathsf{dom}(\Delta_R') = \emptyset$. The last result along with (7) gives us that $P \mid R \approx Q \mid R$ as required. □

The proof for the completeness direction follows the technique shown in [9]. However we need to adapt it to session and buffers.

Definition D.1. An external action ℓ is *definable* if for for a set of names N, actions $succ, fail \notin N$ there is a *testing process* $T\langle N, succ, fail, \ell\rangle$ with the property that for every process P and $\mathsf{fn}(P) \subseteq N$

- $P \xrightarrow{\ell} P'$ implies that $T\langle N, succ, fail, \ell\rangle \mid P \Longrightarrow (\nu\ \mathsf{bn}(\ell))(succ\,[\mathtt{o}\!:\!\mathsf{bn}(\ell)] \mid P')$
- $T\langle N, succ, fail, \ell\rangle \mid P \Longrightarrow Q$, where $Q \Downarrow_{succ}, Q \not\Downarrow_{fail}$ implies that $Q = (\nu \mathsf{bn}(\ell))(succ\,[\mathtt{o}\!:\!\mathsf{bn}(\ell)] \mid P')$

Lemma D.2. *Every external action is definable.*

Proof. The input action cases are straightforward:

1. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{a\langle s\rangle} P' \triangleright \Delta_{P'}$ then $T = \overline{a}\langle s\rangle \mid succ\,[\mathtt{o}\!:\!\mathtt{tt}]$.
2. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{s?\langle v\rangle} P' \triangleright \Delta_{P'}$ then $T = \overline{s}\,[\mathtt{o}\!:\!v] \mid succ\,[\mathtt{o}\!:\!\mathtt{tt}]$.
3. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{s\&\ell} P' \triangleright \Delta_{P'}$ then $T = \overline{s}\,[\mathtt{o}\!:\!\ell] \mid succ\,[\mathtt{o}\!:\!\mathtt{tt}]$.

The requirements of Definition D.1 can be verified with simple transitions.

Output actions cases:

1. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\overline{a}\langle s\rangle} P' \triangleright \Delta_{P'}$ then we have,

$$T = fail\langle\rangle \mid a[\varepsilon] \mid succ\,[\mathtt{o}\!:\!\varepsilon] \mid a(y).\ (\mathtt{if}\ y = s\ \mathtt{then}\ (fail[\varepsilon] \mid succ!\langle y\rangle; \mathbf{0})\ \mathtt{else}\ \ (\nu\ b)(b(x).succ!\langle y\rangle; \mathbf{0} \mid b[\varepsilon]))$$

2. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\overline{a}(s)} P' \triangleright \Delta_{P'}$.
   Then we have:

$$T = fail\langle\rangle \mid a[\varepsilon] \mid succ\,[\mathtt{o}\!:\!\varepsilon] \mid a(y).\ (\mathtt{if}\ y \in N\ \mathtt{then}\ (\nu\ b)(b(x).succ!\langle y\rangle; \mathbf{0} \mid b[\varepsilon])\ \mathtt{else}\ (fail[\varepsilon] \mid succ!\langle y\rangle; \mathbf{0}))$$

3. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{s!\langle b\rangle} P' \triangleright \Delta_{P'}$ or $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{s!\langle s'\rangle} P' \triangleright \Delta_{P'}$ then we have that:

$$T = fail\langle\rangle \mid \overline{s}\,[\mathtt{i}\!:\!\varepsilon] \mid succ\,[\mathtt{o}\!:\!\varepsilon] \mid \overline{s}?(y).\ (\mathtt{if}\ y = b\ \mathtt{then}\ (fail[\varepsilon] \mid succ!\langle y\rangle; \mathbf{0})\ \mathtt{else}\ \ (\nu\ b)(b(x).succ!\langle y\rangle; \mathbf{0} \mid b[\varepsilon]))$$

   respectively for $s!\langle s'\rangle$ we have for the `if` expression $y = s$.

4. If $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{s \oplus l_i} P' \triangleright \Delta_{P'}$ then we have that:

$$T = fail\langle\rangle \mid \overline{s}[\mathtt{i} : \varepsilon] \mid succ[\mathtt{o} : \varepsilon] \mid s \triangleright \{l_i : fail[\varepsilon] \mid succ!\langle y\rangle; \mathbf{0}\}_{i \in I}, 1 \leq i \leq n$$

Again the requirements of Definition D.1 can be verified by simple transitions for each case.

Note that process $(\nu\, b)(b(x).succ!\langle y\rangle; \mathbf{0} \mid b[\varepsilon])$ is used to maintain the entire process typable. Obviously this process is bisimilar to $\mathbf{0}$.

□

Lemma D.3. *If succ is fresh, $b \in \vec{a} \cdot \vec{s}$ and*

$$\Gamma \vdash (\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \triangleright \Delta_1 \cong (\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \triangleright \Delta_2 \tag{8}$$

then

$$\Gamma \vdash P \triangleright \Delta_P \cong Q \triangleright \Delta_Q \tag{9}$$

Proof. Define

$$\begin{aligned} S = \{&(\Gamma \vdash P \triangleright \Delta_P, \Gamma \vdash Q \triangleright \Delta_Q) \mid \\ &\Gamma \vdash (\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \triangleright \Delta_1 \cong (\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \triangleright \Delta_2, succ \text{ is fresh}\} \end{aligned}$$

We shall show that the contextual properties hold in S.

Reduction Closedness: S is reduction closed by the freshness of *succ*. We cannot observed a τ transition on *succ*, so we conclude that if $\Gamma \vdash (\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \triangleright \Delta_1 \Longrightarrow (\nu \vec{a}, \vec{s})(P' \mid succ[\mathtt{o} : b]) \triangleright \Delta_1'$ implies $\Gamma \vdash (\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \triangleright \Delta_2 \Longrightarrow (\nu \vec{a}, \vec{s})(Q' \mid succ[\mathtt{o} : b]) \triangleright \Delta_2'$ then $\Gamma \vdash P \triangleright \Delta_P \Longrightarrow P' \triangleright \Delta_P'$ implies $\Gamma \vdash Q \triangleright \Delta_Q \Longrightarrow Q' \triangleright \Delta_Q'$

It remains to show that $\Delta_P' \bowtie \Delta_Q'$. From results (8) and (9) and the (Par) typing rule we conclude that: $\Delta_1' \bowtie \Delta_2'$, $\Delta_1' = \Delta_P' \cdot succ : [o : T]$ and $\Delta_2' = \Delta_Q' \cdot succ : [o : T]$.

We also have the assumption that *succ* is fresh, so: $\forall\, s \in \mathsf{dom}(\Delta_1),\ s \neq succ$ and $\forall\, s \in \mathsf{dom}(\Delta_2),\ s \neq succ$.

The result is immediate by the definition of the $\bowtie$ operator.

Preserve Observation: If $P \Downarrow c$ then if $c \notin \vec{a} \cdot \vec{s}$ then $(\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \Downarrow c$ implies $(\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \Downarrow c$. From the freshness of *succ*, we conclude $Q \Downarrow c$

Suppose now the case where $c \in \vec{a} \cdot \vec{s}$. We define $T = succ?(x).OK!\langle\rangle; \mathbf{0}$. Under the assumption that $P \Downarrow c$ we conclude that $T \mid (\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \Downarrow OK$.

From the context property of (8) we have that $\Gamma \vdash T \mid (\nu \vec{a}, \vec{s})(P \mid succ[\mathtt{o} : b]) \triangleright \Delta_1 \cong \Gamma \vdash T \mid (\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \triangleright \Delta_2$ and $T \mid (\nu \vec{a}, \vec{s})(Q \mid succ[\mathtt{o} : b]) \Downarrow OK$. But this can only happen if $Q \Downarrow c$ as required.

Context Property: The interesting case is the parallel composition. We shall show that if $\Gamma \vdash P \triangleright \Delta_P\ S\ Q \triangleright \Delta_Q$. Then for arbitrary process R we have that $\Gamma \vdash P \mid R \triangleright \Delta_{PR}\ S\ Q \mid R \triangleright \Delta_{QR}$ and $\Delta_{PR} \bowtie \Delta_{QR}$.

