Under consideration for publication in Math. Struct. in Comp. Science

On Asynchronous Eventful Session Semantics

Dimitrios Kouzapas[1], Nobuko Yoshida[1], Raymond Hu[1], and Kohei Honda[2]

[1]*Department of Computing, Imperial College London*
[2]*Department of Computer Science, Queen Mary, University of London*

Received 23 January 2014

Event-driven programming is one of the major paradigms in concurrent and communication-based programming, where events are typically detected as the arrival of messages on asynchronous channels. Unfortunately, the flexibility and performance of traditional event-driven programming come at the cost of more complex programs: low-level APIs and the obfuscation of event-driven control flow make programs difficult to read, write and verify.

This paper introduces a π-calculus with session types that models *event-driven session programming* (called ESP) and studies its behavioural theory. The main characteristics of the ESP model are asynchronous, order-preserving message passing, non-blocking detection of event/message arrivals, and dynamic inspection of session types. Session types offer formal safety guarantees, such as communication and event-handling safety, and programmatic benefits that overcome problems with existing event-driven programming languages and techniques. The new typed bisimulation theory developed for the ESP model is distinct from standard synchronous or asynchronous bisimulation, capturing the semantic nature of eventful session-based processes. The bisimilarity coincides with reduction-closed barbed congruence.

We demonstrate the features and benefits of event-driven session programming and the behavioural theory through two key usecases. First, we examine an encoding and the semantic behaviour of the event selector, a central component of general event-driven systems, providing core results for verifying type-safe event-driven applications. Second, we examine the Lauer-Needham duality, building on the selector encoding and bisimulation theory to prove that a systematic transformation from multithreaded to event-driven session processes is type- and semantics-preserving.

1. Introduction

Structured event-driven asynchronous communications. *Event-driven programming* is characterised by a reactive flow of control driven by the occurrence of computation events. It is one of the major paradigms in concurrent and communication-based programming, where events are typically detected as the arrival of messages on asynchronous channels. Primary motivations for event-driven programming (against multithreading, the other major paradigm for concurrent programming in practice) include performance and scalability, particularly for highly-concurrent applications such as Web servers (Welsh et al., 2001; Krohn, 2004). Unfortunately, the flexibility and performance of traditional event-driven programming come at the cost of more complex programs: low-level APIs and the obfuscation of event-driven control flow (von Behren et al., 2003a; Adya et al., 2002) make programs difficult to read, write and verify. Consequently, sev-

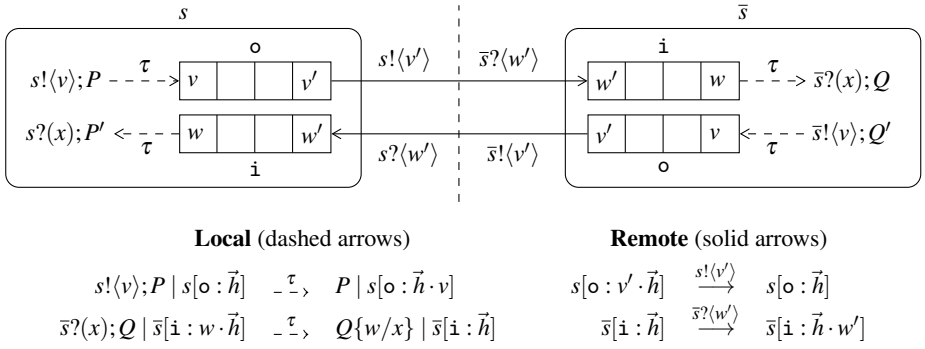
eral recent works (Li and Zdancewic, 2007; Krohn et al., 2007; von Behren et al., 2003b) have proposed simpler thread-based programming interfaces that hide event-driven runtimes.

In contrast to such approaches, our first aim in this article is to develop a high-level, structured and safe programming discipline for event-driven communication programming based on, and extending, *session types* (Takeuchi et al., 1994; Honda et al., 1998). We generalise the existing session types to support *event-driven session programming* (ESP), from which we obtain both formal safety guarantees and programmatic benefits to overcome the problems of traditional event-driven programming.

A key mechanism underlying general event-based systems is the facility for non-blocking detection of heterogeneous events (i.e. of varied types) across a dynamic collection of channels. This facility is provided by e.g. the Unix `select` system call and the Java NIO `Selector` API. In the context of session types, this means an ESP framework should allow us to store a collection of channels of *different* session types, perform *non-blocking checks* for the arrival of asynchronous messages on these channels, and retrieve "ready" channels from the collection to handle the events as directed by their session types. The preceding session type disciplines cannot support ESP due to the lack of *non*-blocking input, which prohibits core event-driven idioms such as event loops, and because statically determined channel types make it impossible to treat a dynamic collection of channels with heterogeneous types.

The first part of this article explores a theoretical basis of ESP using a small process calculus based on (Honda and Tokoro, 1991; Takeuchi et al., 1994; Honda et al., 1998). Following the above observations, the event abstraction is expressed in our formalism in terms of three properties: (1) *asynchrony* (the occurrence of an event is communicated asynchronously from its source); (2) *non-blocking detection* (event occurrences are detected via explicit messaging actions, but without blocking even when no event has occurred); (3) *dynamic session type inspection* (event types are inspected dynamically, but safely, to perform the correct event handling). These properties and the semantics of event-driven asynchronous sessions are captured through two new constructs: the *message arrival predicate* and the *session typecase*. We use FIFO queues at *both* endpoints of a channel to model (1), to fully decouple the communication actions performed by processes from the underlying network, and in particular decouple the asynchronous arrival event of a message at an endpoint from the act of consuming the message. (2) is then modelled by the way the message arrival predicate interacts with these asynchronous endpoints. The last property (3) is implemented directly by combining the session typecase with the new *session set types*. We develop a type theory for ESP based on session set types for the extended session constructs and prove type soundness and communication safety. The session typecase, inspired by the typecase construct in the λ-calculus (Abadi et al., 1991), enables safety verification of programs with type-driven and dynamic control flows, which is central to the ESP model.

Bisimulations in eventful asynchronous session types. The second part of this article investigates semantic foundations of eventful asynchronously-communicating processes by developing a bisimulation theory. Let us outline some of the key technical ideas informally. As described above, our formalism models asynchronous, order-preserving communication over a binary session connection as *asynchronous session communication* by extending the synchronous session calculus (Takeuchi et al., 1994; Honda et al., 1998) with *message queues*. Unlike previous work (Coppo et al., 2007; Honda et al., 2008; Gay and Vasconcelos, 2010), however, we model both



Local (dashed arrows) **Remote** (solid arrows)

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

$$\overline{s}?(x);Q \mid \overline{s}[\mathtt{i}:w\cdot\vec{h}] \xrightarrow{\tau} Q\{w/x\} \mid \overline{s}[\mathtt{i}:\vec{h}] \qquad \overline{s}[\mathtt{i}:\vec{h}] \xrightarrow{\overline{s}?\langle w'\rangle} \overline{s}[\mathtt{i}:\vec{h}\cdot w']$$

Fig. 1. The transitions involved in the communication of messages between two locations.

input and output queues at *each* session endpoint: an *endpoint configuration* for a session endpoint s, written $s[\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}']$, encapsulates an input queue (`i`) with elements $\vec{h}$ and an output queue (`o`) with $\vec{h}'$. Our model is illustrated in Figure 1, which depicts the two (binary) endpoints of one session connection. A message v is first enqueued by sender process $s!\langle v\rangle;P$ in the output queue at endpoint s. The communication medium (i.e. the connection) is responsible for transporting the message from the sender's locality to the receiver's, formalised as a message transfer from the sender's output queue (at s) to the receiver's input queue (at $\overline{s}$, which is a dual endpoint of s). For the receiver process, the message can only be received after this transfer takes place, since only then can the receiver detect the *arrival* of the message (the lower **Remote** transition in Figure 1) and consume it. Note that enqueuing and dequeuing actions within a location are local to each process and are therefore invisible (τ-actions) to external observers (**Local** in Figure 1).

The induced semantics capture a form of asynchronous observables that were not previously studied. For example, in weak asynchronous bisimilarity ($\approx_a$ in (Honda and Tokoro, 1991; Honda and Yoshida, 1995)), message transport order is not observable, while in our semantics, outputs to the same destination are *not* commutable ($s!\langle v_1\rangle;s!\langle v_2\rangle \not\approx s!\langle v_2\rangle;s!\langle v_1\rangle$) as in the synchronous (Milner et al., 1992) semantics ($\approx_s$ in (Honda and Tokoro, 1991; Honda and Yoshida, 1995)). However, two inputs from different sources are commutable ($s_1?(x);s_2?(y) \approx s_2?(x);s_1?(y)$) since the dequeue action is unobservable, whereas this does not hold in the synchronous semantics ($s_1?(x);s_2?(y) \not\approx_s s_2?(x);s_1?(y)$).

To capture output asynchrony in our semantics, we model `i`/`o`-queues at each session endpoint: for example, with `i`/`o`-queues we have $s_1!\langle v_1\rangle;s_2!\langle v_2\rangle \approx s_2!\langle v_2\rangle;s_1!\langle v_1\rangle$ (we cannot observe the remote process performing the enqueue), while we would have $s_1!\langle v_1\rangle;s_2!\langle v_2\rangle \not\approx s_2!\langle v_2\rangle;s_1!\langle v_1\rangle$ if we were without decoupled `i`/`o`-queues.

The reactive nature of processes driven by asynchronous events (Hu et al., 2010), i.e. the *arrival* of messages at local input queues, introduces further subtleties in the observational laws. In our formalism, the facility for detecting message arrival is modelled by a simple `arrive` predicate: for example, $P = \mathtt{if\ arrive}\ s\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$ reduces to P_1 if the `i`-queue at s contains one or more messages; otherwise P reduces to P_2. By `arrive`, we can observe the movement of an *in-transit* message between two locations. For example, $P \mid s[\mathtt{i}:\varepsilon] \mid \overline{s}[\mathtt{o}:v]$ is not equivalent to $P \mid s[\mathtt{i}:v] \mid \overline{s}[\mathtt{o}:\varepsilon]$ because P can reduce to P_1 in the latter (since v has arrived at the local `i`-queue at s) while it cannot in the former. Note that having local `i`/`o`-queues at

each session endpoint is again essential to precisely capture such a subtle behavioural distinction involving `arrive`.

By representing the major elements of practical communication programming, i.e. asynchrony, ordered and unordered communications and event handling, the induced asynchronous behavioural equivalence can justify various syntactic transformations and programming techniques, including the widely-applied but hitherto unjustified transformation from multithreaded to event-driven implementations (which we refer to as the Lauer-Needham transformation, after (Lauer and Needham, 1979)) of large-scale servers.

Outline. This article presents a full version that combines the work published in two extended abstracts, the theoretical developments from (Hu et al., 2010) and the work in (Kouzapas et al., 2011). The core eventful session calculus used through this article was firstly proposed in (Hu et al., 2010). Apart from including the detailed definitions and explanations, further results, more examples and complete proofs, the contributions of this article are: a new typing system for eventful sessions, which is simpler than the one in (Hu et al., 2010) and a new proof of the type-safety of the calculus; the full development of the bisimulation theory from (Kouzapas et al., 2011) together with more detailed analysis of properties of the behavioural theory; a new bisimulation technique for correctness of encodings of selectors and the Lauer-Needham transformation; and more detailed comparisons with other semantics. Since the article focuses on the type and behavioural theory of eventful computations, the design and implementation of Java with eventful session types presented in (Hu et al., 2010) is only outlined in Section 8.2. We also expand the related work from (Hu et al., 2010; Kouzapas et al., 2011).

Section 2 proposes a new session π-calculus, augmented with event-handling communication primitives. Section 3 defines the type system and proves subject reduction and communication-safety theorems. Section 4 defines a typed bisimulation for eventful sessions, proves that it coincides with the reduction-closed barbed congruence (Honda and Yoshida, 1995) and studies its properties with respect to determinacy and confluence (Philippou and Walker, 1997). Section 5 demonstrates properties of the bisimulation through key examples and compares our bisimulation to other existing theories. Section 6 examines the event selector construct, the central component of most event-driven systems in practice, and its behavioural properties. Section 7 formalises the Lauer-Needham transformation and proves that the transformation is type- and semantics-preserving using the results of the previous sections. Section 8 discusses related work and Section 9 concludes. The Appendices give additional details for definitions and full proofs.

2. A Process Model for Eventful Sessions

We formalise the key concepts of event-driven communications programming. The calculus is the π-calculus with session primitives (Honda et al., 1998; Mostrous and Yoshida, 2009) based on asynchronous communication semantics (Honda and Tokoro, 1991), to which we add minimal extensions for event-driven session programming: `i/o` *message queues*, the *message arrival predicate* and *session typecase*, and *session set types* for session typing. Depending on the context, we can refer to both the calculus (Eventful Session π-calculus) and the associated programming methodology (event-driven session programming) as ESP for short.

```
1.  new sel r in
2.    (register s_1 to r in register s_2 to r in ...register s_n to r in
3.      μX.select x from r in
4.        typecase x of {
5.          ?(U_1);?(U_1);!⟨U_2⟩;∅ : x?(y_1);register x to r in X,
6.          ?(U_1);!⟨U_2⟩;∅ :        x?(y_2);x!⟨v⟩;X
7.        }
8.    )
```

Fig. 2. An eventful session (ESP) implementation of a basic event loop using a session selector macro construct.

2.1. *A Basic Event Loop*

We start with a simple example of event-driven session programming. Event-driven concurrency is best understood in comparison to multithreading. Consider the following multithreaded session server.

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

The server process (on the left), listening on shared channel a, can unfold the n parallel "threads" (processes) required to handle the n client processes (on the right) concurrently. The annotation S, which declares the communication protocol (i.e. session type) that the server follows may be, e.g. $?(U_1);?(U_1);!\langle U_2\rangle;\emptyset$, which says that we receive a sequence of two messages (values v_1 and v_2) of type U_1 before sending a value v of type U_2 in reply.

Figure 2 gives an event-driven server with the same capability to handle concurrent clients according to type S, but without needing to fork a new thread for each client; indeed, the event-driven server comprises a *single* thread, regardless of the number of clients. We will make use of a few high-level macros to focus on the key concepts. The central component is the *event selector*, henceforth referred to as just *selector* for short. A selector offers two main functions. One is to store (*register*) session channels (line 2), which the selector then monitors for event occurrences, i.e. message arrivals. The other is to retrieve (*select*) a stored session channel (line 3) at which a message has arrived and is ready for reading.

To serve the n clients from above, the event-driven server process first creates a new selector `sel` with name r (line 1), and registers sessions $s_1 \ldots s_n$ with the n clients to the selector via r (line 2). (For this introductory example, these sessions are assumed here to be already established; the full selector that also handles session initiation events from an arbitrary number of concurrent clients is left to Section 6.) We then enter the main *event loop* (line 3). In each iteration, the server will select a session s_i that is enabled for reading (waiting until one satisfies this condition), remove it from the internal storage and substitute s_i for x (line 3). Finally, the `typecase` in line 4 tests the selected s_i against the specified session type cases. If it is a newly established session, s_i will have the type specified by the first case in line 5: the server will proceed by receiving the first U_1 message, then re-registering s_i back to the selector to await the arrival of the second message. Otherwise, s_i will correspond to the second case (line 6): the server will receive the second U_1 message and send the U_2; this session is now completed and the server proceeds to the next iteration. In this way, session types are used to determine not only the type of the expected

$$
\begin{array}{llll}
\text{(Identifiers)} & u ::= a,b \mid x,y & k ::= s,\overline{s} \mid x,y & n ::= a,b \mid s,\overline{s} \\
\text{(Values)} & v ::= \mathtt{tt},\mathtt{ff} \mid a,b \mid s,\overline{s} & & \\
\text{(Expressions)} & e ::= v \mid x,y,z \mid n=n \mid \mathtt{arrive}\ u \mid \mathtt{arrive}\ k \mid \mathtt{arrive}\ k\ h & & \\
\text{(Processes)} & P,Q ::= u(x:S).P \mid \overline{u}(x:S).P \mid k!\langle e\rangle;P \mid k?(x);P \mid k\oplus l;P \mid k\&\{l_i:P_i\}_{i\in I} & & \\
 & \quad \mid \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \mid (\nu\, a)P \mid P \mid Q \mid \mathbf{0} \mid \mu X.P \mid X & & \\
 & \quad \mid \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i:S_i):P_i\}_{i\in I} \mid a[\vec{s}] & & \\
 & \quad \mid \overline{a}\langle s\rangle \mid (\nu\, s)P \mid s[S,\mathtt{i}:\vec{h},\mathtt{o}:\vec{h}'] & & \\
\text{(Messages)} & h ::= v \mid l & &
\end{array}
$$

Fig. 3. The syntax of ESP processes.

event, but also the point in the protocol at which the event is occurring, ensuring that the event is handled correctly. The key characteristic of the event-driven server is that, by only selecting sessions with arrived messages, the event loop can safely and efficiently interleave the handling of multiple, concurrent clients in a single thread because delayed message arrival in any one session does not block the execution of any other session.

The simple event loop in this example is an instance of the basic design pattern behind event-driven systems. Having outlined this key interaction structure, the following sections present the core ESP calculus, operational semantics and session type system. The calculus incorporates `typecase` as a primitive. However, the selector object itself (and the `register`/`select` operations) can be encoded into the core calculus using the simple `arrive` primitive; Section 6 shall revisit this example to formalise selector behaviour and prove the soundness of the encoding using the asynchronous session bisimulation that we develop in Section 4. The overall idea is that sessions can be registered and selected via session delegation (passing session endpoints as messages, (Honda et al., 1998)), and the registered sessions can be tested for message arrivals using the non-blocking `arrive` predicate.

2.2. *Syntax of the Eventful Session π-Calculus*

Processes. Figure 3 gives the syntax of ESP processes. We explain the new programming primitives and process terms (`arrive` predicate for non-blocking inspection of messages queues, `typecase` for dynamically matching the type of a session, session endpoint configurations containing localised `i`/`o`-queues) and introduce asynchronous session initiation (cf. synchronous session initiation in preceding works (Honda et al., 1998)).

The syntax defined in the final row of the BNF for (Processes) is called *run-time syntax*. Closed terms which do not contain run-time syntax are called *programs* (see below for the notions of bound and free variables).