If we consider that *succ* may occur in R, then to conclude the above it is enough to show that: $\Gamma \vdash (\nu \vec{a}, \vec{s})(P \mid succ'[\mathtt{o} : b]) \mid R \triangleright \Delta \cong (\nu \vec{a}, \vec{s})(Q \mid succ'[\mathtt{o} : b] \mid Q) \triangleright \Delta'$ where

$succ'$ is fresh. Assume the process: $T = succ?(x).succ'!\langle \mathtt{tt}\rangle; R \mid succ'[\mathtt{o}:\varepsilon]$, then from the contextual property of the theorem assumption, we have that: $\Gamma \vdash (\nu \vec{a},\vec{s})(P \mid succ[\mathtt{o}:b]) \mid T \triangleright \Delta \cong (\nu \vec{a},\vec{s})(Q \mid succ[\mathtt{o}:b] \mid T) \triangleright \Delta'$.

By the above, it is easy to prove (D.1).

We now show the typing result. Due to localisation, we have $\Delta_P \cap \Delta_R = \emptyset$ and $\Delta_Q \cap \Delta_R = \emptyset$. From the definition of $\cong$, we have that $\Delta_P \bowtie \Delta_Q$ This implies: $\Delta_P \odot \Delta_R \bowtie \Delta_Q \odot \Delta_R$, as required. □

We are know ready to prove the completeness direction.

Lemma D.4 (Completness). $\cong$ *implies* $\approx$

Proof. For the proof we show that if

$$\Gamma \vdash P \triangleright \Delta_P \;\cong\; Q \triangleright \Delta_Q \text{ and} \tag{10}$$

$$\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta'_P \tag{11}$$

then $\Gamma \vdash Q \triangleright \Delta_Q \xrightarrow{\ell} Q' \triangleright \Delta'_Q$ and $\Gamma \vdash P' \triangleright \Delta'_P \cong Q' \triangleright \Delta'_Q$

Suppose (10) and (11). 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.

Because $\cong$ is context preserving we have that $\Gamma \vdash P \mid T \triangleright \Delta_{PT} \cong Q \mid T \triangleright \Delta_{QT}$. By Lemma D.1 we have that $\Gamma \vdash P \mid T \triangleright \Delta_{PT} \Longrightarrow (\nu\ \mathsf{bn}(\ell))(succ[\mathtt{o}:\mathsf{bn}(\ell)]|P') \triangleright \Delta$ so by the definition of $\cong$ (Definition 3.2), we have that $\Gamma \vdash T \mid Q \triangleright \Delta_{QT} \Longrightarrow R \triangleright \Delta'$. According to the second part of the Definition D.1, we can write:

$$\Gamma \vdash R \triangleright \Delta' \;=\; \Gamma \vdash (\nu\ \mathsf{bn}(\ell))(succ[\mathtt{o}:\mathsf{bn}(\ell)]|Q') \triangleright \Delta' \tag{12}$$

$$\Gamma \vdash Q \triangleright \Delta_Q \stackrel{\ell}{\Longrightarrow} Q' \triangleright \Delta'_Q \tag{13}$$

Now we can derive $\Gamma \vdash (\nu\ \mathsf{bn}(\ell))(succ[\mathtt{o}:\mathsf{bn}(\ell)]|P') \triangleright \Delta \cong R \triangleright \Delta'$ and $\Gamma \vdash (\nu\ \mathsf{bn}(\ell))(succ[\mathtt{o}:\mathsf{bn}(\ell)]|P') \triangleright \Delta \cong \Gamma \vdash (\nu\ \mathsf{bn}(\ell))(succ[\mathtt{o}:\mathsf{bn}(\ell)]|Q') \triangleright \Delta'$. By Lemma D.3 we conclude that:

$$\Gamma \vdash P' \triangleright \Delta'_P \cong Q' \triangleright \Delta'_Q \tag{14}$$

$$\Delta'_P \bowtie \Delta'_Q \tag{15}$$

We began with the assumption that $\Gamma \vdash P \triangleright \Delta_P \cong \Gamma \vdash Q \triangleright \Delta_Q$ and we concluded to (13), (14) and (15). Thus $\cong$ implies $\approx$.

□

D.2 Bisimulation Properties

Proof for Lemma 3.4 We define input and output actions.

Definition (Input/Output Actions).

1. ℓ is an input action if $\ell \in \{a\langle s\rangle, a\langle s\rangle, s?\langle v\rangle, s?(v), s\&l_i\}$

2. ℓ is an output action if $\ell \in \{\overline{a}\langle s\rangle, \overline{a}(s), s!\langle v\rangle, s!(v), s \oplus l_i\}$

Lemma (Input and Output Asynchrony).

Suppose $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

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

Proof. For the first part of Lemma 3.4, there are two cases.

Case (1) The action takes place on a shared channel. Process P has the form $R \mid a[\vec{s}]$, thus $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \stackrel{a\langle s\rangle}{\Longrightarrow} R' \mid a[\vec{s'} \cdot s] \triangleright \Delta'$ with $\vec{s} \cdot s = \vec{s}_\delta \cdot \vec{s'}, (\vec{s}_\delta = \vec{s'}_\delta \cdot s \wedge \vec{s'} = \emptyset) \vee \vec{s'} = \vec{s''} \cdot s$, $\Delta' = \Delta'_R \odot s : S$. So $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \stackrel{a\langle s\rangle}{\longrightarrow} R \mid a[\vec{s} \cdot s] \triangleright \Delta'' \Longrightarrow R' \mid a[\vec{s'}] \triangleright \Delta'''$ with $\vec{s} \cdot s = \vec{s}_\delta \cdot \vec{s'}, (\vec{s}_\delta = \vec{s'}_\delta \cdot s \wedge \vec{s'} = \emptyset) \vee \vec{s'} = \vec{s''} \cdot s$ and $\Delta''' = \Delta'_R \odot s : S$.

We conclude the same processes with the same typing as required.

Case (2) Input communication takes place on a session channel. It is similar using a session queue.

For the second part of Lemma 3.4, there are two cases.

The first case is the action happening on a shared channel. Then $\Gamma \vdash P \triangleright \Delta \Longrightarrow P'' \mid \overline{a}\langle s\rangle \stackrel{a!\langle s\rangle}{\longrightarrow} P'' \Longrightarrow P' \triangleright \Delta'$.

From this we can always conclude that $\Gamma \vdash P \triangleright \Delta \Longrightarrow P' \mid \overline{a}\langle s\rangle \stackrel{a!\langle s\rangle}{\longrightarrow} P' \triangleright \Delta'$.

It is obvious that $\Delta = \Delta'$.

For the second case, we have the action to be observed on a session channel. $\Gamma \vdash P \mid s\,[\mathtt{o} : v \cdot \vec{h}] \triangleright \Delta \stackrel{s!\langle v\rangle}{\Longrightarrow} P' \mid s\,[\mathtt{o} : \vec{h'}] \triangleright \Delta'$, where $\vec{h'} = \vec{h} \cdot \vec{h_\delta}$.

Thus we can see that $\Gamma \vdash P \mid s\,[\mathtt{o} : v \cdot \vec{h}] \triangleright \Delta \Longrightarrow P' \mid s\,[\mathtt{o} : v \cdot \vec{h'}] \stackrel{s!\langle v\rangle}{\longrightarrow} P' \mid s\,[\mathtt{o} : \vec{h'}] \triangleright \Delta'$, where $v \cdot \vec{h'} = v \cdot \vec{h} \cdot \vec{h_\delta}$, which concludes the same result as above, with $\Delta = \Delta'$. □

Proof for Lemma 3.8

Lemma. Let P be session determinate and Q be a session derivative of P. Then $P \approx Q$.

Proof. The proof considers induction on the length of $\Longrightarrow_s$ transition. The basic step is trivial. For the induction step 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 $s?(x).Q \mid s\,[\mathtt{i} : v \cdot \vec{h}] \approx Q \mid s\,[\mathtt{i} : \vec{h}]$.

Due to $s \notin \mathsf{fn}(R)$ we can compose in parallel bisimilar processes and get that $P' \approx P''$ as required.

The rest of the cases follow similar arguments. □

Confluence by Construction

Lemma D.1. *1. If P,Q be confluent processes and $\mathsf{fn}(P)\cap\mathsf{fn}(Q)=\emptyset$ and $P\mid Q$ is typable then $P\mid Q$ is confluent.*
2. If P is confluent and $a\notin\mathsf{fn}(P)$ then $a(x).P\mid a[\varepsilon]$ is confluent.

Proof. For the first part of the proof we need to show that if $\Gamma\vdash P\mid Q\triangleright\Delta\stackrel{\tilde{\ell}}{\Longrightarrow}R\triangleright\Delta'$, then if $R\stackrel{\ell_1}{\longrightarrow}R_1$ and $R\stackrel{\ell_2}{\Longrightarrow}R_2$ then $R_1\stackrel{\ell_2}{\Longrightarrow}R'$, $R_2\stackrel{\ell_1}{\Longrightarrow}R''$ and $R'\approx R''$.

Since $\mathsf{fn}(P)\cap\mathsf{fn}(Q)=\emptyset$, we have that $R=P'\mid Q'$, $P\stackrel{\tilde{\ell'}}{\Longrightarrow}P'$ and $Q\stackrel{\tilde{\ell''}}{\Longrightarrow}Q'$ and P' and Q' are confluent.

We now do a case analysis on the possible $\stackrel{\ell_1}{\Longrightarrow},\stackrel{\ell_2}{\Longrightarrow}$ transitions and proof the bisimilarity of the resulting processes.

To present a case assume that ℓ_1,ℓ_2 happen on P' but $\Longrightarrow$ transition is observed in both P' and Q'. Recall that the P' and Q' cannot interact because of free name disjointness. So we have that $P'\mid Q'\stackrel{\ell_1}{\longrightarrow}P_1\mid Q'$ and $P'\mid Q'\stackrel{\ell_2}{\Longrightarrow}P_2\mid Q_2$. Because P' is confluent we have that $P_1\mid Q'\stackrel{\ell_2}{\Longrightarrow}P''\mid Q''$ and $P_2\mid Q_2\stackrel{\ell_1}{\Longrightarrow}P'''\mid Q'''$ and $P''\approx P'''$ (because of the confluence of P').

We also conclude that: $Q'\Longrightarrow Q''$ and $Q'\Longrightarrow Q'''$ and from Lemma 3.8, we have that $Q'\approx Q''$ $Q'\approx Q'''$ which concludes that $Q''\approx Q'''$.

To show that $P''\mid Q''\approx P'''\mid Q'''$ we need to check the actions that each process can do. Again recall that P'',Q'' and P''',Q''' are free name disjoint. So if $P''\mid Q''\stackrel{\ell}{\longrightarrow}P_0\mid Q''$ and $P''\mid Q''\stackrel{\ell'}{\longrightarrow}P''\mid Q_0$ then $P'''\mid Q'''\stackrel{\ell}{\Longrightarrow}P'_0\mid Q''$ and $P'''\mid Q'''\stackrel{\ell'}{\Longrightarrow}P'''\mid Q'_0$ because $P''\approx P'''$ and $Q''\approx Q'''$. We apply a similar analysis for the other direction and the result follows as required.

For the second part of the proof, we have that $a(x).P\mid a[\varepsilon]\stackrel{a\langle s\rangle}{\longrightarrow}a(x).P\mid a[s]\longrightarrow P\{s/x\}\mid a[\varepsilon]$ and $P\{s/x\}\mid a[\varepsilon]$ is confluent by Part 1 of this lemma. □

Proof for Lemma 3.9

Lemma D.5. *Let typable, localised P and actions ℓ_1,ℓ_2 such that $\mathsf{subj}(\ell_1),\mathsf{subj}(\ell_2)$ are session names and $\ell_1\bowtie\ell_2$. If $\Gamma\vdash P\triangleright\Delta\stackrel{\ell_1}{\longrightarrow}P_1\triangleright\Delta_1$ and $\Gamma\vdash P\triangleright\Delta\stackrel{\ell_2}{\longrightarrow}P_1\triangleright\Delta_2$ then $\Gamma\vdash P_1\triangleright\Delta\stackrel{\ell_2\lfloor\ell_1}{\longrightarrow}P'\triangleright\Delta'$ and $\Gamma\vdash P_2\triangleright\Delta\stackrel{\ell_1\lfloor\ell_2}{\longrightarrow}P'\triangleright\Delta'$*

Proof. The result is an easy case analysis on all the possible comptations of ℓ_1,ℓ_2.

We give an interesting case. Let $(\nu\ a)(P\mid s_1[\mathsf{o}:\vec{h_1}\cdot a]\mid s_2[\mathsf{o}:\vec{h_2}\cdot a])\stackrel{s_1!\langle a\rangle}{\longrightarrow}P\mid s_1[\mathsf{o}:\vec{h_1}]\mid s_2[\mathsf{o}:\vec{h_2}\cdot a]$ and $(\nu\ a)(P\mid s_1[\mathsf{o}:\vec{h_1}\cdot a]\mid s_2[\mathsf{o}:\vec{h_2}\cdot a])\stackrel{s_2!\langle a\rangle}{\longrightarrow}P\mid s_1[\mathsf{o}:\vec{h_1}\cdot a]\mid s_2[\mathsf{o}:\vec{h_2}]$. Now it is easy to see that $P\mid s_1[\mathsf{o}:\vec{h_1}]\mid s_2[\mathsf{o}:\vec{h_2}\cdot a]\stackrel{s_2!\langle a\rangle}{\longrightarrow}P\mid s_1[\mathsf{o}:\vec{h_1}]\mid s_2[\mathsf{o}:\vec{h_2}]$ and $P\mid s_1[\mathsf{o}:\vec{h_1}\cdot a]\mid s_2[\mathsf{o}:\vec{h_2}]\stackrel{s_1!\langle a\rangle}{\longrightarrow}P\mid s_1[\mathsf{o}:\vec{h_1}]\mid s_2[\mathsf{o}:\vec{h_2}]$ as required. □

Two useful lemmas follow.

Lemma D.6. *Let P be session determinate. Then if $\Gamma\vdash P\triangleright\Delta\stackrel{\ell}{\longrightarrow}P'\triangleright\Delta'$ and $\Gamma\vdash P\triangleright\Delta\stackrel{\ell}{\Longrightarrow}P''\triangleright\Delta''$ then $P'\approx P''$*

Proof. There are two cases:

Case τ:. Follow Lemma 3.8 to get $P \approx P'$ and $P \approx P''$. The result then follows.

Case ℓ:. Suppose that $P \xrightarrow{\ell}_s P'$ and $P \xRightarrow{\ell}_s P''$ implies $P \Longrightarrow_s P_1 \xrightarrow{\ell}_s P_2 \Longrightarrow_s P''$. From Lemma 3.8, 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 3.8 once more to get $P' \approx P''$ as required. □

Lemma D.7. *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 $\Gamma \vdash P \triangleright \Delta \xRightarrow{\ell_2} P_2 \triangleright \Delta_2$. Then $\Gamma \vdash P_1 \triangleright \Delta_1 \xRightarrow{\ell_2 \lfloor \ell_1} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta_2 \xRightarrow{\ell_1 \lfloor \ell_2} P'' \triangleright \Delta''$ and $P' \approx P''$*

Proof. The proof considers a case analysis on the combination of ℓ_1, ℓ_2.