Values $(v, v', \ldots)$ include (boolean) constants, shared channels $(a,b,c,\ldots)$ and session channel endpoints $(s,s',\ldots,\overline{s},\overline{s}',\ldots)$. Names $n,\ldots$ include either shared names or session names. A session channel endpoint s designates one endpoint of a session, and $\overline{s}$ denotes the opposite end of the same session. We often shorten "session endpoint channel" (i.e. the programming/run-time entity at a local configuration used to perform session actions) to just "session channel" for brevity, and for convenient presentation of the reduction and typing rules later, we set $\overline{\overline{s}}$ to be s. Branch labels (simply labels) range over $l, l', \ldots$, variables over x,y,z, and recursion vari-

ables over X, Y, Z. Shared channel identifiers $(u, u', \ldots)$ are shared channels and variables; session identifiers $(k, k', \ldots)$ are session channels and variables. A session message h is a value or a label. Expressions e are values, variables, the matching operator $e = e$ on expressions (which includes the matching operator on names when e is a name), and the *message arrival predicates*: `arrive` u checks if any session initiation request message (explained below) has arrived at u, `arrive` k checks if any (intra) session message has arrived at k, and `arrive` k h checks if the first available message at k, if any, is specifically h. We write $\vec{s}$ and $\vec{h}$ for vectors of sessions and messages, respectively, and ε for the empty vector.

The session initiation actions on shared channels are the request $\overline{u}(x : S).P$ and the accept $u(x : S).P$. The annotation S specifies the session type (explained in Section 3.1) that directs how each bound channel x associated with a session on u should be used. On an established session channel k, output $k!\langle e \rangle; P$ sends the value denoted by e through channel k, input $k?(x); P$ receives a value through k, selection $k \oplus l; P$ chooses and sends the label l through k, and branching $k\&\{l_i : P_i\}_{i \in I}$ follows the branch with the label received through k. The $(\nu\ u)P$ binder restricts a shared channel u to the scope of P. The *session typecase* `typecase` k `of` $\{(x_i : S_i) : P_i\}$ attempts to match the *current* (i.e. run-time) session type of channel k against the specified session types S_i, proceeding to the P_i for the first S_i that matches. The `typecase` construct binds x_i in P_i for all $i \in I$.

Our calculus incorporates two forms of asynchronous communication, *asynchronous session initiation* (Kouzapas, 2009) and *asynchronous session communication* (over an established session). The former models the *unordered* transport of session *request messages* to acceptors (servers) listening on a shared channel. We use $\overline{a}\langle s \rangle$ to represent a request message in transit on shared channel a, carrying the initiation request for a (new) session channel s on a. In practical network communications, messages are buffered for reading on arrival at the destination. This mechanism is formalised by introducing a *shared input buffer* $a[\vec{s}]$, which represents an acceptor's input buffer at a containing pending requests for sessions $\vec{s}$. When a program that makes use of a shared channel a starts, the shared input buffer $a[\varepsilon]$ is always present (this will be enforced by the type system).

Communication in an established session is asynchronous but *order-preserving*, as in a TCP session. For this purpose, each session channel s is associated with an *endpoint configuration* (or simply, configuration) $s[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}]$, which encapsulates both input (`i`) and output (`o`) message queues as well as the current run-time session type S (Section 3.1), called the *active type*, of channel s. Sending a message first enqueues it at the source `o`-queue before it is eventually transferred to the destination `i`-queue, signifying the arrival of that message. For both unordered session requests and ordered session messages, decoupling message transmission and arrival captures the intuitive semantics for `arrive`: only messages that are present in the local input queue can be detected, and *not* those still in transit (i.e `arrive` simply checks whether the input queue is non-empty). For brevity, one or more components may be omitted from a configuration when they are irrelevant, e.g. we may use $s[\mathtt{i} : \vec{h}]$ as an abbreviation of $s[S, \mathtt{i} : \vec{h}, \mathtt{o} : \vec{h'}]$ when only the `i`-queue is required. The $(\nu\ s)P$ binder restricts both session channels s *and* $\overline{s}$, i.e. both endpoints of the session, to the scope of P. The process terms specified in Figure 3 that feature s also apply to $\overline{s}$.

The remaining constructs (conditional, parallel composition, agent definition and instantiation, and inaction) are standard; $\mathbf{0}$ is often omitted. The notions of bound and free names

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

Fig. 4. Structural congruence.

and variables are standard (Mostrous and Yoshida, 2009) with extensions to treat $\mathtt{arrive}\ u$, $\mathtt{arrive}\ k$, $\mathtt{arrive}\ k\ h$, $\overline{a}\langle s\rangle$, $a[\vec{s}]$ and $s[\mathtt{i}:\vec{h}, \mathtt{o}:\vec{h'}]$. We write $\mathtt{fn}(P)$ for the set of free names in P. These sets are defined in the expected way, e.g. $\mathtt{fn}(\mathtt{arrive}\ k\ h) = \mathtt{fn}(k) \cup \mathtt{fn}(h)$ and $\mathtt{fn}(s[\mathtt{i}:\vec{h}, \mathtt{o}:\vec{h'}]) = \mathtt{fn}(s) \cup \mathtt{fn}(\vec{h}) \cup \mathtt{fn}(\vec{h'})$.

We then define structural congruence as the smallest congruence on processes generated by the rules in Figure 4. Most of the rules are standard. The last rule in the second column is for garbage collecting empty shared channel buffers and configurations. The last rule in the third column is for unfolding recursive active types within a configuration in an analogous manner to the standard unfolding of recursive processes.

2.3. *Operational Semantics*

The reduction relation on closed terms $\longrightarrow$ captures the communication and event handling dynamics of ESP processes, and updates the active types recorded in configurations as session interactions progress. We use the standard evaluation contexts $E[-]$ defined as:

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

Figure 5 lists the reduction rules. The first three rules are used for session initiation. Rule [Request1] issues a new request for a session of type S via shared channel a. A fresh (i.e. ν-bound) session with endpoints s (acceptor-side) and $\overline{s}$ (requester-side) and the initial configuration at the requester are generated, dispatching the session request message $\overline{a}\langle s\rangle$. [Request2] enqueues the request in the shared input buffer at a. [Accept] dequeues the first session request, substitutes the bound session variable with the s in the request message, and creates the acceptor-side configuration: the new session is now established between the requester and acceptor. Modelling asynchronous session initiation, as well as asynchronous in-session communication, allows us to treat both of the corresponding kinds of session events in ESP programs, i.e. initiation events and in-session message events. The former is equally important as the latter for implementing fully event-driven servers (this can be seen in the encoding of the full event selector construct in Section 6).

The next five rules are for in-session communication. As described earlier, to send a message, rule [Send] enqueues a value in the $\mathtt{o}$-queue of the *local* configuration and removes the output prefix from the current active type, signifying the completion of this action. [Receive] dequeues the first value from the $\mathtt{i}$-queue of the local configuration and again updates the active type accordingly. [Sel] and [Bra] similarly enqueue and dequeue a label, using the label to select the appropriate case in the active type. Note that these four rules manipulate only the local configurations, and output actions are always non-blocking. The actual transmission of a session

$$
\begin{array}{rcll}
\overline{a}(x:S).P & \longrightarrow & (\nu\, s)(P\{\overline{s}/x\} \mid \overline{s}[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid \overline{a}\langle s\rangle) & (s \notin \mathtt{fn}(P)) \quad \text{[Request1]} \\
a[\vec{s}] \mid \overline{a}\langle s\rangle & \longrightarrow & a[\vec{s}\cdot s] & \text{[Request2]} \\
a(x:S).P \mid a[s\cdot\vec{s}] & \longrightarrow & P\{s/x\} \mid s[S, \mathtt{i}:\varepsilon, \mathtt{o}:\varepsilon] \mid a[\vec{s}] & \text{[Accept]} \\
s!\langle v\rangle;P \mid s[!\langle U\rangle;S, \mathtt{o}:\vec{h}] & \longrightarrow & P \mid s[S, \mathtt{o}:\vec{h}\cdot v] & \text{[Send]} \\
s?(x);P \mid s[?(U);S, \mathtt{i}:v\cdot\vec{h}] & \longrightarrow & P\{v/x\} \mid s[S, \mathtt{i}:\vec{h}] & \text{[Receive]} \\
s\oplus l_i;P \mid s[\oplus\{l_j:S_j\}_{j\in J}, \mathtt{o}:\vec{h}] & \longrightarrow & P \mid s[S_i, \mathtt{o}:\vec{h}\cdot l_i] & (i\in J) \quad \text{[Select]} \\
s\&\{l_i:P_i\}_{i\in I} \mid s[\&\{l_j:S_j\}_{j\in J}, \mathtt{i}:l_{i'}\cdot\vec{h}] & \longrightarrow & P_{i'} \mid s[S_{i'}, \mathtt{i}:\vec{h}] & (i'\in J\subseteq I) \quad \text{[Branch]} \\
s[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'] & \longrightarrow & s[\mathtt{o}:\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}'\cdot v] & \text{[Comm]} \\
E[\mathtt{arrive}\ a] \mid a[\vec{s}] & \longrightarrow & E[b] \mid a[\vec{s}] & ((|\vec{s}|\geq 1)\downarrow b) \quad \text{[Arrive-req]} \\
E[\mathtt{arrive}\ s] \mid s[\mathtt{i}:\vec{h}] & \longrightarrow & E[b] \mid s[\mathtt{i}:\vec{h}] & ((|\vec{h}|\geq 1)\downarrow b) \quad \text{[Arrive-sess]} \\
E[\mathtt{arrive}\ s\ h] \mid s[\mathtt{i}:\vec{h}] & \longrightarrow & E[b] \mid s[\mathtt{i}:\vec{h}] & ((\vec{h}=h\cdot\vec{h}')\downarrow b) \quad \text{[Arrive-msg]}
\end{array}
$$

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

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

Fig. 5. Reduction rules.

message is embodied by [Comm], which removes the first message from the o-queue of the source configuration and enqueues it at the end of the i-queue at the target configuration.

Although input actions block in the standard way if no message is available in the local i-queue, a process can avoid input-blocking using the message arrival predicates. [Arriv-req] evaluates `arrive` a to `tt` if the shared input queue is non-empty; similarly `arrive` s in rule [Arrive-sess] checks for a non empty i-queue on channel s. [Arriv-msg] evaluates `arrive` $s\ h$ to `tt` if the i-queue is non-empty and the first message matches h (by name or label equality). The notation $e\downarrow b$ means e evaluates to the boolean value b. Lastly, [Typecase] performs a *dynamic* inspection of the active type of a session. The operation finds the first S_i that can be matched (via subtyping, Section 3.2) to the current S recorded by the configuration of s; the process proceeds as the corresponding P_i with the x_i bound in P_i substituted by s.

The remaining reduction rules are standard. We define $\twoheadrightarrow = (\longrightarrow \cup \equiv)^*$.

Below we illustrate the use of the `arrive` and `typecase` constructs. Let:

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

and session endpoints $B_1 = s[\mathtt{i}:v_1\cdot v_2], B_2 = s[\mathtt{i}:v_2]$ and $B_3 = s[\mathtt{i}:\varepsilon]$.

Process $P \mid B_3$ reduces to process $P_3 \mid B_3$ since the first `arrive` expression would return false on the empty B_3 i.e. $P \mid B_3 \longrightarrow P_3 \mid B_3$.

Process $P \mid B_1$ reduces to process $s?(x);\mathtt{if}\ \mathtt{arrive}\ s\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \mid B_1$, where it consumes

the first message in the i-queue to obtain $(\texttt{if arrive}\ s\ \texttt{then}\ P_1\ \texttt{else}\ P_2)\{v_1/x\} \mid B_2$. Then the third reduction proceeds to process $P_1\{v_1/x\} \mid B_2$ as:

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

On the other hand, the process $P \mid B_2$ returns true on the first `arrive`-inspection and false on the second one, yielding a reduction of the form:

$$P \mid B_2 \longrightarrow s?(x);\texttt{if arrive}\ s\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid B_2 \longrightarrow\longrightarrow P_2\{v_2/x\} \mid B_3$$

Next let $Q = \texttt{typecase}\ s\ \texttt{of}\ \{(x_1 : S_1) : Q_1, (x_2 : S_2) : Q_2\}$ and $B_1' = s[S_1]$ and $B_2' = s[S_2]$. In process $Q \mid B_1'$ the `typecase` operation matches the type S_1 in B_1' with the $(x_1 : S_1) : Q_1$ case in the `typecase` body and reduces to $Q_1\{s/x_1\} \mid B_1'$ with the bound variable x_1 substituted by session channel s. Similarly for process $Q \mid B_2'$ reduces to $Q_2\{s/x_2\} \mid B_2'$.

3. Types for Eventful Session Processes

This section presents a session typing discipline for ESP processes and establishes some of the key theoretical results of this paper: properties of subtyping (Proposition 3.1), subject reduction (Theorem 3.1), and communication and event-handling safety (Theorem 3.2). We focus on the new typing rules for the event handling primitives, and the type system for run-time syntax which is simplified from (Hu et al., 2010).

3.1. *Type Syntax*

The type syntax is an extension of the standard session types from (Honda et al., 1998).

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

The shared types U include booleans `bool`, and the IO-types (Pierce and Sangiorgi, 1996; Honda and Yoshida, 2007) $\texttt{i}\langle S\rangle$ (accept, i.e. input) and $\texttt{o}\langle S\rangle$ (request, i.e. output) for the shared channels via which sessions of type S are initiated. In the present work, IO-types (often called client/server types) are used to control locality (shared channel buffers are located only at the server side) and the associated typed transitions, playing a central role in our behavioural theory. In the session types S, the output type $!\langle T\rangle;S$ represents sending a value of type T, then continuing as S; dually for input type $?(T);S$. Selection type $\oplus\{l_i : S_i\}_{i\in I}$ describes the selection of one of the labels l_i, then continuing as S_i. Branching type $\&\{l_i : S_i\}_{i\in I}$ waits with I options, behaving as type S_i if the label l_i is selected. End type `end` represents session completion and is often omitted. For recursive types $\mu X.S$, we assume the type variables are guarded in the standard way (Pierce, 2002, § 21.8). Lastly, we introduce the *session set type* $\{S_i\}_{i\in I}$, which represents a set of possible behaviours designated by the S_i. Session set types are used to type the `typecase` construct.

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

Fig. 6. The generating function for the session subtyping relation.

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

Fig. 7. Session type duality.

3.2. *Session Subtyping*

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

Formally, the subtyping relation is defined on the set of all closed and contractive types $\mathscr{T}$ (Pierce, 2002) as follows: for $T',T\in\mathscr{T}$, T' is a subtype of T, written $T'\leq T$, if (T',T) is in the largest fixed point of the monotone function $\mathscr{F}:\mathscr{P}(\mathscr{T}\times\mathscr{T})\to\mathscr{P}(\mathscr{T}\times\mathscr{T})$ given in Figure 6. Line 2 is standard: $\mathtt{i}\langle S\rangle$ and $\mathtt{o}\langle S\rangle$ are invariant on S since this notation supports both S and $\overline{S}$ (see duality below). Line 7 gives the standard rules for recursion. In Lines 3 and 4, the linear output (resp. input) is contravariant (resp. covariant) on the message type that follows (Mostrous and Yoshida, 2009). In Line 5, a select that requires support for more labels means fewer peers can be safely composed; dually for branching in Line 6. The ordering of set types in line 8, says that if every element in the set type $\{S'_j\}_{j\in J}$ has a subtype in $\{S_i\}_{i\in I}$, then the latter is at least as composable as the former. The final clause states that singleton set types are transparent (i.e. the enclosed type can be "unwrapped") up-to subtyping.

We now clarify the semantics of $\leq$ through *duality*. Figure 7 defines the dual of S, denoted $\overline{S}$, in the standard way with a straightforward extension for session set types.
The following Lemma and Proposition relate subtyping with the notions of duality and composability:

Lemma 3.1. $S_1\leq S_2$ iff $\overline{S_2}\geq\overline{S_1}$.

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

We now define the set of *composable* types of a session type S as follows:

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

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

Subtyping can be completely characterised by composability.

Proposition 3.1 (Subtyping Properties). (1) $\leq$ is a preorder; (2) $S_1 \leq S_2$ if and only if $\mathtt{comp}(S_2) \subseteq \mathtt{comp}(S_1)$.

Proof. (1) is standard, while (2) uses Lemma 3.1. For both, see Appendix A for details. □

3.3. *Type System for Programs*

We first define the typing judgements for programs (i.e. closed processes that do not contain run-time syntax, Section 2) and expressions.

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

with

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

The *shared environment* Γ is a map from variables and shared channels to constant types and shared channel types. Recursion variables are also recorded to type recursive processes. The *linear environment* Δ is a map from variables and session channels to session types. The linear environment is also used to note the shared channels for which a shared channel buffer is always present. A linear environment that assigns each session to the `end` type is called complete. Then we read the program typing judgement as: program P is typed under shared environment Γ and uses channels according to linear environment Δ. In the expression judgement, expression e has type T under environments Γ and Δ. We may omit Δ from the latter if it is clear from the context. Our use of the $\cdot$ operator between maps and other entities is defined when the argument domains are disjoint. For example, $\Delta \cdot k : S$ is defined if $k \notin \mathtt{dom}(\Delta)$. The notation $\Delta \cdot a$ is shorthand for concatenating the Δ map and a map from a to some dummy value.

Figure 8 gives the typing rules for programs. We assume the environment of the conclusion is always defined. The system is similar to (Hu et al., 2010; Bettini et al., 2008). Rule (SChan) types shared channels in accordance with the environment Γ. A shared input type can be used as a shared output type by rule (SChan'). Rules (Bool) and (Match) assign the boolean type to the boolean constants `tt`, `ff` and value matching expressions (similarly for other boolean expressions, e.g. e `and` e). Rules (AReq), (AMsg), (AVal) and (ALab) type the `arrive` predicates with the boolean type; (AReq) checks that u is indeed a shared channel, and (AVal) checks that the specified v corresponds to the expected message type on that session.

To type the session initiation actions, we define S^{-} to be the subset of session types S where set types only occur in the object position of the input and output types.

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

For example, $!\langle T \rangle ; \{S_i\}_{i \in I}$ is not within this subset. The S^{-} type is used to restrict the session

$$\Gamma \cdot u : U \vdash u : U \quad \text{(SChan)} \qquad \Gamma \cdot u : \mathtt{i}\langle S\rangle \vdash u : \mathtt{o}\langle S\rangle \quad \text{(SChan')} \qquad \frac{\Gamma \vdash n : T \vee \Delta = \Delta' \cdot n : T}{\Gamma, \Delta \vdash n : T}\ \text{(Name)}$$

$$\Gamma \vdash \mathtt{tt}, \mathtt{ff} : \mathtt{bool} \quad \text{(Bool)} \qquad \frac{\Gamma, \Delta \vdash e_i : T_i \quad i \in \{1,2\}}{\Gamma, \Delta \vdash e_1 = e_2 : \mathtt{bool}}\ \text{(Match)}$$

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Typing rules for programs.

types allowed in the session initiation prefix, in particular to prevent set types from occurring as an active type in a session queue endpoint.

Rules (Req) and (Acc) check if the shared environment maps the shared channel to the output (resp. input) shared channel type, consistent with the S^{-} annotation and the usage of the bound session variable. The initiation annotation is restricted to ensure that the active type of the session at run-time has an S^{-} shape (see [Request1] and [Accept] in Figure 5), so that the execution of `typecase` can resolve the type of the session to a specific case.

Rules (Send) and (Recv) require that a value being sent (resp. received) on a session channel is typed according to the shared environment Γ. Rule (Deleg) types the delegation of session channels. It requires the session channel being sent to be present with the correct type in the linear environment Δ before it is sent. Rule (Srecv) types the receiving of a delegated session

channel. The channel should be present in the linear environment Δ after being received. Rules (Sel) and (Bra) type the selection and branching actions respectively.

Rule (Conc) concatenates the disjoint environment typings of parallel processes. Rule (If) checks for a boolean condition and for equality of the session types in the two cases. Rule (CRes) restricts a shared channel and its buffer by removing its typing from both the shared and linear environments. Rule (EBuff) types empty shared channel buffers by recording the shared channel in the linear environment. Rules (Rec) and (Var) type recursive processes. The empty process is typed with a *complete* linear environment (where all channels are typed with `end`) by rule (Inact). Lastly, rule (Typecase) types the `typecase`: the usage of the target session channel in each of the sub-processes is collected into a set of possible behaviours represented by the set type.

3.4. *Type System for Run-time Syntax*

This section extends the type system for programs (Section 3.3) to the full type system for run-time syntax. Our new system significantly simplifies that in (Hu et al., 2010) by adapting the approach developed in (Bettini et al., 2008). First we define an additional type category T, which includes session types and *message types*:

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

Message types abstract from the values stored in queues, and are used for typing endpoint configurations. A message type M is either an input $M_{\mathtt{i}}$ or an output $M_{\mathtt{o}}$ queue abstraction. Incoming messages and branch labels enqueued in an `i`-queue are recorded as $?(T)$ and $\&l$ respectively. Similarly, $!\langle T\rangle$ and $\oplus l$ for outgoing messages and select labels in an `o`-queue. $\emptyset$ is used to type empty queues. We then extend the linear environment to include the $s : \mathsf{T}[S]$ type, where the $[S]$ part denotes the run-time type of the s endpoint configuration. Note that the $\mathsf{T}[S]$ type is present if the s endpoint is present in the typed process. A configuration by itself is typed as $M[S]$, where M denotes the enqueued message types and S is the active type. The $M[S]$ type is composed with the S' type of a session via the operator $*$ as explained next. The linear environment Δ is now given by the extended grammar:

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

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

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

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

Operator $*$ is used to reconstruct the overall session type for a session endpoint from the typing of the session channel $s : S$ and the associated message type $s : M$.

Outputs on an endpoint s enqueue messages in the `o`-queue, before they are delivered to the opposite endpoint (i.e. the opposite `i`-queue). For this case, $*$ is defined to "undo" the outputs by using the $M_{\mathtt{o}}$ from the `o`-queue of the s configuration to restore the session type S of session

channel s. On the other hand, messages in the $\mathtt{i}$-queue of endpoint s have already been delivered from the opposite endpoint and are ready for reading, so the definition of $*$ consumes the message type $M_{\mathtt{i}}$ of the s configuration by truncating the corresponding prefix of the session type S of session channel s.

With this intuition for the $*$ operator, the definition can be clarified as follows. In the cases where we combine a session type S with an output message type $!\langle T\rangle;M_{\mathtt{o}}$ or $\oplus l_k;M_{\mathtt{o}}$, we concatenate the message type prefix and the session type S to obtain $!\langle T\rangle;S$ and $\oplus\{l_i : S_i\}_{i\in I}$ for all $k \in I$, and continue this concatenation inductively. Note that in the case of the selection type, the result is non-deterministic. This follows the intuition that session types can be subsumed up-to subtyping, e.g. $[l_1] \oplus * S_1 = \oplus\{l_1 : S_1\}$ or $[l_1] \oplus * S_2 = \oplus\{l_1 : S_1, l_2 : S_2\}$ where $\oplus\{l_1 : S_1\}$ is a subtype of $\oplus\{l_1 : S_1, l_2 : S_2\}$. For the case of input message types, we expect both the session type S and the message type $M_{\mathtt{i}}$ to have the matching prefixes. The $*$ operator then consumes the prefixes, again proceeding inductively.

Lastly, we overload the $*$ operator for composing extended linear environments. We require that the two linear environments that are being composed have disjoint domains, except in the case where one environment records a session type S for the common session name s and the other records a message type M for s. In any other cases, the $*$ operator is undefined. For example, consider linear environments $\Delta_1 = \{s_1 : S_1 \cdot s_2 :?(T_2);S_2\}$ and $\Delta_2 = \{s_1 : [S_1']!\langle T_1\rangle \cdot s_2 :?(T_2)[S_2']\}$. Then $\Delta_1 * \Delta_2 = \{s_1 :!\langle T_1\rangle;S_1[S_1'] \cdot s_2 : S_2[S_2']\}$. If we have a linear environment $\Delta_3 = \{s_1 : S_1'\}$, then the operation $\Delta_1 * \Delta_3$ is undefined.

We now present the typing rules for the run-time type system. Most of the typing rules are directly inherited from the program type system (Figure 8). Figure 9 lists the additional rules for run-time syntax, and we replace the program typing rule (Conc) with (QConc) (explained below). An endpoint configuration for s is typed with the message type M and its active type $[S]$. Rules (InQ) and (OutQ) respectively type the empty $\mathtt{i}$- and $\mathtt{o}$-queues with the empty message type. Rule (RcvQ) takes the typing of $\mathtt{i}$-queue tail and prefixes the message type for the head element. Rule (BraQ) is similar, but handles the branching by prefixing the label message type. Likewise, rules (OutQ) and (SelQ) type $\mathtt{o}$-queues, but build the type in the reverse direction. Rules (SRcvQ) and (DelQ) deal with the typing of delegated sessions, but are otherwise similar to (RcvQ) and (SndQ). Regarding parallel composition, rule [Conc] is replaced by rule (QConc), which uses $*$ for combining types of the parallel components. As explained above, $*$ collects together the type information of an endpoint in a process and the messages in the associated configuration. Recall that $*$ is not defined for other pairings of s in both environments (i.e. if a common s is used for two different sessions, or two configurations for s), and note that when $\mathtt{dom}(\Delta_1) \cap \mathtt{dom}(\Delta_2) = \emptyset$, it coincides with the $\cdot$ operator in (Conc). Rule (SRes) types session restriction by asserting that the session endpoints have dual typings in the standard manner, and checking for the presence of the expected configurations. Rule (Buff) is used in conjunction with (EBuff) (from Figure 8) to type non-empty shared channel buffers by recording the session channels encapsulated by the enqueued request messages in the linear environment. Finally, (ReqM) types in-transit session request messages.

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

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

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

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

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

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

Fig. 9. Additional and extended typing rules for the run-time processes.

3.5. *Subject Reduction*

In this subsection we prove that our typing theory is sound through a subject reduction theorem. We then use subject reduction to prove that a typed process would never reduce to an error state.

We begin by introducing the preliminary definitions.

Definition 3.1 (Well-configured Linear Environments). We say that Δ is *well-configured* if whenever $s \in \mathtt{dom}(\Delta)$, then either $\Delta(s) = S$ with $\Delta(\overline{s}) = \overline{S}$, or $\Delta(s) = S\,[S_1^-]$ with $\Delta(\overline{s}) = \overline{S}\,[S_2^-]$.

Definition 3.2 (Linear Environment Reduction). We define:

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

Theorem 3.1 (Subject Congruence and Reduction).

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

Proof. The proof for (i) is standard. For (ii), we prove by induction on the reduction relation. Note that we need to use the typing system for the run-time processes. For the detailed proof, see Appendix B. □

We now prove communication safety. We say an *s-redex* is a parallel composition of two *s*-processes that has one of the following shapes:

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

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

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

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

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

Fig. 10. Labelled transition system.

(a) $s!\langle v\rangle;P \mid s[!\langle T\rangle;S]$ (b) $s\oplus l_i;P \mid s[\oplus\{l_j:S_j\}_{j\in J}]$ with $i\in J$
(c) $s?(x);P \mid s[?(T);S,\mathtt{i}:v\cdot\vec{h}]$ (d) $s\&\{l_i:P_i\}_{i\in I} \mid s[\&\{l_j:S_j\}_{j\in J},\mathtt{i}:l_{i'}\cdot\vec{h}]$ with $i'\in J\subseteq I$
(e) $s[\mathtt{o}:v\cdot\vec{h}] \mid \overline{s}[\mathtt{i}:\vec{h}']$
(f) $E[\mathtt{arrive}\ s\ v] \mid s[?(U);S,\mathtt{i}:\vec{h}]$ with v of type U, and $\vec{h}=\varepsilon$ or $\vec{h}=v'\cdot\vec{h}'$, v' of type U
(g) $E[\mathtt{arrive}\ s\ l_i] \mid s[\&\{l_j:S_j\}_{j\in J},\mathtt{i}:\vec{h}]$ with $i\in J$, and $\vec{h}=\varepsilon$ or $\vec{h}=l_{i'}\cdot\vec{h}'$, $l_{i'}\in J$
(h) $\mathtt{typecase}\ s\ \mathtt{of}\ \{(x_i:S_i):P_i\} \mid s[S]$ with $\exists i\in I.\ S_i\leq S$

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

A process P is an *error* if up to structural congruence (following (Honda et al., 2008, § 5)), P contains two s-processes which do not form an s-redex, or an expression in P contains a type error in the standard sense. As a corollary of subject reduction (Theorem 3.1), we obtain:

Theorem 3.2 (Communication and Event-Handling Safety). If P is a well-typed program, then $\Gamma\vdash P\triangleright\emptyset$, and P never reduces to an error.

Proof. Assuming the reduction of a typable process to an error (page 16), we show that the error is not typable, thus leading to a contradiction. See Appendix B for details. □

4. Asynchronous Session Bisimulation and its Properties

This section presents a behavioural theory for the ESP-calculus. We first define a typed bisimulation theory, and then show that the typed bisimulation coincides with typed reduction-closed barbed congruence (Honda and Yoshida, 1995). We then use the bisimulation to study the confluence and determinacy properties of the session types. The results of this section are used to study properties of event-driven programming in the rest of the article.

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

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

where the labels denote, respectively, the session accept, request and bound request, input, output, bound output, branching, selection and the τ-action. Writing $\mathtt{subj}(\ell)$ (resp. $\mathtt{obj}(\ell)$) denotes the set of free subjects (resp. object) in ℓ; and $\mathtt{fn}(\ell)$ (resp. $\mathtt{bn}(\ell)$) denotes the set of free (resp. bound) names in ℓ. Moreover $\mathtt{n}(\ell)$ defines the union $\mathtt{fn}(\ell)\cup\mathtt{bn}(\ell)$:

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

Definition 4.1 (Context). A context is defined as

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

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

The symmetric operator $\ell \asymp \ell'$ on labels that denotes that ℓ is a dual of ℓ', is defined as:

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

Figure 10 gives the untyped labelled transition system (LTS). $\langle$Acc$\rangle$/$\langle$Req$\rangle$ are for the session initialisation. The next four rules $\langle$In$\rangle$/$\langle$Out$\rangle$/$\langle$Bra$\rangle$/$\langle$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 $\langle$Local$\rangle$. Other compositional rules are standard.

The typed LTS builds upon the untyped LTS. 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 11. A transition $(\Gamma,\Delta) \xrightarrow{\ell} (\Gamma',\Delta')$ means that an environment (Γ,Δ) allows an action ℓ to take place, and the resulting environment is (Γ',Δ'), constraining process transitions through the linear and shared environments. This constraint is at the heart of our typed LTS, accurately capturing interactions in the presence of sessions and local buffers.

The first rule says that reception of a message via a is possible only when a is input-typed (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 o-mode and no queue exists. Similarly for a bound output action. The two session output rules ($\ell = s!\langle v\rangle$ and $s!(a)$) are the standard value output and a scope opening rule. The next is for value input. Label input and output are defined similarly. Note that we send and receive only a shared channel which has o-mode. This is because a new accept should not be

$$
\begin{array}{rll}
\Gamma(a) = \mathtt{i}\langle S\rangle, a \in \Delta, s \text{ fresh} & \text{implies} & (\Gamma,\Delta) \xrightarrow{a\langle s\rangle} (\Gamma,\Delta\cdot s : \overline{S}) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta & \text{implies} & (\Gamma,\Delta) \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta) \\
\Gamma(a) = \mathtt{o}\langle S\rangle, a \notin \Delta, s \text{ fresh} & \text{implies} & (\Gamma,\Delta) \xrightarrow{\overline{a}\langle s\rangle} (\Gamma,\Delta\cdot s : S) \\
\Gamma \vdash v : U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathtt{dom}(\Delta) & \text{implies} & (\Gamma,\Delta\cdot s : !\langle U\rangle;S) \xrightarrow{s!\langle v\rangle} (\Gamma,\Delta\cdot s : S) \\
\overline{s} \notin \mathtt{dom}(\Delta) & \text{implies} & (\Gamma,\Delta\cdot s : !\langle \mathtt{o}\langle S'\rangle\rangle;S) \xrightarrow{s!\langle a\rangle} (\Gamma\cdot a : \mathtt{o}\langle S'\rangle,\Delta\cdot s : S) \\
\Gamma \vdash v : U \text{ and } U \neq \mathtt{i}\langle S'\rangle \text{ and } \overline{s} \notin \mathtt{dom}(\Delta) & \text{implies} & (\Gamma,\Delta\cdot s : ?(U);S) \xrightarrow{s?\langle v\rangle} (\Gamma,\Delta\cdot s : S) \\
\overline{s} \notin \mathtt{dom}(\Delta) & \text{implies} & (\Gamma,\Delta\cdot s : \oplus\{l_i : S_i\}_{i\in I}) \xrightarrow{s\oplus l_k} (\Gamma,\Delta\cdot s : S_k) \\
\overline{s} \notin \mathtt{dom}(\Delta) & \text{implies} & (\Gamma,\Delta\cdot s : \&\{l_i : S_i\}_{i\in I}) \xrightarrow{s\&l_k} (\Gamma,\Delta\cdot s : S_k) \\
\Delta \longrightarrow \Delta' \vee \Delta = \Delta' & \text{implies} & (\Gamma,\Delta) \xrightarrow{\tau} (\Gamma,\Delta')
\end{array}
$$

Fig. 11. Labelled transition rules for environments.

created without its queue in the same location. The final rule ($\ell = \tau$) follows the reduction rules defined before Theorem 3.1. Environments can also be observed on the silent action without changing their state. The LTS omits delegation since it is not necessary in the present inquiry (due to the definition of localisation, see the following paragraph).

Definition 4.2 (Typed Transition). The *typed transition* relation is defined as $\Gamma_1 \vdash P_1 \triangleright \Delta_1 \xrightarrow{\ell} \Gamma_2 \vdash P_2 \triangleright \Delta_2$ if (1) $P_1 \xrightarrow{\ell} P_2$; and (2) $(\Gamma_1,\Delta_1) \xrightarrow{\ell} (\Gamma_2,\Delta_2)$ with $\Gamma_i \vdash P_i \triangleright \Delta_i$.

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

Write $\leftrightharpoons$ for a symmetric and transitive closure of $\longrightarrow$ over linear environments.

Definition 4.3 (Typed Relation). We say a relation $\mathscr{R}$ on typed processes is a *typed relation* if, whenever it relates two typed processes, $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ we have $\Delta_1 \leftrightharpoons \Delta_2$.

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

Localisation and bisimulation. Our bisimulation is a typed relation over processes which are *localised*, in the sense that they are equipped with all necessary local queues. As we discussed in Introduction, this is essential to accurately capture the semantics of asynchronous output as well as the `arrive` primitive.

First, we say an environment Δ is *delegation-free* if it contains types which are generated by deleting S from value type T in the syntax of types defined in § 3 (i.e. neither $!(S);S'$ nor $?(S);S'$ appears in Δ). Similarly for Γ. Note that a process under a delegation free environment can perform delegations at hidden channels by rule $\langle$Local$\rangle$).

Definition 4.4 (Localisation). Let P be closed and $\Gamma \vdash P \triangleright \Delta$ where Γ and Δ are delegation-free. Then we say P is *localised* w.r.t. Γ,Δ if

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

Being localised means that a process owns all necessary queues as specified in environments. We further 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, while $s?(x);s!\langle x+1\rangle;\mathbf{0} \mid s[\mathtt{i}:\vec{h}_1,\mathtt{o}:\vec{h}_2]$ 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 always stay. We call the typed relation $\mathscr{R}$ a congruence if $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} \Gamma \vdash P_2 \triangleright \Delta_2$ implies that $\forall C, \Gamma \vdash C[P_1] \triangleright \Delta_1' \mathscr{R} \Gamma \vdash C[P_2] \triangleright \Delta_2'$ for some Δ_1' and Δ_2'. We can now introduce reduction-closed barbed congruence and asynchronous bisimilarity.

Definition 4.5 (Reduction-Closed Barbed Congruence). We write

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

$\Gamma \vdash P \triangleright \Delta \Downarrow n$ means $\exists P'.P \twoheadrightarrow P'$ and $\Gamma \vdash P' \triangleright \Delta \downarrow n$. A typed relation $\mathscr{R}$ is a *reduction-closed barbed congruence* if it is a congruence and satisfies the following conditions for each $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$ whenever $\Gamma \vdash P_1 \triangleright \Delta_1, \Gamma \vdash P_2 \triangleright \Delta_2$ are localised:

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

The maximum reduction-closed barbed congruence (Honda and Yoshida, 1995), is denoted by $\cong$.

Definition 4.6 (Asynchronous Session Bisimulation). A typed relation $\mathscr{R}$ over localised processes is a *weak asynchronous session bisimulation* or often a *bisimulation* if, whenever $\Gamma \vdash P_1 \triangleright \Delta_1 \mathscr{R} P_2 \triangleright \Delta_2$, it holds:

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

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

We extend $\approx$ to possibly non-localised closed terms by relating them when their minimal localisations are related by $\approx$ (given $\Gamma \vdash P \triangleright \Delta$, its *minimal localisation* adds empty queues to P for the input shared channels in Γ and session channels in Δ that are missing their queues). Further $\approx$ is extended to open terms in the standard way (Honda and Yoshida, 1995).

4.2. *Properties of Asynchronous Session Bisimilarity*

This subsection studies central properties of asynchronous session semantics.

Characterisation of reduction-closed barbed congruence. We first show that the bisimilarity coincides with the naturally defined reduction-closed barbed congruence (Honda and Yoshida, 1995).

Theorem 4.1 (Soundness and Completeness). $\approx = \cong$.