Case $\ell_1 = s_1!\langle v_1 \rangle, \ell_2 = s_2?\langle v_2 \rangle$.

$$
\begin{aligned}
P \mid s_1[\mathtt{o}:\vec{h_1} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_2}] &\xrightarrow{\ell_1}_s P_1 \mid s_1[\mathtt{o}:\vec{h_1}] \mid s_2[\mathtt{i}:\vec{h_2}] \\
&\Longrightarrow_s P_1' \mid s_1[\mathtt{o}:\vec{h_1'}] \mid s_2[\mathtt{i}:\vec{h_2'}] \\
&\xrightarrow{\ell_2}_s P_1' \mid s_1[\mathtt{o}:\vec{h_1'}] \mid s_2[\mathtt{i}:\vec{h_2'} \cdot v_2] \\
&\Longrightarrow_s P' \mid s_1[\mathtt{o}:\vec{h_1''}] \mid s_2[\mathtt{i}:\vec{h_2''}] \\
P \mid s_1[\mathtt{o}:\vec{h_1} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_2}] &\Longrightarrow_s P_0 \mid s_1[\mathtt{o}:\vec{h_0} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_0'}] \\
&\xrightarrow{\ell_2}_s P_0' \mid s_1[\mathtt{o}:\vec{h_0} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_0'} \cdot v_2] \\
&\Longrightarrow_s P_2 \mid s_1[\mathtt{o}:\vec{h_2'} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_2''} \cdot v_2] \\
&\Longrightarrow_s P_2' \mid s_1[\mathtt{o}:\vec{h_3} \cdot v_1] \mid s_2[\mathtt{i}:\vec{h_3'}] \\
&\xrightarrow{\ell_2}_s P_2' \mid s_1[\mathtt{o}:\vec{h_4}] \mid s_2[\mathtt{i}:\vec{h_4'}] \\
&\Longrightarrow_s P'' \mid s_1[\mathtt{o}:\vec{h'}] \mid s_2[\mathtt{i}:\vec{h''}]
\end{aligned}
$$

By using Lemma 3.4, 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 the lemmas D.5, 3.4 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 considers Lemma 3.8.

We summarise the above to prove Lemma 3.9.

Lemma. If P is 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

$$\forall \Gamma \vdash P \triangleright \Delta_0 \xRightarrow{\tilde{\ell}}_s Q \triangleright \Delta$$

and Q can only do $\xRightarrow{\tilde{\ell}}_s$ actions. By Lemma D.7 (resp. D.6 we can see that P gives us the required property. □

Proof for Lemma 3.11

Lemma. Let $\mathscr{R}$ be a determinate upto-expansion relation. Then $\mathscr{R}$ is inside a bisimulation.

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

□

E Comparison with Asynchronous/Synchronous Calculi

E.1 Behavioural Theory for Session Type System with Input Buffer Endpoints

Before we prove the relations in § 6, we define a behavioural theory for the asynchronous session π-calculus with two end-point queues but without IO-queues [6, 7, 27].

$$\langle \mathsf{Acc}_A \rangle \quad a[\vec{s}] \xrightarrow{a\langle s\rangle} a[\vec{s}\cdot s] \quad \langle \mathsf{Req}_A \rangle \quad \overline{a}\langle s\rangle \xrightarrow{\overline{a}\langle s\rangle} \mathbf{0} \qquad \langle \mathsf{In}_A \rangle \quad s[\vec{h}] \xrightarrow{s?\langle v\rangle} s[\vec{h}\cdot v]$$

$$\langle \mathsf{Out}_A \rangle \quad s!\langle v\rangle; P \xrightarrow{s!\langle v\rangle} P \qquad \langle \mathsf{Bra}_A \rangle \quad s[\vec{h}] \xrightarrow{s\&l} s[\vec{h}\cdot l] \qquad \langle \mathsf{Sel}_A \rangle \quad s!\lhd l; P \xrightarrow{s\oplus l} P$$

$$\langle \mathsf{Local}_A \rangle \frac{P \longrightarrow Q}{P \xrightarrow{\tau} Q} \quad \langle \mathsf{Par}_A \rangle \frac{P \xrightarrow{\ell} P' \ \mathsf{bn}(\ell)\cap \mathsf{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\ell} P' \mid Q} \quad \langle \mathsf{Tau}_A \rangle \frac{P \xrightarrow{\ell} P' \ Q \xrightarrow{\ell'} Q' \ \ell \asymp \ell'}{P \mid Q \xrightarrow{\tau} (\nu\, \mathsf{bn}(\ell,\ell'))(P' \mid Q')}$$

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

$$\langle \mathsf{OpenN}_A \rangle \frac{P \xrightarrow{\overline{s}\langle a\rangle} P'}{(\nu\, a)P \xrightarrow{\overline{s}(a)} P'} \quad \langle \mathsf{Alpha}_A \rangle \frac{P \equiv_\alpha P' \ P' \xrightarrow{\ell} Q}{P \xrightarrow{\ell} Q}$$

Fig. 12. Labelled Transition for Session Type System with Two Buffer Endpoint Without IO

A labelled transition system is given in Figure 12. The LTS is similar to the LTS of the calculus studied in this paper (4), except from the output actions. Shared channels, and input actions have identical transition labels. The output actions cannot be observed on the buffer since there is no output buffer defined. Instead they are observed in an output reduction of a process.

E.2 Proofs for Section 6

We prove the results in § 6 for the two asynchronous session typed π-calculus, 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 $s!\langle w\rangle; s!\langle v\rangle; P \mid s\,[\mathsf{o}:\varepsilon] \not\xRightarrow{s!\langle v\rangle}$ as required.

2. **Case** $s_1!\langle v\rangle; s_2!\langle w\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon] \approx s_2!\langle w\rangle; s_1!\langle v\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]$**.**
Relation:

$$\begin{aligned} R = \{\ & (s_1!\langle v\rangle; s_2!\langle w\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]\ ,\ s_2!\langle w\rangle; s_1!\langle v\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]),\\ & (s_2!\langle w\rangle; P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon]\ ,\ P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]\ ,\ s_1!\langle v\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]\ ,\ P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:w]),\\ & (s_2!\langle w\rangle; P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]\ ,\ P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]\ ,\ P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:\varepsilon]),\\ & (P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]\ ,\ P \mid s_1[\mathtt{o}:\varepsilon] \mid s_2[\mathtt{o}:w]),\\ & (P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon]\ ,\ P \mid s_1[\mathtt{o}:v] \mid s_2[\mathtt{o}:\varepsilon])\} \end{aligned}$$

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

$$\begin{aligned} R = \{\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]),\\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]),\\ & (s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]\ ,\ s_1?(x).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]\ ,\ P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]\ ,\ s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (s_2?(y).P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:w]\ ,\ s_2?(y).s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]),\\ & (s_1?(x).s_2?(y).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:w]\ ,\ s_1?(x).P \mid s_1[\mathtt{i}:v] \mid s_2[\mathtt{i}:\varepsilon]),\\ & (P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]\ ,\ P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon])\} \end{aligned}$$

gives the result.

E.3 Arrived Operators in the π-Calculi

In this subsection, we define two arrive inspected calculi that try to simulate blocking and order-preserving properties as the synchronous and asynchronous π-calculi, respectively.

For the synchronous π-calculus, we have blocking and order-preserving input and output and for the asynchronous π-calculus we have non-blocking and non-order preserving input and output.

In the context of the synchronous π-calculus, we cannot easily define the `arrived` operator without a slight compromise of the non blocking input property, due to the asynchronous nature of the `arrived` operator on the input queue of an endpoint.

The synchronous π-calculus can be represented asynchronously by having channel buffers of size one.

Syntax of the Synchronous-like π-Calculus with Arrive.

$$P ::= \mathbf{0} \mid a[\varepsilon] \mid a(x).P \mid \overline{a}\langle v\rangle.P \mid P|P \mid (\nu\ a)P \mid \mathtt{if\ arrived}\ a\ \mathtt{then}\ P\ \mathtt{else}\ P$$

Labelled Transition Semantics of the Synchronous π-Calculus with Arrive.

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

$$\frac{P \xrightarrow{\ell} P',\ \mathsf{fn}(\ell)\cap\mathsf{fn}(Q)=\emptyset}{P|Q \xrightarrow{\ell} P'|Q} \qquad \frac{P \xrightarrow{\ell} P',\ Q \xrightarrow{\ell'} Q',\ \ell \asymp \ell'}{P|Q \xrightarrow{\tau} (\nu\ \mathsf{bn}(\ell,\ell'))(P'|Q')} \qquad \frac{P \xrightarrow{\ell} P'\ n \notin \mathsf{fn}(\ell)}{(\nu\, n)P \xrightarrow{\ell} (\nu\, n)P'}$$

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

$$\texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q|a[\varepsilon] \xrightarrow{\tau} Q|a[\varepsilon]$$
$$\texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q|a[v] \xrightarrow{\tau} P|a[v]$$

In the synchronous π-calculus with `arrived` operator, channel buffers have size of one and we can receive a value from the environment, only if a corresponding process is ready to receive on the buffer channel.