Proof. To prove the soundness direction ($\approx \subset \cong$) we show that $\approx$ is a congruence. The most difficult case is a closure under parallel composition, which requires to check the side condition $\Delta_1' \leftrightharpoons \Delta_2'$ for each case. The input and output prefix congruence are straightforward to prove since we are interested only in closed terms. The completeness ($\cong \subset \approx$) is harder, and we follow the proof method introduced in (Hennessy, 2007, § 2.6). Specifically, we prove that every external action is definable by composing with a testing process $T\langle N, \mathrm{succ}, \ell\rangle$. Process $T\langle N, \mathrm{succ}, \ell\rangle$ has a fresh name succ which allows us to detect the external action ℓ, based on the reduction closure of the reduction congruence. In our proof we extended the testing process T in (Hennessy, 2007, § 2.6) with sessions and session queues. Then we show that processes with the same observables are congruent under the same testing process. See Appendix C for the detailed proof. □

Asynchrony, session determinacy and confluence. We study the properties of our asynchronous session bisimulations based on the notions of (Philippou and Walker, 1997).

Definition 4.7. Let us call ℓ an

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

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

Lemma 4.1 (Input and Output Asynchrony). Suppose $\Gamma \vdash P \triangleright \Delta \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta'$.

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

Proof. The proof is done by induction on the length of the silent transition. For the proof we utilise an intermediate result (see Lemma C.5) where we permute a single hidden action and an input (resp. output) action. For the full proof, see Appendix C.1.1. □

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

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

Due to the asynchronous nature of the typed LTS, it is always possible to observe an input action *before* a series of silent (tau) actions. Hence we have:

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

An example for an output action would be:

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

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

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

We follow the work on the confluence for the π-calculus in (Philippou and Walker, 1997), to define determinacy and confluence. Below and henceforth we often omit the environments in typed transitions. We first define determinacy on typed transitions:

Definition 4.8 (Determinacy). We say $\Gamma' \vdash Q \triangleright \Delta'$ is a *derivative* of $\Gamma \vdash P \triangleright \Delta$ if there exists $\vec{\ell}$ such that $\Gamma \vdash P \triangleright \Delta \stackrel{\vec{\ell}}{\Longrightarrow} \Gamma' \vdash Q \triangleright \Delta'$. We say $\Gamma \vdash P \triangleright \Delta$ is *determinate* if for each derivative $\Gamma' \vdash Q \triangleright \Delta'$ of P and action ℓ, if $\Gamma' \vdash Q \triangleright \Delta' \xrightarrow{\ell} Q' \triangleright \Delta_1$ and $\Gamma' \vdash Q \triangleright \Delta' \stackrel{\widehat{\ell}}{\Longrightarrow} Q'' \triangleright \Delta_2$ then $Q' \approx Q''$.

In the following, any transition on session channels that does not involve `arrive` nor an accept action is called *a session transition*, and a process that only exhibits traces of session transitions is called *session determinate*.

Definition 4.9 (Session Determinacy). Let us write $P \xrightarrow{\ell}_s Q$ if $P \xrightarrow{\ell} Q$ and $\ell = \tau$, then $P \xrightarrow{\ell} Q$ is generated without using [Request1], [Request2], [Accept], [Arrive-req], [Arrive-sess] nor [Arrive-msg] in Figure 5 (i.e. a communication is performed without arrival predicates or accept actions). We extend the definition to $\stackrel{\vec{\ell}}{\Longrightarrow}_s$, $\stackrel{\widehat{\ell}}{\Longrightarrow}_s$ and other labelled relations.

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.

The following definitions follow (Philippou and Walker, 1997) and are useful to define confluence:

Definition 4.10. Let ℓ_1, ℓ_2 be two labels.

— We define $\ell_1 \lfloor \ell_2$ as (1) $\overline{a}\langle s \rangle$ if $\ell_1 = \overline{a}(s')$ and $s' \in \mathtt{bn}(\ell_2)$; (2) $s!\langle s' \rangle$ if $\ell_1 = s!(s')$ and $s' \in \mathtt{bn}(\ell_2)$; (3) $s!\langle a \rangle$ if $\ell_1 = s!(a)$ and $a \in \mathtt{bn}(\ell_2)$; and otherwise ℓ_1.
— We write that $\ell_1 \bowtie \ell_2$ when $\ell_1 \neq \ell_2$ and if ℓ_1, ℓ_2 are input actions then $\mathtt{subj}(\ell_1) \neq \mathtt{subj}(\ell_2)$.

The definition of the $\ell_1 \lfloor \ell_2$ is important to define how the second action is affected by the first action in the LTS relations. For example, consider transitions $\Gamma \vdash P \triangleright \Delta \xrightarrow{s_1!(a)} \Gamma \vdash P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \xrightarrow{s_2!(a)} \Gamma \vdash P_2 \triangleright \Delta_2$. If we observe an output of the shared name a through session channel s_1 after process $\Gamma \vdash P_2 \triangleright \Delta_2$ emits $s_2!(a)$, then we would observe the action $s_1!(a) \lfloor s_2!(a) = s_1!\langle a \rangle$ since a was already scope-extruded by the first action $s_2!(a)$ from P. On the other hand, if we observe action to s_2 with value a after $s_1!(a)$, we obtain the transition $\Gamma \vdash P_1 \triangleright \Delta_1 \xrightarrow{s_2!(a) \lfloor s_1!(a)} \Gamma \vdash P_1' \triangleright \Delta_1'$ with $s_2!(a) \lfloor s_1!(a) = s_2!\langle a \rangle$. In this way, operator $\bowtie$ is used for the confluence definition and relates two actions.

(Milner, 1980) discusses the property of confluence that "of any two possible actions, the occurrence of one will never preclude the other". This is captured by the following confluence definition for the π-calculus from (Philippou and Walker, 1997).

Definition 4.11 (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 $\Gamma \vdash Q \triangleright \Delta \xrightarrow{\ell} Q_1 \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta \stackrel{\ell}{\Longrightarrow} Q_2 \triangleright \Delta_2$, then $\Gamma \vdash Q_1 \triangleright \Delta_1 \Longrightarrow Q_1' \triangleright \Delta_1'$ and $\Gamma \vdash Q_2 \triangleright \Delta_2 \Longrightarrow Q_2' \triangleright \Delta_2'$ with $Q_1' \approx Q_2'$; and (ii) if $\Gamma \vdash Q \triangleright \Delta \xrightarrow{\ell_1} Q_1 \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta \stackrel{\ell_2}{\Longrightarrow} Q_2 \triangleright \Delta_2$, then $\Gamma \vdash Q_1 \triangleright \Delta_1 \stackrel{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} Q_1' \triangleright \Delta_1'$ and $\Gamma \vdash Q_2 \triangleright \Delta_2 \stackrel{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} Q_2' \triangleright \Delta_2'$ with $Q_1' \approx Q_2'$.

The next lemmas are used to prove Theorem 4.2. The first lemma states that session determinate processes are semantically invariant (i.e up-to typed bisimulation) under silent actions:

Lemma 4.2. Let P be session determinate and $\Gamma \vdash P \triangleright \Delta \Longrightarrow Q \triangleright \Delta'$. Then $P \approx Q$.

Proof. The proof considers the induction on the length of $\Longrightarrow_s$-transition. For the proof, see Appendix C.1.2. □

Lemma 4.3. Assume typable, localised P and actions ℓ_1, ℓ_2 such that $\mathtt{subj}(\ell_1), \mathtt{subj}(\ell_2)$ are session names and $\ell_1 \bowtie \ell_2$. If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_2} P_2 \triangleright \Delta_2$ then $\Gamma \vdash P_1 \triangleright \Delta \xrightarrow{\ell_2 \lfloor \ell_1} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta \xrightarrow{\ell_1 \lfloor \ell_2} P' \triangleright \Delta'$

Proof. The proof uses the fact that ℓ_1 and ℓ_2 have different subjects and are observed from session endpoint configurations. For proof, see Appendix C.1.3. □

We show that session transitions are determinate transitions.

Lemma 4.4. Let P be session determinate. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} P' \triangleright \Delta'$ and $\Gamma \vdash P \triangleright \Delta \stackrel{\widehat{\ell}}{\Longrightarrow} P'' \triangleright \Delta''$ then $P' \approx P''$

Proof. There are the following two cases:
Case: τ:
Follow Lemma 4.2 to obtain $P \approx P'$ and $P \approx P''$. The result is then immediate.
Case: ℓ:
Suppose that $P \xrightarrow{\ell}_s P'$ and $P \stackrel{\ell}{\Longrightarrow}_s P''$ implies $P \Longrightarrow_s P_1 \xrightarrow{\ell}_s P_2 \Longrightarrow_s P''$. From Lemma 4.2, we can conclude that $P \approx P_1$, and because of the definition of the bisimulation, we have $P' \approx P_2$. To complete the case, we use Lemma 4.2 once more, which derives $P' \approx P''$, as required. □

We show that session transitions are confluent transitions.

Lemma 4.5. Let P be session determinate and $\ell_1 \bowtie \ell_2$. Then if $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell_1} P_1 \triangleright \Delta_1$ and $\Gamma \vdash P \triangleright \Delta \stackrel{\ell_2}{\Longrightarrow} P_2 \triangleright \Delta_2$, then $\Gamma \vdash P_1 \triangleright \Delta_1 \stackrel{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} P' \triangleright \Delta'$ and $\Gamma \vdash P_2 \triangleright \Delta_2 \stackrel{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} P'' \triangleright \Delta''$ and $P' \approx P''$

Proof. By a case analysis on the labels ℓ_1 and ℓ_2. The case analysis follows the pattern such that: if $P \xrightarrow{\ell_1}_s P_1$ and $P \Longrightarrow_s \xrightarrow{\ell_2}_s \Longrightarrow_s P_2$ then $P_1 \Longrightarrow_s \xrightarrow{\widehat{\ell_2 \lfloor \ell_1}}_s \Longrightarrow_s P_1'$ and $P_1 \Longrightarrow_s \xrightarrow{\widehat{\ell_1 \lfloor \ell_2}}_s \Longrightarrow_s P_2'$, where in each case, we use Lemmas 4.1 and 4.3 to permute the order of the actions $\Longrightarrow_s$ $, \ell_1, \ell_2, \widehat{\ell_2 \lfloor \ell_1}$, and $\widehat{\ell_1 \lfloor \ell_2}$. For the full proof, see Appendix C.1.5. □

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

Theorem 4.2 (Session Determinacy). Let P be session determinate. Then P is determinate and confluent.

Proof. From the definition of confluence (resp. determinacy) and from the definition of P we have that each derivative Q of P is also session determinate. The proof is an immediate result of Lemma 4.5 (resp. Lemma 4.4). □

In the following definition, we build a relation on determinate processes that is later shown to be a bisimulation up-to determinate transitions. We use this relation extensively in the rest of the article in order to reason about session determinate processes and especially event-based optimisations.

Definition 4.12 (Determinate Up-to Expansion Relation). Let $\mathscr{R}$ be a symmetric, typed relation such that if $\Gamma \vdash P \triangleright \Delta \mathscr{R} Q \triangleright \Delta$, then

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

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

Lemma 4.6. Let $\mathscr{R}$ be an up-to expansion relation. Then $\mathscr{R} \subset \approx$.

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

5. Eventful Behavioural Semantics Discussion

This section demonstrates, through examples, the properties of the eventful session programming paradigm as captured by the behavioural semantics developed in Section 4. We study the nature of input and output asynchrony and ordering and the semantic effect of the `arrive` predicate in the event-driven models. We then compare our calculus with other well-known π-calculi.

5.1. *Properties of the ESP Behavioural Theory*

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

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

$$\begin{array}{lcl} s_1?(x);s_2?(y);P \mid B_1 \mid B_2 & \approx & s_2?(y);s_1?(x);P \mid B_1 \mid B_2 \\ s_1!\langle v\rangle;s_2!\langle w\rangle;P \mid B_1 \mid B_2 & \approx & s_2!\langle w\rangle;s_1!\langle v\rangle;P \mid B_1 \mid B_2 \end{array}$$

Permutability shows that actions on different session names are non-blocking and asynchronous. Note that an input and an output action on different sessions cannot generally be permuted:

$$s_1?(x);s_2!\langle v\rangle;P \mid B_1 \mid B_2 \not\approx s_2!\langle v\rangle;s_1?(x);P \mid B_1 \mid B_2$$

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

$$\begin{array}{rcl} s_1?(x);s_1?(y);P \mid B_1 & \not\approx & s_1?(y);s?(x);P \mid B_1 \\ s_1!\langle v\rangle;s_1!\langle w\rangle;P \mid B_1 & \not\approx & s_1!\langle w\rangle;s!\langle v\rangle;P \mid B_1 \end{array}$$

Non permutability on the same session name shows the order-preserving property inside a session, as one would expect from a session type discipline. Following this conclusion, it also holds that:

$$s_1?(x);s_1!\langle v\rangle;P \mid B_1 \not\approx s_1!\langle v\rangle;s_1?(x);P \mid B_1$$

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

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

In the presence of `arrive` s, the bisimulation does not generally hold any more.

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

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

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

The above example shows the arrive construct does not harm the expected behavioural semantics.

4. Arrive inspection ordering. In this example we are dealing with a recursive sequence of `arrive` inspections over session configurations. This model of computation is typical when developing event driven applications. We give a core example of the mentioned case. Let:

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

Both processes recursively inspect sessions s_1 and s_2. We can show by using an *up-to expansion relation* that $P_1 \mid B_1 \mid B_2 \approx P_2 \mid B_1 \mid B_2$. This result is used in Section 6 to verify properties of the selector constructs.

5.2. *Comparisons with Asynchronous and Synchronous Calculi*

In the Introduction, we briefly mentioned differences between the presented asynchronous session bisimulation and existing notions of bisimulations. Below we report more comprehensive comparisons, clarifying the relationship among the different notions of bisimulation proposed for the following variants of the session π-calculus:

1 The asynchronous bisimulation ($\approx_a$) for the session-typed asynchronous π-calculus, defined based on the semantics in (Honda and Tokoro, 1991).
2 The synchronous bisimulation ($\approx_s$) for the session-typed synchronous π-calculus (Takeuchi et al., 1994; Honda et al., 1998).

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

Fig. 12. Comparisons between asynchronous and synchronous calculi: non-blockingness

3 The asynchronous bisimulation ($\approx_2$) for the asynchronous session π-calculus with two endpoint queues without IO queues (Gay and Vasconcelos, 2010; Coppo et al., 2007; Mostrous and Yoshida, 2009).
4 The asynchronous bisimulation $\approx$ fot the asynchronous session π-calculus with two endpoint IO-queues, i.e. Definition 4.6.

Appendix D.1 defines a labelled transition system for the definition of $\approx_2$, i.e a LTS for a session type system with input buffer endpoints and Appendix D.2 gives the proof for the behavioural semantic comparison and properties.

$\approx_2$ is based on the literature (Gay and Vasconcelos, 2010; Coppo et al., 2007; Mostrous and Yoshida, 2009) for the asynchronous session types modelled by the two endpoint queues, without distinction between input and output entries in queues. We call this semantics *non-local* since the sender directly puts the output message on the input queue. The transition relation for non-local semantics is defined by replacing the output and selection rules in Figure 10 by:

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

Figures 12, 13 and 14 summarise distinguishing examples.

— Non-Blocking Input/Output in Figure 12 means inputs/outputs on different channels. The input is non-blocking except in the synchronous semantics, while the output is blocking for both the synchronous and the two-pointed queues.
— The Input/Output Order-Preserving in Figure 13 means that the messages will be received/delivered preserving the order. Both the input and output orders are preserved except the asynchronous semantics.
— The table in Figure 14 explains whether Lemma 4.1 (1) (input advance) or (2) (output delay) is satisfied or not. If not, we place a counterexample. The input can be always advanced and output can be always delayed in both asynchronous and two end-pointed `i`/`o`-queues.

In summary, the asynchronous semantics with two end-pointed `i`/`o`-queues preserves input and output ordering as the synchronous semantics, while inputs and outputs with distinct channels are always non-blocking as the asynchronous semantics.

Another interesting question is the effects of the `arrive` predicate on these combinations. We define the synchronous and asynchronous π-calculi augmented with the `arrive` predicate and

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

Fig. 13. Comparisons between asynchronous and synchronous calculi: order-preservation

	Lemma 4.1 (1)	Lemma 4.1 (2)
$\approx_a$	yes	yes
$\approx_s$	$(\nu\ s)(s!\langle v\rangle;s'?(x);\mathbf{0} \mid s?(x);\mathbf{0})$	$(\nu\ s)(s'!\langle v\rangle;s!\langle v'\rangle;\mathbf{0} \mid s?(x);\mathbf{0})$
$\approx_2$	yes	$s!\langle v\rangle;s'?(x);\mathbf{0} \mid s'[v']$
$\approx$	yes	yes

Fig. 14. Comparisons between asynchronous and synchronous calculi: input and output permutations

local buffers. For the asynchronous π-calculus, we add $a[\vec{h}]$ and `arrive` a in the syntax, and define the following rules for input and outputs.

$$\overline{a}\langle v\rangle \xrightarrow{\overline{a}\langle v\rangle} \mathbf{0} \qquad a[\vec{h}] \xrightarrow{a\langle h\rangle} a[\vec{h}\cdot h] \qquad \texttt{if arrive } a \texttt{ then } P \texttt{ else } Q | a[\varepsilon] \xrightarrow{\tau} Q \mid a[\varepsilon]$$

$$a?(x);P \mid a[\vec{h_1}\cdot h_i\cdot\vec{h_2}] \longrightarrow P\{h_i/x\} \mid a[\vec{h_1}\cdot\vec{h_2}] \qquad \texttt{if arrive } a \texttt{ then } P \texttt{ else } Q \mid a[h\cdot\vec{h}] \xrightarrow{\tau} P \mid a[h\cdot\vec{h}]$$

The above definition precludes order preservation of session requests, 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 the `arrive` predicate. The full semantics can be found in Appendix D.3.

Figure 15 summarises the results which incorporate the `arrive` predicate. Interestingly, in all of the calculi (1–4), the same example is effective to separate two semantics with/without the `arrive`. The `i`/`o`-queues provide non-blocking inputs and outputs, while preserving the input/output ordering, which distinguishes the present framework from other known semantics. As a whole, we observe that the present semantic framework is closest to the asynchronous bisimilarity $\approx_a$, which is restricted by requiring message order-preserving within sessions. Its key properties arise from local, buffered session semantics and typing. We have also seen the semantic significance of the `arrive` predicates, which enables processes to observe the effects of fine-grained synchronisations.

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

Fig. 15. Arrived message detection behaviour in asynchronous and synchronous calculi.

6. Representing High-level Event Constructs in ESP

Among the widely-used event-based programming facilities (including those provided by libraries) in practice, the *selector* construct often serves as the core component of many event-based applications as well as a basis for building other higher-level event-based programming libraries. In brief, the selector is a component implementing a mechanism by which we inspect a set or multiple sets of channels for the arrival of messages. If a message is present, it is dispatched for processing.

This section gives the high-level semantics for a type-safe selector, defines its type-safe and semantic preserving encoding into ESP, and studies the behavioural properties of the selector based on the encoding, which will be directly used for the application in Section 7.

6.1. *Selector semantics*

We first define the core functionality of the selector, which is distilled by three operations: *create* a new selector, *register* a channel with the selector, and *select* (i.e. retrieve from the selector) a channel on which a message has arrived. The syntax in figure 3 is extended as:

$$\begin{array}{lcl} P & ::= & \cdots \\ & \mid & \texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \ \mid\ \texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \ \mid\ \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \ \mid\ r\langle\vec{s}\rangle \end{array}$$

A selector is represented as the process $r\langle\vec{s}\rangle$, where r is the name of the selector and $\vec{s}$ the registered channels in the selector queue. Construct $\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P$ creates a new selector on the bound name r. Selector r is used to register channels with type S. Note that S can be a session set type, allowing sessions with different types to be registered with the selector. The operation $\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P$ registers a session s in selector r and then continues with process P. The selection of a session channel with a non-empty `i`-configuration is performed via process $\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P$, where variable x exists bounded in process P. We then extend ESP with

the following reduction semantics for the selector. We call this extension ESP$^+$.

$$\begin{array}{l}
\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \longrightarrow (\texttt{new}\ r)(P \mid r\langle\varepsilon\rangle) \\
\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \mid r\langle\vec{s}\rangle \longrightarrow P \mid r\langle\vec{s}\cdot s\rangle \\
\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle s\cdot\vec{s}\rangle \mid s[\texttt{i} : \vec{h}] \longrightarrow P\{s/x\} \mid r\langle\vec{s}\rangle \mid s[\texttt{i} : \vec{h}] \qquad (\vec{h} \neq \varepsilon) \\
\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle s\cdot\vec{s}\rangle \mid s[\texttt{i} : \varepsilon] \longrightarrow \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \mid r\langle\vec{s}\cdot s\rangle \mid s[\texttt{i} : \varepsilon]
\end{array}$$

We also have structural rules, e.g. $(\texttt{new}\ r)r\langle\varepsilon\rangle \cong \mathbf{0}$. Operator $\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P$ (binding r in P) creates a new selector $r\langle\varepsilon\rangle$, named r which stores the channels of type S, with the empty queue ε. $\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P$ registers the session channel with r, adding s to the queue $\vec{s}$. The selector $\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P$ checks whether a message is available (i.e. an event has occurred) on the first session in the queue, s (note that x binds P). If so, we execute $P\{s/x\}$; otherwise, s is re-enqueued and the next session is tested.

6.2. *From ESP$^+$ to ESP*

We now show this behaviour can be easily encoded by combining the *message arrival predicate* and recursions.

$$\begin{array}{rcl}
[\![\texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P]\!] & \stackrel{\text{def}}{=} & (\nu\ b)(\overline{b}(x_{\overline{r}}).b(x_r).[\![P]\!] \mid b[\varepsilon]) \\
[\![\texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P]\!] & \stackrel{\text{def}}{=} & \overline{r}!\langle s\rangle;[\![P]\!] \\
[\![r\langle\vec{s}\cdot\vec{s}'\rangle]\!] & \stackrel{\text{def}}{=} & \overline{r}[\texttt{o} : \vec{s}'] \mid r[\texttt{i} : \vec{s}] \\
[\![\texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P]\!] & \stackrel{\text{def}}{=} & \mu Select.r?(x);\texttt{if arrive}\ x\ \texttt{then}\ [\![P]\!]\ \texttt{else}\ \overline{r}!\langle x\rangle;Select
\end{array}$$

The mapping for the other constructs is homomorphic. A selector queue is semantically equivalent to the set of both endpoints of a session channel. We make use of asynchronous session initiation for the new selector qeueue to be conveniently self-generated without any synchronisation. The register operation is syntactic sugar for delegating a session endpoint to be stored at the opposing queue endpoint for monitoring. The use of `arrive` is the key to avoiding blocked inputs in the select operation, allowing the selector to proceed asynchronously while handling any available messages in the inspected session queues. The operations on the collection queue (via r and $\overline{r}$) exchange session channels, hence session delegation (Honda et al., 1998) is essential. We can easily check that this encoding is operationally faithful to the native selector (i.e. the direct ESP extension) using a suitable bisimulation.

Using the above selector encoding, a basic event loop (Example 2.1) is encoded as:

$$\begin{array}{ll}
1. & (\nu\ a)(\overline{a}(x_{\overline{r}}).a(x_r).x_{\overline{r}}!\langle s_1\rangle;\ldots x_{\overline{r}}!\langle s_n\rangle; \\
2. & \quad \mu\,Select.x_r?(y);\ \texttt{if arrive}\ y\ \texttt{then} \\
3. & \qquad\qquad \texttt{typecase}\ y\ \texttt{of}\ \{ \\
4. & \qquad\qquad\quad (?(U_1);?(U_1);?(U_2);\texttt{end}) : y?(y_1);x_{\overline{r}}!\langle y\rangle;Select \\
5. & \qquad\qquad\quad (?(U_1);!\langle U_2\rangle;\texttt{end}) : y?(y_2);y!\langle v\rangle;Select \\
6. & \qquad\qquad \} \\
7. & \qquad\quad \texttt{else}\ x_{\overline{r}}!\langle y\rangle;Select) \mid a[\varepsilon])
\end{array}$$

An event loop implements a core event programming routine. Event handling is performed on the top of the selector through a loop that selects the events which are "ready to be processed" and proceeds with that processing.

6.3. *Typing Event Selectors*

Typing selectors. Typing rules for the extended ESP selector construct naturally follow from the ESP-typing of the selector encoding. The type for a *user* of the selector is written $\overline{\texttt{sel}}\langle S\rangle$, and for the selector itself $\texttt{sel}\langle S\rangle$. For simplicity, we assume these types do not occur as part of other types. The linear environment Δ is extended with two additional type assignments, $r : \overline{\texttt{sel}}\langle S\rangle$ and $r : \texttt{sel}\langle S\rangle$, the latter only used for runtime typing for selector queues. The program typing rules for the selector operations are:

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle}{\Gamma \vdash \texttt{new sel}\langle S\rangle\ r\ \texttt{in}\ P \triangleright \Delta}\ [\text{Selector}] \qquad \frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \quad S' \leq S}{\Gamma \vdash \texttt{register}\ s\ \texttt{to}\ r\ \texttt{in}\ P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \cdot s : S'}\ [\text{Reg}]$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle \cdot x : S}{\Gamma \vdash \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ P \triangleright \Delta \cdot r : \overline{\texttt{sel}}\langle S\rangle}\ [\text{Select}]$$

We define a mapping $[\![\Delta]\!]$ from the typing syntax of the selector to the ESP typing system, where $[\![r : \overline{\texttt{sel}}\langle S\rangle]\!]$ is mapped as $r : S_r \cdot \overline{r} : \overline{S_r}$ when $S_r = \mu X.?(S);X$. All other mappings are identical. The mapping for a selector construct is, as expected, a recursive session type, since a selector process is defined recursively. We obtain, writing ESP$^+$ for the extension of ESP with selectors:

Proposition 6.1 (Soundness of Selector Typing Rules).

1 (Type Preservation) $\Gamma \vdash P \triangleright \Delta$ in ESP$^+$ if and only if $\Gamma \vdash [\![P]\!] \triangleright [\![\Delta]\!]$.
2 (Soundness) $P \equiv P'$ implies $[\![P]\!] \equiv [\![P']\!]$; and $P \longrightarrow P'$ implies $[\![P]\!] \longrightarrow^* [\![P']\!]$.
3 (Safety) A typable process in ESP$^+$ never reduces to an error.

Proof. (1) is proved by typing the mapping from ESP$^+$ to ESP. A full proof can be found in Appendix E.1. (2) is straightforward. (3) is a corollary from (1, 2) and Theorems 3.1 and 3.2. □

This example demonstrates how the fine-grained typing rules of ESP can suggest and justify sound typing rules for high-level event handling constructs through ESP encodings.

6.4. *Behavioural Properties of Selector*

This section investigates basic properties for the event loop, under the hypothesis that each of the event handling processes is sequential and determinate. We can observe that if we arbitrarily permute the entries (session names) inside a selector queue, its behaviour remains the same with respect to the asynchronous session bisimilarity, $\approx$.

In the following definitions, we let $B_i = s_i[\texttt{i} : h_i, \texttt{o} : h_i']$. We also extend the process syntax to $R;Q$ where R is a sequential series of actions, used as a prefix. The *context definition* now allows $C[R]$ where R is replaced at the *hole* $(-)$ of the context (see Definition 4.1).

Definition 6.1. Let $P_{Sel} = \texttt{select}\ x\ \texttt{from}\ r\ \texttt{in}\ \texttt{typecase}\ x\ \texttt{of}\ \{(x_i : S_i) : C[R_i]\}_{1 \leq i \leq m}$ where

— $C = -;\texttt{register}\ x\ \texttt{to}\ r\ \texttt{in}\ \mathit{Select}$.
— *Select* is the recursive variable of the `select` construct in $P_{Sel} = \mu \mathit{Select}.Q$. (i.e. *Select* is the selector's recursive process variable, see the definition in § 6.2).
— $R_i\{s/x_i\}$ is a blocking prefixed, sequential series of actions.
— $C[R_i\{s/x_i\}]$ is session determinate.

Then we define:

— $\mathsf{Sel}_i^n = P_{Sel} \mid r\langle s_i,\ldots,s_k,s_{k+1},\ldots,s_1,s_n,\ldots,s_{i-1}\rangle$ and
— $\mathsf{PermSel}_i^n = P_{Sel} \mid r\langle s_i,\ldots,s_{k+1},s_k,\ldots,s_1,s_n,\ldots,s_{i-1}\rangle$ (i.e. we permute two arbitrary entries in the selector queue in Sel_i^n to obtain a $\mathsf{PermSel}_i^n$).

Hereafter we use the abbreviation $\prod_i P_{1\le i\le m}$ to denote $P_1 \mid P_2 \mid \cdots \mid P_m$.

Lemma 6.1. $\mathsf{Sel}_k^n \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermSel}_k^n \mid \prod_{1\le i\le n} B_i$

Proof. We build a proof based on a similar idea explained in Example 4 in Section 5.1. For full details of the proof, see Appendix E.2. □

We now extend the selector to the *dynamic selector*, where we allow the dynamic addition of session channels to the selector queue via shared channel events. This extension requires the selector to periodically `arrive`-check shared input buffers for available session request messages. For this purpose, we extend, without loss of generality, the selector queue $r\langle v_1 \ldots v_n\rangle$ to register tuples of the form $\vec{v}$, writing $r\langle \vec{v_1} \ldots \vec{v_n}\rangle$. We then register tuples of values with either the form (s,shd) or the form (sd,a), both typed as $(S,\mathtt{io}\langle S'\rangle)$ (with the expected simple extensions for this typing). Values shd and sd are used to respectively represent dummy shared channel and session endpoint placeholders, i.e. the first tuple shape is a wrapper for session channels, and the second for shared names.

The dynamic selection uses name matching to check whether the shared channel component is not the dummy (i.e. whether we are testing a shared channel or not). If so, the `arrive` checks the shared channel for available session requests. Otherwise, the tuple contains a session channel s (or sd) and proceeds like the static selector to check for message arrival on s (or sd). We assume the endpoint corresponding to sd has type `end` and is always empty, $\mathsf{sd}[\mathtt{i} : \varepsilon]$. Hence the expression `arrive` sd will be automatically evaluated to false (`ff`) in the case where the arrival on endpoint sd is checked.

We formally define the encoding for the dynamic selector as follows.

Definition 6.2 (Dynamic Selector Encoding). We extend the register queue $r\langle v_1 \ldots v_n\rangle$ to store tuples of the form $\vec{v}$, writing $r\langle \vec{v_1} \ldots \vec{v_n}\rangle$. A dynamic selector is encoded as:

$$\begin{aligned}[\![\texttt{select}\ (x_s,x_a)\ \texttt{from}\ r\ \texttt{in}\ P]\!] \stackrel{\texttt{def}}{=}\ & \mu Select.r?((x_s,x_a));\texttt{if}\ x_a \neq \mathsf{shd}\ \texttt{and}\ \texttt{arrive}\ x_a \\ & \texttt{then}\ [\![P]\!]\ \texttt{else if}\ \texttt{arrive}\ x_s\ \texttt{then}\ [\![P]\!]\ \texttt{else}\ \bar{r}!\langle(x_s,x_a)\rangle;Select\end{aligned}$$

It is straightforward to extend $[\![P]\!]$ for other constructs and prove the same soundness properties as Proposition 6.1.

Definition 6.3 (Dynamic Selector). We define

$$\begin{aligned}P_{DSel} \quad = \quad & \texttt{select}\ (x_s,x_a)\ \texttt{from}\ r\ \texttt{in}\ \texttt{if}\ x_a = a\ \texttt{then}\ x_a(y).\texttt{register}\ (y,\mathsf{shd})\ \texttt{to}\ r\ \texttt{in} \\ & \texttt{register}\ (x_s,x_a)\ \texttt{to}\ r\ \texttt{in}\ X\ \texttt{else}\ \texttt{typecase}\ x_s\ \texttt{of}\ \{(x_i : S_i) : C[R_i]\}_{1\le i\le m}\end{aligned}$$

where

— $C = -;\texttt{register}\ (x_s,x_a)\ \texttt{to}\ r\ \texttt{in}\ Select$.
— *Select* is the recursive variable of the `select` construct in P_{DSel} (i.e. *Select* is the dynamic selector's recursive process variable, see Definition 6.2).
— $R_i\{s/x_i\}$ is a blocking prefixed, sequential series of actions.

— $C[R_i\{s/x_i\}]$ is session determinate.

Then we define:

— $\mathsf{DSel}_i^n = P_{DSel} \mid r\langle v_i, \ldots, v_k, v_{k+1}, \ldots, v_1, v_n, \ldots, v_{i-1}\rangle$ and
— $\mathsf{PermDSel}_i^n = P_{DSel} \mid r\langle v_i, \ldots, v_{k+1}, v_k, \ldots, v_1, v_n, \ldots, v_{i-1}\rangle$.

Lemma 6.2. $\mathsf{DSel}_k^n \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermDSel}_k^n \mid \prod_{1\le i\le n} B_i$.

Proof. The proof is similar with Lemma 6.1. For the full proof, see Appendix E.4. □

Due to the fact that bisimulation is an equivalence relation, we can use Lemma 6.2 (and Lemma 6.1) to arbitrarily apply a sequence of permutations to the channels in a selector queue and maintain the process behaviour under the hypothesis of sequentiality and determinacy.

The permutation of confluent selectors is a very import result for reasoning and verifying event-loop applications, and is essential to understand the reactive nature of the event-driven programming paradigm (see the next section).

7. Lauer-Needham Transform

In an early work (Lauer and Needham, 1979), 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 (with a single-threaded event loop processing messages sequentially with non-blocking handlers). Following this framework and using high-level asynchronous event primitives such as *selectors* (cf. (Banga et al., 1998; Lea, 2003; Sun Microsystems Inc., 2011)) for the event-based style, many studies compare these two programming styles, often focusing on performance of server architectures (see (Hu et al., 2010, § 6) for recent studies on event programming). These implementations implicitly or explicitly assume a *transformation* from a program written in the thread-based style, especially those which generate a new thread for each service request (as in some of the thread-based web servers), to an *equivalent* event-based program, which treats concurrent services using a single threaded event-loop (as in event-based web servers). However neither the precise semantic effects of such a transformation nor the exact meaning of the associated "equivalence" have been clarified.

We study the semantic effects of such a transformation using the asynchronous session bisimulation. We follow (Lauer and Needham, 1979) to introduce a formal mapping from a thread-based process to an event-loop process. Assuming a server process whose code creates fresh threads at each service invocation, the key idea is to decompose this whole code into distinct smaller code segments each handling the part of the original code starting from a blocking action. Such a blocking action is represented as reception of a message (input or branching). 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 7.1 (Server). A *simple server at a* is a closed process $*a(x).P$ with a typing of form $a : \mathtt{i}\langle S\rangle, b_1 : \mathtt{o}\langle S_1\rangle, .., b_n : \mathtt{o}\langle S_n\rangle$ where P is sequential (i.e. contains no parallel composition $\mid$) and is determinate under any localisation. A simple server is often considered with its localisation with an empty queue $a[\varepsilon]$.

$$
\begin{array}{rcl}
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(\mathtt{sd},a,\emptyset,c_0)\rangle \mid \mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle\ ) \\
 & & \quad \text{where } P_1,..,P_n \text{ are the positive sub-terms of } P; \\
 & & \quad P_1,..,P_{n-m} \text{ are blocking processes whose subjets are respectively typed } S_1,..,S_{n-m}; \\
 & & \quad \text{and } o,\ q \text{ and } \vec{c}=c_1..c_n \text{ are fresh and pairwise distinct.} \\
\mathsf{Loop}\langle o,q\rangle & \stackrel{\text{def}}{=} & *o.\mathtt{select}\ (x_s,x_a,y,z)\ \mathtt{from}\ q\ \mathtt{in}\ \mathtt{if}\ x_a=a\ \mathtt{then}\ \mathtt{new}\ \mathtt{env}\ y'\ \mathtt{in}\ \overline{z}\langle x_s,y'\rangle\ \mathtt{else} \\
 & & \mathtt{typecase}\ x_s\ \mathtt{of}\ \{x_s:S_1:\ \overline{z}\langle x_s,y\rangle,\cdots x_s:S_{n-m}:\ \overline{z}\langle x_s,y\rangle\} \\
\mathsf{CodeBlocks}\langle a,o,q,\vec{c}\rangle & \stackrel{\text{def}}{=} & \mathscr{B}[\![a(w{:}S);P]\!] \mid \prod_{1\le i\le n}\mathscr{B}[\![P_i]\!] \\
\mathscr{B}[\![*a(w{:}S).P]\!] & \stackrel{\text{def}}{=} & *c_0(x_s,y).a(w'{:}S).\mathtt{update}\ (y,w,w')\ \mathtt{in}\ \mathtt{register}\ (x_s,a,\emptyset,c_0)\ \mathtt{to}\ q\ \mathtt{in}\ [\![P,y]\!] \\
\mathscr{B}[\![x^{(i)}?(z:T);Q]\!] & \stackrel{\text{def}}{=} & *c_i(x',y).x'?(z');\mathtt{update}\ (y,z,z')\ \mathtt{in}\ \mathtt{update}\ (y,x,x')\ \mathtt{in}\ [\![Q,y]\!] \\
\mathscr{B}[\![x^{(i)}\&\{l_j:Q_j\}_j]\!] & \stackrel{\text{def}}{=} & *c_i(x',y).x'\&\{l_j:\ \mathtt{update}\ (y,x,x')\ \mathtt{in}\ [\![Q_j,y]\!]\}_j \\
[\![x!\langle e\rangle;Q,y]\!] & \stackrel{\text{def}}{=} & \mathtt{let}\ x'=[\![x]\!]_y\ \mathtt{in}\ x'!\langle[\![e]\!]_y\rangle;\mathtt{update}\ (y,x,x')\ \mathtt{in}\ [\![Q,y]\!] \\
[\![x!\langle k\rangle;Q,y]\!] & \stackrel{\text{def}}{=} & \mathtt{let}\ x'=[\![x]\!]_y\ \mathtt{in}\ \mathtt{let}\ k'=[\![k]\!]_y\ \mathtt{in}\ x'!\langle k'\rangle;\mathtt{update}\ (y,xk,x'k')\ \mathtt{in}\ [\![Q,y]\!] \\
[\![x\oplus l_j;Q,y]\!] & \stackrel{\text{def}}{=} & \mathtt{let}\ z=[\![x]\!]_y\ \mathtt{in}\ z\oplus l_j;[\![Q,y]\!] \\
[\![\overline{b}(z:S);Q,y]\!] & \stackrel{\text{def}}{=} & \overline{b}(z':S);\mathtt{update}\ (y,z,z')\ \mathtt{in}\ [\![Q,y]\!] \\
[\![Q,y]\!] & \stackrel{\text{def}}{=} & \mathtt{let}\ x'=[\![x]\!]_y\ \mathtt{in}\ \mathtt{register}\ (x',\mathtt{shd},c_i,y)\ \mathtt{to}\ q\ \mathtt{in}\ \overline{o}\ \ (Q \text{ is blocking at } x^{(i)}) \\
[\![\mathbf{0},y]\!] & \stackrel{\text{def}}{=} & \overline{o}
\end{array}
$$

Fig. 16. Translation Function for Lauer-Needham Transform

A simple server in the above sense 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 a server does *not* involve accesses to non-trivial mutable local state by threads (semantically ensured by determinacy). A practical example is a web-server which only serves static web pages. Given a server $*a(w:S).P \mid a[\varepsilon]$, its translation, which we call *Lauer-Needham transform* or *LN-transform* for short, is written $LN[\![*a(w:S).P \mid a[\varepsilon]]\!]$.