To demonstrate the compromise done in to achieve this definition consider

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

The first process is in the classic synchronous π calculus. We can only observe one input action. For the second system we observe an asynchronous input action. First a message is put in the communication buffer and then the actual receive happens. Between the two transition an arrive inspection can happen.

The asynchronous π-calculus with `arrived` operator is easier to be defined in a queue context. The idea here is to have endpoints that use a random policy for message exchange:

Syntax of the Asynchronous π-Calculus with Arrive.

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

Labelled Transition Semantics of the Asynchronous π-Calculus with Arrive.

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

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

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

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

$$\texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q|a[\varepsilon] \xrightarrow{\tau} Q|a[\varepsilon]$$
$$\texttt{if arrived}\ a\ \texttt{then}\ P\ \texttt{else}\ Q|a[h\cdot\vec{h}] \xrightarrow{\tau} P|a[\vec{h}]$$

The above definition disallows the order preserving property in the system but keeps the non blocking property as required by the asynchronous π-calculus.

Figure 7 shows that the arrive construct behaves the same on all of the calculi.

F Appendix for Selectors

F.1 Mapping

The selector operator is discussed in detailed along with its reduction properties and applications in [15]. In this section we present an ESP encoding of the selector.

We can extend ESP with the selector operations, with the following reduction semantics.

$$\begin{array}{l}
\texttt{new selector}\ r\ \texttt{in}\ P \longrightarrow (\nu\, r)(P \mid \mathsf{sel}\langle r, \varepsilon\rangle) \qquad \texttt{register}\langle s', r\rangle; P \mid \mathsf{sel}\langle r, \vec{s}\rangle \longrightarrow P \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \\
\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \\
\qquad\qquad \longrightarrow P_i\{s'/_{x_i}\} \mid \mathsf{sel}\langle r, \vec{s}\rangle \mid s'\,[S, \texttt{i}:\vec{h}] \quad (\vec{h} \neq \varepsilon) \\
\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, s'\cdot\vec{s}\rangle \mid s'\,[\texttt{i}:\varepsilon] \\
\qquad\qquad \longrightarrow \texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : P_i\}_{i\in I} \mid \mathsf{sel}\langle r, \vec{s}\cdot s'\rangle \mid s'\,[\texttt{i}:\varepsilon]
\end{array}$$

where in the third line S and T_i satisfies the condition for `typecase` in Figure 3. We also include the structural rules with garbage collection rules for queues as $(\nu\, r)\mathsf{sel}\langle r, \varepsilon\rangle \equiv \mathbf{0}$. Operator `new selector` r `in` P (binding r in P) creates a new selector $\mathsf{sel}\langle r, \varepsilon\rangle$, named r and with the empty queue ε. Operator $\texttt{register}\langle s', r\rangle; P$ registers a session channel s to r, adding s' to the original queue $\vec{s}$. $\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i : T_i) : P_i\}_{i\in I}$ retrieves a registered session and checks the availability to test if an event has been triggered. If so, find the match of the type of s' among $\{T_i\}$ and select P_i; if not, the next session is tested.

We now show this behaviour can be easily encoded by combining *arrival predicates* and *typecase*. Below we omit type annotations.

$$\begin{array}{l}
[\![\texttt{new selector}\ r\ \texttt{in}\ P]\!] \stackrel{\text{def}}{=} (\nu\, b)(\overline{b}(r).b(\overline{r}).[\![P]\!] \mid b : [\varepsilon]) \quad [\![\texttt{register}\langle s, r\rangle; P]\!] \stackrel{\text{def}}{=} \overline{r}!\langle s\rangle; [\![P]\!] \\
[\![\texttt{let}\ x = \texttt{select}(r)\ \texttt{in typecase}\ x\ \texttt{of}\ \{(x_i : S_i) : P_i\}_{i\in I}]\!] \stackrel{\text{def}}{=} \mathsf{Select}(r\overline{r}) \\
[\![\mathsf{sel}\langle r, \vec{s}\cdot\vec{s}'\rangle]\!] \stackrel{\text{def}}{=} r\,[\texttt{o}:\vec{s}'] \mid \overline{r}\,[\texttt{i}:\vec{s}]
\end{array}$$

$$\begin{array}{l}
\texttt{def}\ \mathsf{Select}(x\overline{x}) = \overline{x}?(y); \texttt{if arrived}\ y\ \texttt{then typecase}\ y\ \texttt{of}\{(x_i : S_i) : [\![P_i]\!]\}_{i\in I} \\
\qquad\qquad\qquad\qquad \texttt{else}\ x!\langle y\rangle; \mathsf{Select}\langle x\overline{x}\rangle \qquad\qquad\qquad\qquad \texttt{in}\ \ \mathsf{Select}\langle r\overline{r}\rangle
\end{array}$$

The use of `arrived` is the key to avoid blocked inputs, allowing the system to proceed asynchronously. The operations on the collection need to carry session channels, hence the use of delegation (linear channel passing) is essential [11]. We can easily check that the embedding operationally simulates the selector given above as the extension of ESP, and that, under a suitable bisimulation, that it is semantically faithful.

F.2 Typing

The typing rules for the selector are naturally suggested from the ESP-typing of its encoding. We write the type for a *user* of a selector storing channels of type T, by $\overline{\texttt{sel}}(T)$, and the type for a selector itself by $\texttt{sel}(T)$. For simplicity we assume these

types do not occur as part of other types. The linear environment Δ now includes two new type assignments, $r{:}\overline{\texttt{sel}}(T)$ and $r{:}\texttt{sel}(T)$. The typing rules for the selector follow.

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r{:}\overline{\texttt{sel}}(T)}{\Gamma \vdash \texttt{new selector } r \texttt{ in } TP \triangleright \Delta}(\textsf{Selector}) \quad \frac{\Gamma \vdash P \triangleright \Delta \cdot r{:}\overline{\texttt{sel}}(T) \quad S \leq T}{\Gamma \vdash \texttt{register}\langle s, r\rangle; P \triangleright \Delta \cdot r{:}\overline{\texttt{sel}}(T) \cdot s{:}S}(\textsf{Register})$$

$$\frac{\forall i \in I.\Gamma \vdash P_i \triangleright \Delta \cdot r{:}\overline{\texttt{sel}}(T) \cdot x_i{:}S_i \quad S_i \leq T}{\Gamma \vdash \texttt{let } x = \texttt{select}(r) \texttt{ in typecase } x \texttt{ of } \{(x_i{:}S_i) : P_i\}_{i\in I} \triangleright \Delta \cdot r{:}\overline{\texttt{sel}}(T)}(\textsf{Select})$$

The typing rule for the selector queue is similar to the runtime typing for a shared input queue. cf. Appendix C. By setting $[\![\Delta]\!]$ as the compositional mapping such that $[\![r{:}\overline{\texttt{sel}}(T)]\!]$ is given as $r{:}S_r \cdot \overline{r}{:}\overline{S_r}$ where $S_r = \mu \mathsf{X}.!(T);\mathsf{X}$, and otherwise identity, as well as extending the notion of error to the internal typecase of the select command, we obtain, writing ESP^+ for the extension of ESP with the selector:

Proposition F.1 (Soundness of Selector Typing Rules).

1. (Type Preservation) $\Gamma \vdash P \triangleright \Sigma$ *in* ESP^+ *if and only if* $\Gamma \vdash [\![P]\!] \triangleright [\![\Sigma]\!]$.
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 the error.*

Full proofs and further discussion are given in [15].

G Appendix: Lauer-Needham Transform

This appendix gives the detailed illustration of the Lauer-Needham transformation. Our translation uses the notations in Figure 13 for brevity, including: *pairs*, *polyadic inputs/outputs* [25], a *refined typecase*, a *refined selector*, and an *environment* as used in the standard CPS transform. All of them are easily encodable in the eventful calculus in [15].