Our translation uses the notations in Figure 16 for brevity, including: *pairs*; *polyadic inputs/outputs*; a *refined typecase* (which treats types for both shared and session channels); a *refined selector* (which uses the refined typecase); and an *environment* (as used in the standard CPS transform, to carry over session state). All of them are easily encodable in ESP.

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 7.2 (Lauer-Needham Transform). Let $*a(w:S).P$ be a simple server. Then the mapping $LN[\![*a(w:S).P]\!]$ is inductively defined by the rules in Figure 16, 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* $x^{(i)}$), where the indices from 1 to $n-m$ are used for the prefixes and the rest for the definitions ($n \ge m \ge 0$). 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.

The main map $LN[\![*a(w:S).P]\!]$ 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 $q\langle(\mathtt{sd}, a, \emptyset, c_0)\rangle$ named q with a single element $(\mathtt{sd}, a, \emptyset, c_0)$.

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.

4 An environment context, denoted with y, used for storing the context of each event. We create the environment with operator `new env` y `in` P which binds variable y in P and returns a new environment context. Operator $[\![x]\!]_y$ returns name x of the environment context y. Operator `update` (y, x, v) `in` P updates name x of context y with value v. All operators can be encoded in session types.

The initial execution of $LN[\![*a(w : S).P]\!]$ 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 first matched against the shared name a, to determine whether the selected channel is the shared name a, or a session name. If it is the latter, the session name is checked and typed by the *typecase*. Initially the selector, 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 detect 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.

Example 7.1 (Lauer-Needham Transform). As an example of a 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 $?(\mathtt{nat}); !\langle\mathtt{nat}\rangle; ?(\mathtt{nat}); !\langle\mathtt{nat}\rangle; \mathtt{end}$ at a which can be read: *a process should first expect to receive a message of type* `nat` *and send it, then to receive a* `nat`*, and finish by sending a result.* We extract the blocking subterms from this process as follows.

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

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

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

which is used for processing each message. Above, the operation $\texttt{update}\ (y,x,x)\ \texttt{in}$ 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 main event-loop denoted Loop, is given as:

$$
\begin{array}{lcl}
\texttt{Loop} & = & *o.\texttt{select}\ (x_s,x_a,y,z)\ \texttt{from}\ sel\ \texttt{in}\ \texttt{if}\ x_a = a\ \texttt{then}\ \texttt{new}\ \texttt{env}\ y\ \texttt{in}\ \ \overline{z}\langle x_s,y\rangle\ \texttt{else} \\
& & \texttt{typecase}\ x_s\ \texttt{of}\ \{ \\
& & \qquad x_s:?(\texttt{nat});!(\texttt{nat});?(\texttt{nat});!(\texttt{nat}) \quad : \overline{z}\langle x_s,y\rangle \\
& & \qquad x_s:?(\texttt{nat});!(\texttt{nat}) \quad : \overline{z}\langle x_s,y\rangle \\
& & \}
\end{array}
$$

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.

Because a server does not allow, by construction, its internal shared state to be accessed by the threads it spawns, it is currently stateless.[†] Hence we have:

Lemma 7.1. The process $*a(x).R \mid a[\varepsilon]$, where R is session determinate and sequential, is confluent.

The proof is straightforward by using the properties in Section 4.

Lemma 7.2. The process $LN[\![*a(x).P \mid a[\varepsilon]]\!]$ is confluent.

Proof. For proof, see Appendix F. □

Theorem 7.1 (Semantic Preservation). Let $*a(w:S).R \mid a[\varepsilon]$ be a simple server. Then we have $*a(w:S).P \mid a[\varepsilon] \ \approx\ LN[\![a(w:S).P \mid a[\varepsilon]]\!]$.

Proof. The proof constructs a determinate up-to expansion relation, cf. Definition 4.12 and Lemma 4.6. The up-to expansion relation contains each process pair that has all the parallel processes on a blocking prefix for the threaded server and starts from the Loop process for the thread-free process, with arbitrary localisations. We show the conditions needed by Definition 4.12 by using Lemmas 7.1, 7.2, as well as Lemma 6.2. We conclude the proof through Lemma 4.6. For details of the proof, see Appendix F. □

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

8. Related Work and Implementation

8.1. *Related Work*

Confluence. Some of the key proof methods of our work draw their ideas from (Philippou and Walker, 1997), 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 from theirs in that we investigate the effect of asynchronous IO queues and its relationship to confluence.

Using the confluence and determinacy guaranteed by session types, and through observations on the semantics of the `arrive` predicate, we have demonstrated that our 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.

Expressiveness. The work (Beauxis et al., 2008) 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 (Honda and Tokoro, 1991). 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 (Philippou and Walker, 1997; Beauxis et al., 2008) treats event-based programming, including such examples as thread elimination nor does it deal with performance analysis.

Programming constructs that can test the presence of actions or events have been studied in the context of the Linda language (Busi et al., 2000) and CSP (Lowe, 2009; Lowe, 2010).

(Busi et al., 2000) compares the 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) (Honda and Tokoro, 1991), 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 (Lowe, 2009) 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 the interest to investigate event primitives using process calculi. A subsequent work (Lowe, 2010) studies the expressiveness of its variants by investigating a compositional semantics, a congruence result of the operational semantics, and the full abstraction theorem of the trace equivalences. 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 (Busi et al., 2000) is a tuple space where our input/output order preserving examples (which treat different objects with the same session channel) cannot be naturally (and efficiently) defined. The same point applies for (Lowe, 2009; Lowe, 2010). As another difference, the nature of localities has not been considered

either in (Busi et al., 2000; Lowe, 2009; Lowe, 2010) since no notion of a local or remote tuple or environment is defined. Further, neither large applications which include these constructs (§ 7), the equivalences as we treated, nor the performance analysis of the proposed primitives have been discussed in (Busi et al., 2000; Lowe, 2009; Lowe, 2010).

Session typed formalisms. The present paper is the first to include the facility for the type-safe detection of message arrival combined with dynamic inspection of session types at runtime, and develop the behavioural theory for reasoning about eventful programming properties. Our selector re-registers session channels in the selector queue, while using message arrival and session type matching: this complex causality with session delegations is not typable in the progress typing systems in (Bettini et al., 2008; Caires and Vieira, 2010). The asynchronous communication semantics and recursive types are the key features of event programming (as found in SMTP servers in (Hu et al., 2010) and the LN-transformation), which are not treated in (Castagna and Padovani, 2009; Castagna et al., 2009). The key properties for event programming are ensured by static checking (for safety properties) in our integration of sessions and events. The work (Pérez et al., 2012) proves the proof conversions induced by Linear Logic interpretation coincide with an observational equivalence over a strict subset of the binary synchronous session processes. Similar prefix commutations are justified by their conversion rules. None of the above work studies the behavioural theory (bisimulation) nor applications in eventful programming. Such a semantic framework offers a foundation for understanding the relationships among different event-based programming constructs, as we have seen in § 6.

Dynamic types. Dynamic typing with the typecase construct in the λ-calculus is studied in (Abadi et al., 1991; Abadi et al., 1995) where (1) `typecase` is applied for general expressions e; (2) the type can be matched against the type patterns with free type variables; and (3) the default case can be selected if there is no matching (motivated by the use of untyped IO). Our work differs from theirs in that we treat the typecase for types for communication flows, that we impose a stronger constraint on the typecase through session set types dispensing with the default case, and that we use non-trivial subtyping on session set types to control the typecase. Below we outline how the type matching in (2) and the default case (3) can be easily incorporated into our framework, using ESP (the full theory is found in (Hu, 2011)).

First we extend the syntax of the typecase to $\texttt{typecase}\ e\ \texttt{of}\ \{T_i : P_i\}_{i\in I}$. We then introduce the typing system for expressions similar to (Abadi et al., 1991). We define a type matching mechanism, as a matching function from closed type variables to closed types, and use a similar typing system as in (Abadi et al., 1991). These are simple additions, which do not change the basic nature of the type discipline.

For (3), the default case, we include a small, but important additional rule, $\texttt{end} \leq S$ for any $S \in \mathscr{S}$ ($\mathscr{S}$ is the set of all types), to the construction of the subtyping relation in §3.2: this rule means that under the asynchronous communication semantics, doing nothing (`end`) never leads to a lack of composability (the process sends nothing at that channel, and a message from its peer is just buffered). By encoding the type for "default" $\bot$ as $\{\texttt{end}\}$, we can type the default case, since $\bot$ can be raised to an arbitrary session type by [Subs]. For example, we can type $\texttt{typecase}\ k\ \texttt{of}\ \{T_1 : P_1,\ T_2 : P_2,\ \bot : P_3\}$ where the third branch is the default case and the type $\bot$ in the default case indicates that the type of k is unknown, hence P_3 is never allowed to use k

except as a value of a message it may send through another channel. While the extensions in the theory are straightforward, our practical choice is not to include either (2) or (3). This is because (2) may lead to relatively inefficient type matching algorithm for `typecase` (Abadi et al., 1991), while (3) breaks the progress property (note **0** has an arbitrary session type). We believe that the default case is better handled as a session exception (Hu et al., 2008) with clarity and flexibility.

8.2. *Implementation*

The ideas developed in the theory of this paper have been implemented as an extension to earlier work on integrating session types and Java for type-safe concurrent and distributed session programming (Hu et al., 2008). In SJ (Session Java), session programming starts with the declaration of the intended communication protocols as session types. The communication actions comprising a session, such as message passing, branching and recursion, are implemented as operations on session-typed channel endpoints called *session sockets*, objects of Java type `SJSocket`. Following the design of Java, the additional session type information has an explicit notation similar to that for generics, e.g. `SJSocket{?(Data).?(Data).!(Result)}`. The SJ compiler statically checks session implementations against the declared protocols, ensuring correct communication behaviour.

The ESJ (Eventful SJ) (Hu et al., 2010) extension adds the facilities for session-typed asynchronous event handling examined in this paper. ESJ provides high-level session typed APIs for ESP, such as the session event selector (see § 6), as practical abstractions on top of the basic primitives studied in the ESP formalism. In the way that Java programmers can consider `SJSocket` to be a session-typed extension of the standard `java.net.Socket` API, the ESJ selector API is a session-typed extension of the standard `java.nio.channels.Selector` API that supports registering and selecting `SJSocket` endpoints. The type checker in the previous SJ compiler was extended to support session set types by treating the existing session types as singleton set types.

As a brief illustration, Figure 17 lists an excerpt from an ESJ server implementation that corresponds to the basic event loop example in Figure 2 (Section 2.1). The first line declares the `pSelector` session type by the keyword `protocol`. In this example, we instantiate the U_1 and U_2 message types in Figure 2 to `Data` and `Result` Java types respectively. Thus, the `pSelector` session set type specifies the two points in the protocol where the server receives each `Data` message. The remainder of the server code is contained within the outermost ESJ `using` statement. The syntax is borrowed from C# and has the same meaning with regards to automatic disposal of the declared resources, in this case, the ESJ session selector object assigned to the `using` variable `sel` of type `SJSelector{pSelector}`. Here, `SJSelector` is the Java class type of the session selector object (part of the ESJ API for event-driven session programming) and `pSelector` explicitly specifies the session types of `SJSocket` endpoints that may be registered to the selector.

For this example, we keep the code simple (as in Figure 2) by manually `register`-ing to `sel` some sessions already initiated (but no further actions performed) with the concurrent clients on lines 6–8. This means that the `client1`, `client2`, etc. variables are of type `SJSocket{?(Data).?(Data).!<Result>}` at this point in the code. The main event loop of the server is on lines 9–22. Inside the `while` loop, the inner `using` statement declares a session socket variable `s` of type `SJSocket{pSelector}`. The `select` method of `SJSelector` is called on `sel` to return and assign an event-ready session socket to `s` (blocking until one is ready). At this point, ESJ requires the

```
1  // A session set type containing the two event types.
2  protocol pSelector { ?(Data).?(Data).!<Result>, ?(Data).!<Result> }
3
4  // Create a selector of type pSelector.
5  using(SJSelector{pSelector} sel = SJSelector.create(params)) {
6    sel.register(client1); // Register event source session(s) with the selector.
7    sel.register(client2);
8    ...
9    while(run) { // Main event loop.
10     using(SJSocket{pSelector} s = sel.select()) { // Select a session event.
11       typecase(s) { // Identify the type of the occurred event.
12         when(SJSocket{?(Data).?(Data).!<Result>} s1) {
13           Data d1 = s1.receive(); // Handle the first Data event and..
14           sel.register(s1); //..re-register the session with the selector.
15         }
16         when(SJSocket{?(Data).!<Result>} s2) {
17           Data d2 = s2.receive(); // Handle the second Data event, then..
18           s2.send(new Result(...)); // ..send the Result; session completed.
19         }
20       }
21     }
22   }
23 }
```

Fig. 17. Eventful Session Java (ESJ) (Hu et al., 2010) code corresponding to the basic event loop in Figure 2.

session type of `s` to be declared `pSelector` since `select` may return a session socket at either of the two protocol states contained in this set type.

Finally, lines 11–20 illustrate the ESJ syntax for `typecase`. As in Figure 2, the `typecase` is used to dynamically determine the session type of the returned socket, and hence whether the socket should be used to `receive` the first `Data` message (and re-register the socket to the selector), or to receive the second `Data` message and `send` a `Result`. Following the theory, the `typecase` rebinds the session socket to new variables `s1` and `s2` of the appropriate type in each `when` case. As stipulated by the formal typing system, the ESJ type checker ensures that the `typecase` covers all the cases of the `pSelector` set type, so at least one case is guaranteed to match at runtime, and that the session-typed variables (`s`, `s1`, `s2`) are used correctly within their respective `using` and `when` scopes.

In comparison to typical event-driven code, such as a standard Java NIO implementation, and aside from the formal safety guarantees, we observe that ESJ code benefits from higher-level abstraction and explicit structuring guided by session typing. In particular, the notion of communication session event explicitly encompasses both the message type and protocol state, making clear the distinction between the first and second `Data` arrival events.

Further ESJ examples can be found in (Hu et al., 2010), including an ESJ implementation of an event-driven SMTP server (and client) as a real-world application use case. The server is interoperable with standard, non-SJ (i.e. not session-typed) SMTP clients such as Outlook, Thunderbird and Apple Mail. The SMTP example also demonstrates how an ESJ server can dynamically accept and register an arbitrary number of concurrent client sessions (a detail that

has been abridged in the above example). Type-safe interoperability is enabled by two main elements. One is the design of the ESJ Runtime to incorporate a variety of transports, including TCP, HTTP and shared memory, uniformly under the SJ session abstraction. This means a single ESJ selector instance is capable of monitoring sessions running over heterogeneous transports as well as being of heterogeneous types. The implementation architecture for the transport-independent ESJ selector is discussed in more detail in (Hu et al., 2010) (see Figure 7 there for a diagrammatic overview). While ESJ programs are statically checked to be session type-safe by the compiler, the second element is run-time monitoring of (SMTP) sessions by the ESJ Runtime to ensure that non- session-typed peers indeed conform to the same protocol.

Performance (scalability) benchmarks for the ESJ Runtime (Hu et al., 2010; SJ, 2010) (with basic multithreaded and event-driven implementations in standard Java as base cases) demonstrate the feasibility of integrating session types and event-driven programming, and affirm the application of the Lauer-Needham transform in practice. Micro-benchmarks and a macro-benchmark using the SMTP server show that thread-eliminated ESJ programs exhibit higher average throughput and better response-time than the multithreaded versions as the server is loaded by an increasing number (ranging up to 1000) of concurrent clients.

9. Conclusion

We have proposed a formal theory of the consistent integration of session types and events, and studied its behavioural semantics, offering a basis for structured asynchronous eventful communications programming. We provided formal semantics and a type system for an eventful session π-calculus, and proved its expressiveness through a study of bisimulation relations. We also give comprehensive comparisons on behavioural properties with other existing calculi. To our best knowledge, our work is the first to present such an extension for session types and its behavioural theory, allowing one to reason about a wide range of eventful programming applications including the LN-transformation. Although we developed the eventful session calculus based on the binary session type theory, we believe the calculus and its typing system are readily extended to multiparty sessions (Honda et al., 2008). The main elements we introduce for modelling ESP are (1) `i`/`o`-queues, (2) the arrive predicate, (3) session typecase and (4) set types. Adapting (1)–(3) to the standard multiparty session type framework is straightforward because they do not directly affect the global types, and the local types which are projected from a global type are very similar to binary session types; similarly for the typed processes. Concerning (4), it would be possible to specify the interaction between the user processes and the server process as a global type with set types as message types (for the delegation of registered/selected endpoints). However, the main application protocols between users would be unaffected: with respect to the ESP methodology demonstrated in this article, the role of set types is more in verifying correct event handling by processes rather than the specification of user application protocols by global types (the latter would typically be at a high level that abstracts from concrete multithreaded or event-driven implementations).

In addition to further development and extension of Eventful SJ (Hu et al., 2010), as future work, we plan to investigate bisimulation theories under multiparty session types (Honda et al., 2008) and the relationship with a linear logic interpretation of sessions, which connects a behavioural theory and permutation laws under locality assumption (Caires and Pfenning, 2010).

References

Abadi, M., Cardelli, L., Pierce, B. C., and Plotkin, G. D. (1991). Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268.

Abadi, M., Cardelli, L., Pierce, B. C., and Rémy, D. (1995). Dynamic typing in polymorphic languages. *J. Funct. Program.*, 5(1):111–130.

Adya, A., Howell, J., Theimer, M., Bolosky, W. J., and Douceur, J. R. (2002). 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*.

Banga, G., Druschel, P., and Mogul, J. C. (1998). Better operating system features for faster network servers. *SIGMETRICS Performance Evaluation Review*, 26(3):23–30.

Beauxis, R., Palamidessi, C., and Valencia, F. D. (2008). On the asynchronous nature of the asynchronous pi-calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 473–492. Springer.

Bettini, L. et al. (2008). Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer.

Busi, N., Gorrieri, R., and Zavattaro, G. (2000). Comparing three semantics for linda-like languages. *Theor. Comput. Sci.*, 240(1):49–90.

Caires, L. and Pfenning, F. (2010). Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer.

Caires, L. and Vieira, H. T. (2010). Conversation types. *Theor. Comput. Sci.*, 411(51-52):4399–4440.

Castagna, G. et al. (2009). Foundations of session types. In *PPDP'09*, pages 219–230. ACM.

Castagna, G. and Padovani, L. (2009). Contracts for mobile processes. In *CONCUR 2009*, number 5710 in LNCS, pages 211–228.

Coppo, M., Dezani-Ciancaglini, M., and Yoshida, N. (2007). Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31.

Gay, S. and Vasconcelos, V. T. (2010). Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50.

Hennessy, M. (2007). *A Distributed Pi-Calculus*. CUP.

Honda, K. and Tokoro, M. (1991). An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147.

Honda, K., Vasconcelos, V. T., and Kubo, M. (1998). Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer.

Honda, K. and Yoshida, N. (1995). On reduction-based process semantics. *TCS*, 151(2):437–486.

Honda, K. and Yoshida, N. (2007). A uniform type structure for secure information flow. *TOPLAS*, 29(6).

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM.

Hu, R. (2011). SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., and Honda, K. (2010). Type-safe eventful sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag.

Hu, R., Yoshida, N., and Honda, K. (2008). Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541. Springer.

Kouzapas, D. (2009). A session type discipline for event driven programming models. Master's thesis, Imperial College London. http://www.doc.ic.ac.uk/teaching/distinguished-projects/2010/d.kouzapas.pdf.

Kouzapas, D., Yoshida, N., and Honda, K. (2011). On asynchronous session semantics. In *FMOODS/FORTE*, volume 6722 of *LNCS*, pages 228–243.

Krohn, M. (2004). Building secure high-performance web services with OKWS. In *ATEC'04*, pages 15–15. USENIX Association.

Krohn, M., Kohler, E., and Kaashoek, M. F. (2007). Events can make sense. In *ATC'07*, pages 1–14. USENIX Association.

Lauer, H. C. and Needham, R. M. (1979). On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19.

Lea, D. (2003). *Scalable IO in Java*. `http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf`.

Li, P. and Zdancewic, S. (2007). Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. *SIGPLAN Not.*, 42(6):189–199.

Lowe, G. (2009). Extending CSP with tests for availability. *Proceedings of Communicating Process Architectures (CPA 2009)*.

Lowe, G. (2010). Models for CSP with availability information. In Fröschle, S. B. and Valencia, F. D., editors, *EXPRESS'10*, volume 41 of *EPTCS*, pages 91–105.

Milner, R. (1980). *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, Berlin.

Milner, R., Parrow, J., and Walker, D. (1992). A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.*, 100(1).

Mostrous, D. and Yoshida, N. (2009). Session-based communication optimisation for higher-order mobile processes. In *TLCA'09*, volume 5608 of *LNCS*, pages 203–218. Springer.

Pérez, J. A., Caires, L., Pfenning, F., and Toninho, B. (2012). Linear logical relations for session-based concurrency. In *ESOP*, volume 7211 of *LNCS*, pages 539–558. Springer.

Philippou, A. and Walker, D. (1997). On confluence in the pi-calculus. In *ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 314–324. Springer.

Pierce, B. and Sangiorgi, D. (1996). Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454.

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

SJ (2010). SJ homepage. `http://www.doc.ic.ac.uk/~rhu/sessionj.html`.

Sun Microsystems Inc. (2011). New IO APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`.

Takeuchi, K., Honda, K., and Kubo, M. (1994). An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413.

von Behren, R., Condit, J., and Brewer, E. (2003a). Why events are a bad idea (for high-concurrency servers). In *HOTOS'03*, pages 4–4. USENIX Association.

von Behren, R. et al. (2003b). Capriccio: scalable threads for internet services. In *SOSP '03*, pages 268–281. ACM.

Welsh, M., Culler, D. E., and Brewer, E. A. (2001). SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP'01*, pages 230–243. ACM Press.

Appendix A. Properties of Subtyping

Proposition A.1 (Subtyping Properties). The set of *composable* types of a session type S is defined as: $\mathtt{comp}(S) = \{S' \mid S' \leq \overline{S}\}$.

(i) $\leq$ is a preorder.
(ii) (semantics of $\leq$) $S_1 \leq S_2$ if and only if $\mathtt{comp}(S_2) \subseteq \mathtt{comp}(S_1)$.

Proof. **Part (i).** Transitivity and reflexivity are proved following (Pierce, 2002, Theorems 21.3.6–7). We demonstrate the main cases for session set types.

For transitivity, a relation $\mathscr{R} \subseteq \mathscr{T} \times \mathscr{T}$ is transitive if closed under the monotone function $TR(\mathscr{R}) = \{(x,y) \mid \exists z \in \mathscr{T}.\{(x,z),(z,y)\} \subseteq \mathscr{R}\}$. We note that if $TR(\mathscr{F}(\mathscr{R})) \subseteq \mathscr{F}(TR(\mathscr{R}))$, then the greatest fixed point of $\mathscr{F}$ is transitive, and show $TR(\mathscr{F}(R)) \subseteq \mathscr{F}(TR(\mathscr{R}))$ by taking

$(T,T') \in TR(\mathscr{F}(\mathscr{R}))$. By definition of TR, there exists a T'' such that $(T,T''),(T'',T') \in \mathscr{F}(\mathscr{R})$, and we proceed by cases on T'' to show $(T,T') \in \mathscr{F}(TR(\mathscr{R}))$.
Case: $T'' = \{S''_k\}_{k\in K}$
By definition of $\mathscr{F}$, $(T,T'') \in \mathscr{F}(\mathscr{R})$ implies $T = \{S_i\}_{i\in I}$, $\forall k \in K, \exists i \in I.(S_i, S''_k) \in \mathscr{R}$. There are two subcases for $(T'',T') \in \mathscr{F}(\mathscr{R})$. First, $T' = \{S'_j\}_{j\in J}$, $\forall j \in J, \exists k \in K.(S''_k, S'_j) \in \mathscr{R}$. By definition of TR, $\forall j \in J, \exists i \in I.(S_i, S'_j) \in TR(\mathscr{R})$. Hence, by definition of $\mathscr{F}$, $(\{S_i\}_{i\in I}, \{S'_j\}_{j\in J}) \in \mathscr{F}(TR(\mathscr{R}))$. Second, $T' = S'$, $|K| = 1, (S''_1, S') \in \mathscr{R}$. By definition of TR, $\exists i \in I.(S_i, S') \in TR(\mathscr{R})$. Hence, by definition of $\mathscr{F}$, $(\{S_i\}_{i\in I}, S') \in \mathscr{F}(TR(\mathscr{R}))$.
The other cases are standard, with similar treatment of the subcases where T has the shape $\{S_i\}_{i\in I}$.

For reflexivity, let the identity relation $\mathscr{I} = \{(T,T) \mid T \in \mathscr{T}\}$, and $\mathscr{R} \subseteq \mathscr{T} \times \mathscr{T}$ is $\mathscr{F}$-consistent if $\mathscr{R} \subseteq \mathscr{F}(\mathscr{R})$. By the principle of coinduction, if $\mathscr{I}$ is $\mathscr{F}$-consistent, then the greatest fixed point of $\mathscr{F}$ contains $\mathscr{I}$. To show $\mathscr{I}$ is $\mathscr{F}$-consistent, we take $(T,T) \in \mathscr{I}$ and proceed by cases on T to show $(T,T) \in \mathscr{F}(\mathscr{I})$.
Case: $T = \{S_i\}_{i\in I}$
By definition of $\mathscr{I}$, $\forall I \in I.(S_i, S_i) \in \mathscr{I}$. Hence, by defintion of $\mathscr{F}$, $(\{S_i\}_{i\in I}, \{S_i\}_{i\in I}) \in \mathscr{F}(\mathscr{I})$, since the condition $\forall j \in J, \exists i \in I.(S_i, S_j) \in \mathscr{I}$ is trivially satisfied when $I = J$.
The remaining cases are standard.

Part (ii). By Lemma 3.1, $S_1 \leq S_2$ iff $\overline{S_1} \geq \overline{S_2}$. But by definition $\overline{S_1} \geq \overline{S_2}$ iff $\mathsf{comp}(S_2) \subseteq \mathsf{comp}(S_1)$, as required. □

Appendix B. Subject Reduction and Communication and Event-Handling Safety

This Appendix relates to the proof of subject reduction for the asynchronous session types typing system, and the communication and event-handling safety.

Lemma B.1 (Weakening Lemma). Let $\Gamma \vdash P \triangleright \Delta$.

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

Proof. For part (i) we apply the induction on the structure of ESP process syntax. The base cases are trivial. We demonstrate the inductive step. Suppose $P = u(x : S).P_1$. Then we have $\Gamma \vdash P_1 \triangleright \Delta \cdot x : S$. By the inductive hypothesis, we have $\Gamma \cdot X : \Delta' \vdash P_1 \triangleright \Delta \cdot x : S$ and $X \notin \mathtt{dom}(\Gamma)$. We can easily conclude that $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$. Next we demonstrate the case for the typecase process. Let $P = \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i\in I}$. By the inductive hypothesis, we obtain that for each $i \in I$, $\Gamma \cdot X : \Delta' \vdash P_i \triangleright \Delta_i$ and $X \notin \mathtt{dom}(\Gamma)$. It easy to conclude that $\Gamma \cdot X : \Delta' \vdash P \triangleright \Delta$. The rest of the inductive step cases are similar.

Parts (ii) is similar to part (i).

For part (iii) we again use induction on the structure of ESP process syntax. It is easy to see the basic step for process $\mathbf{0}$, where we obtain $\Gamma \vdash \mathbf{0} \triangleright k : \mathtt{end}$ by the typing rule [Inact]. We apply a case analysis for the induction step. Let $P = u(x : S).P_1$. By the inductive hypothesis, we obtain that $\Gamma \vdash P_1 \triangleright \Delta \cdot x : S \cdot k : \mathtt{end}$ with $k \notin \mathtt{dom}(\Delta)$. We can now easily conclude that $\Gamma \vdash P \triangleright \Delta \cdot k : \mathtt{end}$. The rest of the cases are similar. □

Lemma B.2 (Strengthening Lemma).

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

Proof. For part (i) we apply the induction on the structure of ESP process syntax. The base cases are trivial. We demonstrate the inductive step. Let $P = u(x:S).P_1$ and $\Gamma \cdot X : \Delta' \vdash P_1 \triangleright \Delta \cdot x : S$. By the inductive hypothesis, we have that $\Gamma \vdash P_1 \triangleright \Delta \cdot x : S$. We can easily conclude that $\Gamma \vdash P \triangleright \Delta$. Let $P = \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$ and for each $i \in I$, $\Gamma \cdot X : \Delta' \vdash P_i \triangleright \Delta_i$. From the inductive hypothesis, we obtain that for each $i \in I$, $\Gamma \vdash P_i \triangleright \Delta_i$ and $X \notin \mathtt{dom}(\Gamma)$. It is easy to conclude that $\Gamma \vdash P \triangleright \Delta$. The rest of the induction step cases are similar.

Parts (ii) and (iii) are similar to part (i). □

Lemma B.3 (Substitution Lemma).

(i) If $\Gamma \cdot x : U, \Delta \vdash e : U'$ and $\Gamma \vdash v \triangleright U$, then $\Gamma, \Delta \vdash e\{v/x\} : U'$.
(ii) If $\Gamma, \Delta \cdot x : T \vdash e : U$ and s fresh, then $\Gamma, \Delta \cdot s : S \vdash e\{s/x\} : U$.
(iii) If $\Gamma \cdot x : U \vdash P \triangleright \Delta$ and $\Gamma \vdash v \triangleright U$, then $\Gamma \vdash P\{v/x\} \triangleright \Delta$.
(iv) If $\Gamma \vdash P \triangleright \Delta \cdot k : T$, then $\Gamma \vdash P\{s/k\} \triangleright \Delta \cdot s : T$.

Proof. We apply induction on the definition of ESP process syntax. The base cases are trivial. We demonstrate the inductive step.

Parts (i) and (ii) are proved with a simple induction on the structure of expressions e.

For Part (iii), Let $P = u(y:S).P_1$. Then we have $\Gamma \cdot x : U \vdash P_1 \triangleright \Delta \cdot y : S$. From the inductive hypothesis, we have that $\Gamma \vdash P_1\{v/x\} \triangleright \Delta \cdot y : S$. We can now easily conclude that $\Gamma \vdash P\{v/x\} \triangleright \Delta$. For the typecase case, let $P = \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}$. From the inductive hypothesis, we get that for each $i \in I$, $\Gamma \cdot x : U \vdash P_i \triangleright \Delta_i$ and $\Gamma \vdash v : U$. It is easy to derive that $\Gamma \vdash P\{v/x\} \triangleright \Delta$. The rest of the cases are similar.

For part (iv) we demonstrate the interesting case for the `typecase` construct.
Let $\Gamma \vdash \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i \in I} \triangleright \Delta \cdot k' : T$. From the inductive hypothesis, we obtain that for each $i \in I$, $\Gamma \vdash P_i\{s/k'\} \triangleright \Delta \cdot s : T$. From typing rule [Typecase], we conclude that $\Gamma \vdash \mathtt{typecase}\ k\ \mathtt{of}\ \{(x_i : S_i) : P_i\}_{i \in I}\{s/k'\} \triangleright \Delta \cdot s : T$. Note that the same results holds if $k' = k$. The rest of the cases are trivial. □

Theorem B.1 (Subject Congruence and Reduction). *(Theorem 3.1)*

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

Proof. The proof for (i) subject congruence uses a case analysis on the structural congruence rule and it is standard. We demonstrate one basic case since the rest of the cases are similar. Let $\Gamma \vdash P \mid Q \triangleright \Delta$ and $P \mid Q \equiv Q \mid P$. From typing rule [Cong] we have that $\Delta = \Delta_1 \cup \Delta_2$ with $\Gamma \vdash P \triangleright \Delta_1$ and $\Gamma \vdash Q \triangleright \Delta_2$. It is trivial to see that $\Gamma \vdash Q \mid P \triangleright \Delta$ because $\Delta_1 \cup \Delta_2 = \Delta_2 \cup \Delta_1$. The rest of the cases are trivial.

For (ii) subject reduction, we prove by induction on the reduction relation.