The formal mapping follows. Below we say a process is *positive* if it is either an acceptor, an input, a branching or a definition, and is *blocking* if it is positive and is not a definition. The *subject* of a positive term is the initial channel name if it is blocking, and the initial process variable if it is a definition.

Definition G.1 (Lauer-Needham Transform). Let $*a(w:S);P$ be a simple server. Then the mapping $\mathscr{LN}[\![*a(w:S);P]\!]$ is inductively defined by the rules in Figure 13, assuming the following annotation on P: the subjects of distinct positive subterms in P are labelled with distinct numerals from 1 to n, as in e.g. $x^{(i)}?(y);Q$ (then we say the prefix is *blocking at* $x^{(i)}$), such that 1 to $n-m$ are used for the prefixes and the rest for the definitions ($n \geq m \geq 0$). We also assume all environments have the same fields named by the (free and bound) variables occurring in P which we assume to be pairwise distinct.

As outlined in the main section, the main map $\mathscr{LN}[\![*a(w:S);P]\!]$ consists of:

1. An *event loop* $\mathsf{Loop}\langle o, q\rangle$ which denotes a loop invoked at o without parameters. It also uses a selector queue q. It is composed with $\overline{o}$, initiating the loop.
2. A selector queue $q\langle a, c_0\rangle$ named q with a single element $\langle a, c_0\rangle$.
3. A collection of *code blocks* $\mathsf{CodeBlocks}\langle a, o, q, \vec{c}\rangle$, each defined using an auxiliary map $\mathscr{B}[\![R]\!]$ and $[\![Q, y]\!]$. Its behaviour is illustrated below.

$$\mathscr{LN}[\![*a(w{:}S);P]\!] \stackrel{\text{def}}{=} (\nu o,q,\vec{c})(\ \mathsf{Loop}\langle o,q\rangle \mid \overline{o} \mid q\langle a,c_0\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle\ )$$

where $P_1,..,P_n$ are the positive sub-terms of P; $P_1,..,P_{n-m}$ the blocking ones whose subjets are respectively typed $S_1,..,S_{n-m}$; and o, q and $\vec{c} = c_1..c_n$ are fresh and pairwise distinct.

$$\begin{aligned}\mathsf{Loop}\langle o,q\rangle \stackrel{\text{def}}{=} *o.\ &\texttt{let}\ w = \texttt{select}(q)\ \texttt{in}\ \texttt{typecase}\ w\ \texttt{of}\ \{\\ &(x:S,\ z:\mathsf{env}):\ \texttt{new}\ y:\mathsf{env}\ \texttt{in}\ \overline{z}\langle y\rangle,\\ &(x:S_1,\ y:\mathsf{env},\ z:S_1,\mathsf{env}):\ \overline{z}\langle x,y\rangle,\ \ldots\\ &(x:S_{n-m},\ y:\mathsf{env},\ z:S_{n-m},\mathsf{env}):\ \overline{z}\langle x,y\rangle\\ &\}\end{aligned}$$

$$\mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle \stackrel{\text{def}}{=} \mathscr{B}[\![a(w{:}S);P]\!] \mid \textstyle\prod_{1\le i\le n}\mathscr{B}[\![P_i]\!]$$

$$\mathscr{B}[\![*a(w{:}S).P]\!] \stackrel{\text{def}}{=} *c_0(y).a(w'{:}S).\texttt{update}(y,w,w');\ \texttt{register}\langle q,a,c_0\rangle;[\![P,y]\!]$$

$$\mathscr{B}[\![x^{(i)}?(z:T).Q]\!] \stackrel{\text{def}}{=} *c_i(x',y).x'?(z').\texttt{update}(y,z,z');\ \texttt{update}(y,x,x');\ [\![Q,y]\!]$$

$$\mathscr{B}[\![x^{(i)} \rhd \{l_j:Q_j\}_j]\!] \stackrel{\text{def}}{=} *c_i(x',y).x' \rhd \{l_j:\ \texttt{update}(y,x,x');\ [\![Q_j,y]\!]\}_j$$

$$[\![x!\langle e\rangle;Q,y]\!] \stackrel{\text{def}}{=} \texttt{let}\ x' = [\![x]\!]_y\ \texttt{in}\ x'!\langle[\![e]\!]_y\rangle;\texttt{update}(y,x,x');\ [\![Q,y]\!]$$

$$[\![x!\langle k\rangle;Q,y]\!] \stackrel{\text{def}}{=} \texttt{let}\ x' = [\![x]\!]_y\ \texttt{in}\ \texttt{let}\ k' = [\![k]\!]_y\ \texttt{in}\ x'!\langle k'\rangle;\texttt{update}(y,xk,x'k');\ [\![Q,y]\!]$$

$$[\![x \lhd l_j;Q,y]\!] \stackrel{\text{def}}{=} \texttt{let}\ z = [\![x]\!]_y\ \texttt{in}\ z \lhd l_j;[\![Q,y]\!]$$

$$[\![\overline{b}(z:S);Q,y]\!] \stackrel{\text{def}}{=} \overline{b}(z':S);\texttt{update}(y,z,z');\ [\![Q,y]\!]$$

$$[\![Q,y]\!] \stackrel{\text{def}}{=} \texttt{let}\ x' = [\![x]\!]_y\ \texttt{in}\ \texttt{register}\langle q,x',c_i,y\rangle;\overline{o}\ \ (Q \text{ is blocking at } x^{(i)})$$

$$[\![\mathbf{0},y]\!] \stackrel{\text{def}}{=} \overline{o}$$

Fig. 13. Translation Function for Lauer-Needham Transform

The initial execution of $\mathscr{LN}[\![*a(w:S);P]\!]$ starts from the *event-loop*. It fetches a channel at which a message has arrived by *select*: what it finds in the selector queue is checked and typed by the *typecase* construct from [15] (as illustrated in Appendix A). Initially it will only find a request via a. After finding it, the loop then creates a brand new environment and jumps to the *initial code block* at c_0, passing the environment.

Once invoked, the initial code block, $\mathscr{B}[\![a(w:S);P]\!]$, receives a fresh session channel through the buffer of a, saves it in the environment, and moves to $[\![P,y]\!]$. The code $[\![P,y]\!]$ 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 that blocking session channel, the associated continuation and the current environment in a *selector queue*. Then the control flow returns to the *event loop*.

The event loop then tries to sense the arrival of a message again by scanning the registered channels (shared and session). Assume it finds a message via a session channel this time. It then decides its type by *typecase* and invokes the corresponding continuation code block, passing the session channel and the environment. 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.

G.1 Lauer-Needham Transform properties

The translation $TE[\![*a(w:S);P]\!]$ radically changes the original code organisation. However the two technical underpinnings of our event-based formalism, types and bisimulations, offer a firm and tractable basis to reason about these processes, relating the event-based code to the original server. Below we discuss the key properties of our translation and its extensions. We say a closed typed process is *sequential* if its arbitrary well-typed multi-step transition derivative has at most one redex pair.

Proof for Lemma 4.4

Lemma. $*a(w:S);R \mid a[\varepsilon]$ is confluent.

Proof. R is confluent by definition.

A derivative of $*a(w:S);R \mid a[\varepsilon]$ is $R_1 \mid \ldots \mid R_n \mid *a(w:S);R \mid a[\varepsilon] \mid \prod_{k\in I} s_i[\mathtt{i}:\vec{h_{ii}},\mathtt{o}:\vec{h_{oi}}] \equiv R_1 \mid \ldots \mid R_n \mid *a(w:S);R \mid a(w:S);R \mid a[\varepsilon] \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}]$ where $I=\{1,\ldots,n\}$, which counts the session endpoints in the system.

Processes $R_1,\ldots,R_n$ do not share any names, so process:
$R_1 \mid \ldots \mid R_n \mid a(w:S);R \mid a[\varepsilon] \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}]$ is confluent by Lemma (D.1).

From structural congruence we conclude that the entire process is confluent. □

Proof for Lemma 4.5

Lemma. $TE[\![*a(w:S);P \mid a[\varepsilon]]\!]$ is confluent.

Proof. The structure of the transformation is constructed out of session transitions $\Longrightarrow_s$ and transitions on shared names. As we will see the transitions on shared names are confluent.

An ambiguous transition is the arrive transition defined in the selector used in the Loop process. We need to show that for the arrive derivative of the initial transformation the process follows the definition of confluent processes. Consider the process:

$$P = \mathtt{if\ arrived}\ s_i\ \mathtt{then}\ \text{typecase}\ s_i\ \text{of}\{\ldots\}\ \mathtt{else}\ \mathsf{Select} \mid a[\vec{s}] \\ \mid sel\langle(s_k,\xi_k,c_k),\ldots,(s_{k-1},\xi_{k-1},c_{k-1})\rangle \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}] \mid \mathsf{CodeBlocks}$$

where $I=\{1,\ldots,n\}$, which counts the session endpoints in the system.

When vector $\vec{h_{ii}} \neq \varepsilon$(for some i) then it is trivial to show the definition of confluence processes by showing a bisimulation on the resulting processes.

In the case where $\vec{h_{ii}} = \varepsilon$ and assuming that $l_1 = s_i?\langle v\rangle$ we have to show the confluent transitions and show that the resulting processes are bisimilar. We do a case analysis on the possible transitions $\stackrel{l_2}{\Longrightarrow}$.

Case $l_2 = s_j?\langle v' \rangle$**.**

$$
\begin{aligned}
P \xrightarrow{s_i?\langle v \rangle} P_1 = {} & \texttt{if arrived}\ s_i\ \texttt{then typecase}\ s_i\ \text{of}\{\dots\}\ \texttt{else Select} \mid a[\vec{s}] \mid \\
& sel\langle (s_k, \xi_k, c_k), \dots, (s_{k-1}, \xi_{k-1}, c_{k-1}) \rangle \mid \prod_{m \in I - \{i,j\}} s_m[\texttt{i}:\vec{h_{im}}, \texttt{o}:\vec{h_{om}}] \mid \\
& s_j[\texttt{i}:\vec{h_{ij}}, \texttt{o}:\vec{h_{oj}}] \mid s_i[\texttt{i}:\vec{h_{ii}} \cdot v, \texttt{o}:\vec{h_{oi}}] \mid \text{CodeBlocks} \\
P \xrightarrow{s_j?\langle v' \rangle} P_2 = {} & Q \mid a[\vec{s_2}] \mid sel\langle (s_{2k}, \xi_{2k}, c_{2k}), \dots, (s_{2k-1}, \xi_{2k-1}, c_{2k-1}) \rangle \mid \\
& \prod_{m \in I - \{i,j\}} s_m[\texttt{i}:\vec{h_{2im}}, \texttt{o}:\vec{h_{2om}}] \mid s_i[\texttt{i}:\vec{h2_{2ii}}, \texttt{o}:\vec{h_{2oi}}] \mid s_j[\texttt{i}:\vec{h_{2ij}} \cdot v', \texttt{o}:\vec{h_{2oj}}]
\end{aligned}
$$

with:

$$
\begin{aligned}
& \vec{s} = \vec{s_2'} \cdot \vec{s_2} \\
m = j \quad & , \vec{h_{im}} \cdot v' = \vec{h'_{2im}} \cdot \vec{h_{2im}} \\
1 \le m \le n \ & , \vec{h_{im}} = \vec{h'_{2im}} \cdot \vec{h_{2im}}, \vec{h_{2om}} = \vec{h'_{2om}} \cdot \vec{h_{om}}
\end{aligned}
$$

From the confluence definition, we have that:

$$
\begin{aligned}
P_1 \xrightarrow{s_j?\langle v' \rangle} P' = {} & Q' \mid a[\vec{s'}] \mid sel\langle (s'_k, \xi'_k, c'_k), \dots, (s'_{k-1}, \xi'_{k-1}, c'_{k-1}) \rangle \mid \\
& \prod_{m \in I - \{i,j\}} s_m[\texttt{i}:\vec{h'_{im}}, \texttt{o}:\vec{h'_{om}}] \mid s_i[\texttt{i}:\vec{h'_{ii}}, \texttt{o}:\vec{h'_{oi}}] \mid s_j[\texttt{i}:\vec{h'_{ij}}, \texttt{o}:\vec{h'_{oj}}] \\
P_2 \xrightarrow{s_i?\langle v \rangle} P'' = {} & Q'' \mid a[\vec{s''}] \mid sel\langle (s'_k, \xi'_k, c'_k), \dots, (s'_{k-1}, \xi'_{k-1}, c'_{k-1}) \rangle \mid \\
& \prod_{m \in I - \{i,j\}} s_m[\texttt{i}:\vec{h'_{im}}, \texttt{o}:\vec{h'_{om}}] \mid s_i[\texttt{i}:\vec{h''_{ii}}, \texttt{o}:\vec{h''_{oi}}] \mid s_j[\texttt{i}:\vec{h''_{ij}}, \texttt{o}:\vec{h''_{oj}}]
\end{aligned}
$$

with:

$$
\begin{aligned}
& \vec{s} = \vec{s_1} \cdot \vec{s'} \\
m = j \quad & \vec{h_{im}} \cdot v' = \vec{h_{1im}} \cdot \vec{h'_{im}} \\
1 \le m \le n \ & \vec{h_{im}} = \vec{h_{1im}} \cdot \vec{h'_{im}}, \vec{h'_{om}} = \vec{h_{1om}} \cdot \vec{h_{om}} \\
& \vec{s''} = \vec{s_2} \cdot \vec{s''} \\
m = i \quad & \vec{h_{2im}} \cdot v = \vec{h''_{2im}} \cdot \vec{h''_{im}} \\
1 \le m \le n \ & \vec{h_{2im}} = \vec{h''_{2im}} \cdot \vec{h''_{im}}, \vec{h''_{om}} = \vec{h''_{2om}} \cdot \vec{h_{2om}}
\end{aligned}
$$

At this point we need to show that $P' \approx P''$:
Let the symmetric relation $\mathscr{R}$ such that:

$$\mathscr{R} = \left\{ (P,Q) | P, Q \text{ have the form of } P', P'' \text{ respectively} \right\}$$

Assume that $P \mathscr{R} Q$ and $P \xrightarrow{s_k?\langle v \rangle} P'$. Then we can always observe $Q \xrightarrow{s_k?\langle v \rangle} Q'$ and $P' \mathscr{R} Q'$.

Suppose $P \xrightarrow{s_k!\langle v\rangle} P'$. Then by combining the relation between the buffers of P and Q, we have $Q \xrightarrow{s_k!\langle v\rangle} Q'$ or if $s_k[\mathtt{o}:\varepsilon]$ then $Q \Longrightarrow \xrightarrow{s_k!\langle v\rangle} Q'$ and $P'\mathscr{R}Q'$ in both cases.

We have similar results for $a\langle s\rangle$ and $\overline{a}\langle s\rangle$. as well as in the bounded objects actions.

The last case is also trivial. Suppose $P \longrightarrow P'$. Then $Q \Longrightarrow Q'$ and $P'\mathscr{R}Q'$.

This completes the case when $l_2 = s_i?\langle v'\rangle$. The case for $l_2 = s_i?(v')$ is similar. So is the case for $l_2 = a\langle s\rangle$. Cases $l_2 = s_i!\langle v\rangle, l_2 = s_i!(v)$ follow the same arguments. Case $l_2 = \overline{a}\langle s\rangle$ cannot happen by the specification of thread elimination.

Shared transitions should also be confluent. Suppose
$\overline{c_i}(s,\xi) \mid sel\langle s,\vec{\xi},c_i\rangle \mid \mathsf{Buffers}\langle I\rangle \mid \mathsf{Codeblocks}$. From here the confluence definition is trivial to be observe.

All other derivatives of $TE[\![*a(w:S);P \mid a[\varepsilon]]\!]$ are confluent due to the presence of $\Longrightarrow_s$ transition.

By the confluence definition the result follows. □

Proof for Lemma 4.6

Lemma Let
$P_1 = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_1,\xi_1,c_1),\ldots,(s_i,\xi_i,c_i),(s_j,\xi_j,c_j),\ldots,(s_n,\xi_n,c_n)\rangle \mid a[\vec{s}]$
$\mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}])$ and
$P_2 = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_1,\xi_1,c_1),\ldots,(s_j,\xi_j,c_j),(s_i,\xi_i,c_i),\ldots,(s_n,\xi_n,c_n)\rangle \mid a[\vec{s}]$
$\mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}])$.

Then $P_1 \approx P_2$.

Proof. Assume a relation $\mathscr{R}$ such that

$$Q_1 = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_k,\xi_k,c_k),\ldots,(s_i,\xi_i,c_i),(s_j,\xi_j,c_j),\ldots,(s_{k-1},\xi_{k-1},c_{k-1})\rangle \mid a[\vec{s}] \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}]),$$

$$Q_2 = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_k,\xi_k,c_k),\ldots,(s_j,\xi_j,c_j),(s_i,\xi_i,c_i),\ldots,(s_{k-1},\xi_{k-1},c_{k-1})\rangle \mid a[\vec{s}] \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}},\mathtt{o}:\vec{h_{om}}])$$

and $P_1\mathscr{R}P_2$, $Q_1\mathscr{R}Q_2$.

We want to show that $\mathscr{R}$ is an upto-expansion relation.

From Lemma 4.5 we have that Q_1, Q_2, P_1 and P_2 are confluent.

The requirement of typing holds, since both processes have the same typing due to the existence of the same endpoints in both related processes.

For every observable action ℓ the case is trivial, since if $Q_1 \xrightarrow{\ell} Q_1'$ then $Q_2 \xrightarrow{\ell} \mathscr{R}Q_1'$.

It remains to show the case for $\ell = \tau$.

If $Q_1 \longrightarrow Q_1'$ then

$$Q_2 \Longrightarrow Q_2' = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle\ldots,(s_{k-1},\xi_{k-1},c_{k-1}),(s_k,\xi_k,c_k),\ldots,(s_j,\xi_j,c_j),(s_i,\xi_i,c_i)\rangle \mid a[\vec{s}] \mid \prod_{m\in I} s_m[\mathtt{i}:\vec{h_{im}'},\mathtt{o}:\vec{h_{om}'}])$$

and

$$Q_1' \Longrightarrow (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle \ldots, (s_{k-1}, \xi_{k-1}, c_{k-1}), (s_k, \xi_k, c_k), \ldots, (s_i, \xi_i, c_i), (s_j, \xi_j, c_j)\rangle \mid a[\vec{s}] \mid \prod_{m \in I} s_m\,[\mathtt{i} : \vec{h'_{im}}, \mathtt{o} : \vec{h'_{om}}])$$

and $Q_1' \mathscr{R} Q_2'$ as required. □

Proof for Theorem 4.7

Theorem Let $*a(w:S);R \mid a[\varepsilon]$ be a simple server. Then $*a(w:S);P \approx TE[\![*a(w:S);P \mid a[\varepsilon]]\!]$.

Proof. Assume a relation $\mathscr{S}$ such that

$$\begin{array}{l} P = R_1 \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s}] \mid \prod_{m\in I} s_m\,[\mathtt{i} : \vec{h'_{im}}, \mathtt{o} : \vec{h'_{om}}] \\ Q = (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle (s_1, \xi_1, c_1), \ldots, (s_n, \xi_n, c_n)\rangle \mid a[\vec{s}] \\ \qquad \mid \prod_{m\in I} s_m\,[\mathtt{i} : \vec{h'_{im}}, \mathtt{o} : \vec{h'_{om}}]) \end{array}$$

and $(P,Q) \in \mathscr{S}$

with $R_1, \ldots, R_n$ blocking subterms of R

We will show that $\mathscr{S}$ is an upto-expansion relation.

From Lemmas 4.4 and 4.5, we verify the first requirement.

$$\frac{\Gamma \vdash a : \langle S\rangle, \Gamma \vdash R \triangleright x : S,\ \Gamma \vdash a[\vec{s}] \triangleright \vec{s} : \vec{S}}{\Gamma \vdash a(x).R \mid a[\vec{s}] \triangleright a \cdot \vec{s} : \vec{S}}$$

$$\frac{\Gamma \vdash \prod_{i\in I}(R_i \mid s_i\,[\mathtt{i} : \vec{h_{ii}}, \mathtt{o} : \vec{h_{oi}}]) \triangleright \prod_{i\in I} \vec{s_i} : S_i}{\Gamma \vdash R_1 \mid \ldots \mid R_n \mid *a(x).R \mid \mathsf{Buffers}\langle I\rangle \mid a[\vec{s}] \triangleright a \cdot \prod_{i\in I} \vec{s_i} : S_i \cdot \vec{s} : \vec{S}}$$

$$\frac{\forall\, R \in \mathsf{InputSubTerms}(R), \Gamma \vdash [\![R, y]\!] \triangleright \emptyset,\ \Gamma \vdash \mathscr{I}[\![R]\!] \triangleright \emptyset}{\Gamma \vdash \mathsf{CodeBlocks} \triangleright \emptyset}$$

$$\Gamma \vdash \mathsf{Loop} \triangleright r : S_r \cdot \bar{r} : S_{\bar{r}}$$

$$\frac{\Gamma \vdash \prod_{i\in I}(s_i\,[\mathtt{i} : \vec{h_{ii}}, \mathtt{o} : \vec{h_{oi}}] \mid r[\ldots] \mid \bar{r}[\ldots]) \triangleright \prod_{i\in I} \vec{s_i} : S_i \cdot r : S_r \cdot \bar{r} : S_r'}{\Gamma \vdash (\nu\ o\vec{c}r)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid a[\vec{s}] \mid \mathsf{Buffers}\langle I\rangle \mid r\langle s_i \vec{y}_i\rangle) \triangleright a \cdot \prod_{i\in I} \vec{s_i} : S_i \cdot \vec{s} : \vec{S}}$$

We can check both processes have the same typing.

For the case when observing external actions, the result is trivial since one process can simulate the other.

For $l = \tau$ we have:

If

$$(\nu\ \vec{c}or)\ (\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_1,\xi_1,c_1),\ldots,(s_n,\xi_n,c_n)\rangle \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]) \longrightarrow$$
$$(\nu\ \vec{c}or)\ (P \mid r\langle(s_1,\xi_1,c_1),\ldots,(s_n,\xi_n,c_n)\rangle \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}])$$

then

$$R_1 \mid \ldots \mid R_n \mid *a(x).R \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}] \Longrightarrow$$
$$R'_1 \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]$$

with R'_1 to be a blocking subprocess of R and

$$(\nu\ \vec{c}or)\ (P \mid r\langle(s_1,\xi_1,c_1),\ldots,(s_n,\xi_n,c_n)\rangle \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]) \Longrightarrow$$
$$(\nu\ \vec{c}or)\ (\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_2,\xi_2,c_2),\ldots,(s_1,\xi_1,c_1)\rangle \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}])$$

as required.

Now for the symmetric direction we have that if:

$$R_1 \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}] \longrightarrow$$
$$R_1 \mid \ldots \mid R'_i \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]$$

then we can use Lemma 4.6 to choose a bisimilar process to the event server with a permuted selector such that

$$(\nu\ \vec{c}or)\ (\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle(s_i,\xi_i,c_i),\ldots\rangle \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]) \Longrightarrow$$
$$(\nu\ \vec{c}or)\ (\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle\ldots,(s_i,\xi_i,c_i)\rangle \mid a[\vec{s'}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]$$

and

$$R_1 \mid \ldots \mid R'_i \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}] \Longrightarrow$$
$$R'_1 \mid \ldots \mid R'_i \mid \ldots \mid R_n \mid *a(x).R \mid a[\vec{s'}] \mid \textstyle\prod_{m\in I} s_m\,[\mathtt{i}:\vec{h'_{im}},\mathtt{o}:\vec{h'_{om}}]$$

The resulting processes are related by $\mathscr{S}$.

Then we conclude the case with Lemma 3.11.