Case: [Request1]
$\Gamma \vdash \overline{a}(x).P \triangleright \Delta \longrightarrow (\nu\ s)(P\{\overline{s}/x\} \mid \overline{a}\langle s\rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) \triangleright \Delta'$. By rule (Req), we have that $\Gamma \vdash P \triangleright \Delta \cdot \overline{x} : \overline{S}$. By rules (InQ, OutQ), we obtain that $\Gamma \vdash s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \emptyset$. Then by rule (Areq), we have $\Gamma \vdash \overline{a}\langle s\rangle \triangleright s : S$. We now apply rule (Conc) to obtain $\Gamma \vdash P\{\overline{s}/x\} \mid \overline{a}\langle s\rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \Delta \cdot s : S \cdot \overline{s} : \overline{S}$. Rule (Sres) gives us $\Gamma \vdash (\nu\ s)(P\{\overline{s}/x\} \mid \overline{a}\langle s\rangle \mid \overline{s}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]) \triangleright \Delta$, as required.
Case: [Request2]
$\Gamma \vdash \overline{a}\langle s\rangle \mid a[\vec{s}] \triangleright \vec{s} : \vec{S} \cdot s : S \longrightarrow a[\vec{s} \cdot s] \triangleright \vec{s} : \vec{S} \cdot s : S$. We type the processes that compose the left hand side process using typing rules (Queue), (Areq). By rule (Conc) and the definition of $*$ we obtain the typing $\vec{s} : \vec{S} \cdot s : S$. The right hand side is typed using typing rule (Areq) to obtain the same result.
Case: [Accept]
$\Gamma \vdash a(x).P \mid a[s \cdot \vec{s}] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S \longrightarrow P\{s/x\} \mid a[\vec{s}] \triangleright \Delta \cdot \vec{s} : \vec{S} \cdot s : S$. For the left hand side, we use rules (Queue), (Acc) and (Conc), to get the typing result. From rule (Acc) we have that $\Gamma \vdash P\{s/x\} \triangleright \Delta$. From here is easy to find the same typing for the right hand side.
Case: [Send] (Value)
$\Gamma \vdash s!\langle v\rangle; P \mid s[S_1, \mathtt{o} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P \mid s[S_2, \mathtt{o} : v \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_2]$, where $\Gamma \vdash \vec{h} : \vec{T}, \Gamma \vdash v : T$. For the left hand side we type $\Gamma \vdash s!\langle v\rangle; P \triangleright \Delta \cdot s : !\langle T\rangle; S'$ and $\Gamma \vdash s[!\langle T\rangle; S'_1, \mathtt{o} : \vec{h}] \triangleright s : O \cdot s[!\langle T\rangle; S'_1]$. Using (Conc) we get $!\langle T\rangle; S' * O = S$. Now if we type the right hand side we get $\Gamma \vdash s!\langle v\rangle; P \triangleright \Delta \cdot s : S'$ and $\Gamma \vdash s[S'_1, \mathtt{o} : v \cdot \vec{h}] \triangleright s :!\langle T\rangle; O \cdot s[S'_1]$. We compose to get $S' * !\langle T\rangle; O = !\langle T\rangle; S' * O = S$ and $S'_1 = S_2$.
Case: [Receive] (Value)
$\Gamma \vdash s?(x); P \mid s[S_1, \mathtt{i} : v \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P\{v/x\} \mid s[S_2, \mathtt{i} : \vec{s}] \triangleright \Delta \cdot s : S \cdot s[S_2]$. For the left hand side we have $\Gamma \vdash s?(x); P \triangleright \Delta \cdot s :?(T); S'$ and $\Gamma \vdash s[?(T); S'_1, \mathtt{i} : v \cdot \vec{h}] \triangleright s :?(T); I \cdot s[?(T); S'_1, \mathtt{i} :]$. We compose and get $?(T); S' * ?(T); I = S' * I = S$. For the right hand side we have $\Gamma \vdash P \triangleright \Delta \cdot s : S'$ and $\Gamma \vdash s[S_2, \mathtt{i} : \vec{h}] \triangleright s : I \cdot s[S_2, \mathtt{i} :]$. By composition we get $S' * I = S$ and $S'_1 = S_2$.
Case: [Receive] (Delegation)
$\Gamma \vdash s?(x); P \mid s[S_1, \mathtt{i} : s' \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s' : S' \cdot s[S_1] \longrightarrow P\{s'/x\} \mid s[S_2, \mathtt{i} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s' : S' \cdot s[S_2]$. We have $\Gamma \vdash s?(x); P \triangleright \Delta \cdot s :?(S'); S''$, $\Gamma \vdash s[S_1, \mathtt{i} : s' \cdot \vec{h}] \triangleright s :?(S'); I \cdot s' : S' \cdot s[?(S'); S'_1]$ and $\Delta \cdot s : ?(S'); S'' * s :?(S'); I \cdot s' : S' \cdot s[?(S'); S'_1] = \Delta \cdot s :?(S'); S \cdot s' : S' \cdot s[?(S'); S'_1]$. For the right hand side we have $\Gamma \vdash P\{s'/x\} \triangleright \Delta \cdot s : S'' \cdot s' : S'$, $\Gamma \vdash s[S_2, \mathtt{i} : \vec{h}] \triangleright s : I \cdot s[S'_1]$ and $\Delta \cdot s : S'' \cdot s' : S' * s : I \cdot s[S'_1] = \Delta \cdot s :?(S'); S \cdot s' : S' \cdot s[S'_1]$.
Case: [Send] (Delegation)
Similar to the above case.
Case: [Sel]
$\Gamma \vdash s \oplus v; P \mid s[S_1, \mathtt{o} : \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_1] \longrightarrow P \mid s[S_2, \mathtt{o} : l \cdot \vec{h}] \triangleright \Delta \cdot s : S \cdot s[S_2]$. The proof is similar to [Send] case.
Case: [Bra]
$\Gamma \vdash s\&\{l_i : P_i\}_{i \in I} \mid s[S_1, \mathtt{i} : l_k \cdot \vec{h}] \triangleright \Delta \cdot s : S' \cdot s[S_1] \longrightarrow P_k \mid s[S_2, \mathtt{i} : \vec{s}] \triangleright \Delta \cdot s : S' \cdot s[S_2]$ where $S' = S_k * M_{\mathtt{i}}$ and $\Gamma \vdash s[S_1, \mathtt{i} : \vec{s}] \triangleright s : M_{\mathtt{i}} \cdot s[S_1]$. The proof is similar to the (Receive) case.
Case: [Comm]
$\Gamma \vdash P \mid s[S_1, \mathtt{o} : \vec{h} \cdot v] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'}]\varepsilon \triangleright \Delta_1 \longrightarrow P \mid s[S_1, \mathtt{o} : \vec{h}] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'} \cdot v] \triangleright \Delta_1$. $\Gamma \vdash P \triangleright \Delta \cdot s : S_1 \cdot \overline{s} : S_2$ with $\Gamma \vdash P \mid s[!\langle T\rangle; S_1, \mathtt{o} : \vec{h} \cdot v] \mid \overline{s}[?(T); S_2, \mathtt{i} : \vec{h'}] \triangleright \Delta \cdot s :!\langle T\rangle; S \cdot \overline{s} :?(T); \overline{S}$ from the induction hypothesis. If we type the right hand side we have that $\Gamma \vdash P \mid s[S_1, \mathtt{o} : \vec{h}] \mid \overline{s}[S_2, \mathtt{i} : \vec{h'} \cdot v] \triangleright \Delta \cdot s : S \cdot \overline{s} : \overline{S}$ as required.

Case: [Typecase]
$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{S_i : P_i\}_{i\in I} \mid s[S_k, \texttt{i} : \vec{h}, \texttt{o} : \vec{h'}] \triangleright \Delta_1 \longrightarrow P_k \mid s[S_k, \texttt{i} : \vec{h}, \texttt{o} : \vec{h'}] \triangleright \Delta_2$.
$\Gamma \vdash \texttt{typecase}\ s\ \texttt{of}\ \{S_i : P_i\}_{i\in I} \mid s[S_k, \texttt{i} : \vec{h}, \texttt{o} : \vec{h'}] \triangleright \Delta \cdot s : \{S_i\}_{i\in I}$. From the right hand side of the reduction, we have $\Gamma \vdash P_k \mid s[S_k, \texttt{i} : \vec{h}, \texttt{o} : \vec{h'}] \triangleright \Delta \cdot s : S_k$. Since $S_k \leq \{S_i\}_{i\in I}$, we use [Subs] to obtain $\Delta_1 = \Delta_2$ and Δ_2 well-configured from the induction hypothesis.
Case: [arrive]
$\Gamma \vdash \texttt{if arrive}\ s\ \texttt{then}\ P\ \texttt{else}\ Q \mid s[\texttt{i} : \vec{h}] \triangleright \Delta \cdot s : T\ *\ s : M \longrightarrow P \mid s[\texttt{i} : \vec{h}] \triangleright \Delta' \cdot s : T\ *\ s : M$.
$\Gamma \vdash \texttt{if arrive}\ s\ \texttt{then}\ P\ \texttt{else}\ Q \triangleright \Delta \cdot s : T$. From [If] we have that $\Gamma \vdash P \triangleright \Delta \cdot s : T$ and $\Gamma, \Delta \vdash \texttt{arrive}\ s \triangleright \texttt{bool}$ so, $\Gamma \vdash P \mid s[\texttt{i} : \vec{s}] \triangleright \Delta \cdot s : T\ *\ s : M$ and thus $\Delta = \Delta'$ as required. □

Theorem B.2 (Communication and Event-Handling Safety). *(Theorem 3.2)* If P is a well-typed process, then P never reduces to an error.

Proof. Communication safety follows as a corollary from subject reduction (Theorem 3.1). Assuming the reduction of a typable process to an error (page 16), we show that the error is not typable, thus leading to a contradiction. We demonstrate key cases for `typecase` and `arrive`, corresponding to cases (h) and (f) in the definition of s-redexes (page 16). These cases ensure that a well-typed process does not reduce to a stuck `typecase` term where the current active type of the session cannot be matched to any of the specified type cases, nor a term in which the `arrive` predicate is used to check the arrival of a message of an unexpected type.

Assume a process $P \longrightarrow P'$, where $\Gamma \vdash P \triangleright \Delta$ and Δ is well-configured. By Theorem 3.1, $\Gamma \vdash P' \triangleright \Delta'$, $\Delta \longrightarrow \Delta'$ and Δ' is well-configured. Say P' is an error. Then P' contains, up to structural congruence, a term Q that is the parallel composition of two s-processes that do not form an s-redex. Note that the definition of the $*$ operator and (QConc) implicitly prevent the parallel composition of two s-processes from being well-typed unless one term is an s-configuration and the other is either an $\bar{s}$-configuration or an s-process that is not a configuration. We proceed by cases to show Q, and thus P', is not typable.
Case: $Q = \texttt{typecase}\ s\ \texttt{of}\ \{(s_i : S_i) : P_i\}_{i\in I} \mid s[S]$ where $\nexists i \in I.S_i \leq S$.
To type Q, rule (QConc) must compose for $\Delta'(s)$ some S', where $\{S_i\}_{i\in I} \leq S'$, and $M[S]$, where M is message type of the s-configuration. By definition of $*$ composition of linear environments, $S' = S$, contradicting $\nexists i \in I.S_i \leq S$.
Case: $Q = E[\texttt{arrive}\ s\ v] \mid s[?(U); S, \texttt{i} : \vec{h}]$ with v of type U, and $\vec{h} = v' \cdot \vec{h'}$, v' not of type U.
Consider the subcase where the E-context is the `if`-term (the others are similar). By (If) and (AVal), the type of s is of the shape $?(U); S'$. However, to type Q, rule (QConc) must compose for $\Delta'(s)$ some $?(U); S''$, where $S' \leq S''$, and $?(U'); M_{\texttt{i}}[?(U); S]$, where $U' \neq U$, in which case the $*$ operator is not defined. □

Before we proceed, note that:

Lemma B.4. Let $P = P_1 \mid s[\texttt{i} : \varepsilon, \texttt{o} : \varepsilon]$ and $\Gamma \vdash P \triangleright \Delta$ with Δ well-configured. Then if $P \longrightarrow^* P_2 \mid s[\texttt{i} : \vec{h_1}, \texttt{o} : \vec{h_2}]$ then $\vec{h_1} = \varepsilon$ or $\vec{h_2} = \varepsilon$.

The above lemma states that at least one of the queues in a session endpoint is empty during any execution.

Proof. The proof is by induction on the length of $\rightarrow\rightarrow$. The basic step is trivial. For the inductive step we have three cases:

(i) $\Gamma \vdash P_2 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \triangleright \Delta$.
(ii) $\Gamma \vdash P_2 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \varepsilon] \triangleright \Delta$.
(iii)$\Gamma \vdash P_2 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \vec{h_1}] \triangleright \Delta$.

with Δ well-configured in all three cases (Subject Reduction Theorem 3.1) and $|\vec{h_1}| \geq 1$.

We prove part (ii), with parts (i) and (iii) being similar.

Let $P_2 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \varepsilon] \longrightarrow P_2' \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : v]$. This implies that $P_2 = (\nu\ \vec{n})(P_3 \mid s!\langle v\rangle; P_4)$ or $P_2 = (\nu\ \vec{n})(P_3 \mid s \oplus l; P_4)$ with $s \notin \vec{n}$. Because the input queue is non-empty, we must have the reduction such that $P_1 \mid s[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \twoheadrightarrow P_1' = (\nu\ \vec{n'})(Q_1 \mid \bar{s}!\langle w\rangle; Q_2 \mid s!\langle v\rangle; P_4 \mid s[\mathtt{i} : \vec{h_1'}, \mathtt{o} : \varepsilon]) \longrightarrow (\nu\ \vec{n})(Q_1 \mid Q_2 \mid s!\langle v\rangle; P_4 \mid s[\mathtt{i} : \vec{h_1}, \mathtt{o} : \varepsilon])$, with $P_3 = Q_1 \mid Q_2$. Obviously such P_1' is untypable since the endpoints of s do not have dual types. This leads to a contradiction. The rest of the cases rely on the untypability of reduction $\twoheadrightarrow$ to prove the case by contradiction. □

Appendix C. Proof for Theorem 4.1

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

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

Lemma C.1 (Soundness). $P \approx Q$ implies $P \cong Q$.

Proof.

Reduction closeness and barb observation properties are easy to be verified. The only remaining property is showing that $\approx$ is a congruence.

Congruence for the output prefix, the restriction prefix, the conditional and recursion constructs are easy to be verified. The input congruence is similar to output congruence, since we are dealing with programs, which are processes without free variables. We give the result for congruence of the parallel operator.

Parallel Congruence
Assume relation

$$\mathscr{S} = \{((\nu\ \tilde{a}, \tilde{s})(P \mid R), (\nu\ \tilde{a}, \tilde{s})(Q \mid R)) \ \mid\ P \approx Q, \forall R \cdot P \mid R, Q \mid R \text{ are typable}, \forall \tilde{a}, \tilde{s}\} \tag{1}$$

We show that $\mathscr{S}$ is a typed relation.
Since $P \approx Q$ we have that $\Gamma \vdash P \triangleright \Delta$ and $\Gamma \vdash Q \triangleright \Delta'$ with $\Delta \leftrightharpoons \Delta'$. Since P, Q are localised and R is localised and $P \mid R, Q \mid R$ are typable then $\mathtt{dom}(\Delta) \cap \mathtt{dom}(\Delta') = \emptyset$. Using [Conc] and the $*$ definition, we obtain the result.

We show that S is a bisimulation. There are three cases:

Case (1) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \xrightarrow{\ell} P' \mid R \triangleright \Delta_1'$. Then $\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta_P'$.
By the definition of $\mathscr{S}$, we have that $\Gamma \vdash Q \triangleright \Delta_Q \stackrel{\ell}{\Longrightarrow} Q' \triangleright \Delta_Q'$. Thus we have $\Gamma \vdash Q \mid R \triangleright \Delta_2 \stackrel{\ell}{\Longrightarrow} Q' \mid R \triangleright \Delta_2'$.
Case (2) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \xrightarrow{\ell} P \mid R' \triangleright \Delta_1'$. Then $\Gamma \vdash R \triangleright \Delta_R \xrightarrow{\ell} R' \triangleright \Delta_R'$.

By the above, we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \xrightarrow{\ell} Q \mid R' \triangleright \Delta_2'$. By $\Delta_1' \leftrightharpoons \Delta_2'$, we conclude $P \mid R \approx Q \mid R$ as required.

Case (3) Suppose $\Gamma \vdash P \mid R \triangleright \Delta_1 \longrightarrow (\nu\ \tilde{a},\tilde{s})(P' \mid R') \triangleright \Delta_1'$. Then we have

$$\Gamma \vdash P \triangleright \Delta_P \xrightarrow{\ell} P' \triangleright \Delta_P' \tag{2}$$

By the definition of S, we have:

$$\Gamma \vdash Q \triangleright \Delta_Q \Longrightarrow\xrightarrow{\ell}\Longrightarrow Q' \triangleright \Delta_Q' \tag{3}$$

By (3), we have that $\Gamma \vdash Q \mid R \triangleright \Delta_2 \Longrightarrow (\nu\ \tilde{a},\tilde{s})(Q' \mid R') \triangleright \Delta_2'$. Then by $\Delta_1' \leftrightharpoons \Delta_2'$, we have $P \mid R \approx Q \mid R$, as required. □

The proof for the completeness direction follows the technique shown in (Hennessy, 2007). However we need to adapt it to session and buffers.

Definition C.1 (definability). Let N be a set of shared and session names. An external action ℓ is *definable* if for a set of names N, action $\mathrm{succ}, \notin N$ there is a *testing process* $T\langle N, \mathrm{succ}, \ell\rangle$ with the property that for every process P and $\mathtt{fn}(P) \subseteq N$,

— $\Gamma \vdash P \triangleright \Delta_1 \xrightarrow{\ell} P' \triangleright \Delta_1'$ implies that
$\Gamma \vdash T\langle N, \mathrm{succ}, \ell\rangle \mid P \triangleright \Delta \twoheadrightarrow (\nu\ \mathtt{bn}(\ell), b)(\mathrm{succ}[\mathtt{o} : \mathtt{bn}(\ell)] \mid R \mid P') \triangleright \Delta'$.

— $\Gamma \vdash T\langle N, \mathrm{succ}, \ell\rangle \mid P \triangleright \Delta \twoheadrightarrow Q \triangleright \Delta'$, where $Q \Downarrow_{\mathrm{succ}}$ implies that
$Q = (\nu\ \mathtt{bn}(\ell), b)(\mathrm{succ}[\mathtt{o} : \mathtt{bn}(\ell)] \mid R \mid P')$ where $\Gamma \vdash P \triangleright \Delta_1 \stackrel{\ell}{\Longrightarrow} P' \triangleright \Delta_1'$.

$R = b(x).R'$ or $R = \mathbf{0}$. Note that $b(x).R$ is used to keep the composition $P \mid T\langle N, \mathrm{succ}, \ell\rangle$ typable. Also $R \stackrel{l}{\not\longrightarrow}$ either due to the restriction of b, or because $R = \mathbf{0}$.

Lemma C.2. Every external action is definable.

Proof. The input action cases are straightforward:

1 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{a\langle s\rangle} P' \triangleright \Delta'$ then $T\langle \emptyset, \mathrm{succ}, a\langle s\rangle\rangle = \overline{a}(x).R \mid \mathrm{succ}[\mathtt{o} : \mathtt{tt}]$.

2 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s?\langle v\rangle} P' \triangleright \Delta'$ then $T\langle \emptyset, \mathrm{succ}, s?\langle v\rangle\rangle = (\nu\ b)(s!\langle v\rangle; b(x).R) \mid \mathrm{succ}[\mathtt{o} : \mathtt{tt}]$.

3 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s\&l} P' \triangleright \Delta'$ then $T\langle \emptyset, \mathrm{succ}, s\&l\rangle = (\nu\ b)(s \oplus l; b(x).R) \mid \mathrm{succ}[\mathtt{o} : \mathtt{tt}]$.

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

Output actions cases:

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

$$\begin{array}{rcl} T\langle\{s\}, \mathrm{succ}, a\langle s\rangle\rangle & = & (\nu\ b)(a(x).(\mathtt{if}\ x = s\ \mathtt{then}\ \mathrm{succ}!\langle x\rangle; R\ \mathtt{else}\ b(x).\mathrm{succ}!\langle x\rangle; R)) \mid \\ & & \mathrm{succ}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \mid a[\varepsilon] \end{array}$$

2 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s!\langle b\rangle} P' \triangleright \Delta'$ then we have that

$$\begin{array}{rcl} T\langle\{b\}, \mathrm{succ}, s!\langle b\rangle\rangle & = & (\nu\ b)(s?(x); (\mathtt{if}\ x = b\ \mathtt{then}\ \mathrm{succ}!\langle x\rangle; b(x).R\ \mathtt{else}\ b(x). \\ & & (\mathrm{succ}!\langle x\rangle; R)) \mid \mathrm{succ}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon] \end{array}$$

3 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s!(b)} P' \triangleright \Delta'$ then we have that

$$T\langle\{b\}, \mathrm{succ}, s!(b)\rangle \quad = \quad (\nu\ b)(s?(x); (\mathtt{if}\ x = b\ \mathtt{then}\ \mathrm{succ}!\langle x\rangle; b(x).R\ \mathtt{else}\ b(x).\\ (\mathrm{succ}!\langle x\rangle; R)) \mid \mathrm{succ}[\mathtt{i} : \varepsilon, \mathtt{o} : \varepsilon]$$

4 If $\Gamma \vdash P \triangleright \Delta \xrightarrow{s \oplus l_k} P' \triangleright \Delta'$ then we have that:

$$T\langle\emptyset, \mathrm{succ}, s \oplus l_k\rangle \quad = \quad (\nu\ b)(s\&\{l_k : \mathrm{succ}!\langle \mathtt{tt}\rangle; R, l_i : b(x).R\}_{i\in I}), 1 \leq i \leq n$$

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

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

$$\Gamma \vdash (\nu \vec{a}, \vec{s}, b)(P \mid succ[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta_1 \cong (\nu \vec{a}, \vec{s}, b)(Q \mid succ[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta_2 \tag{4}$$

then

$$\Gamma \vdash P \triangleright \Delta_P \cong Q \triangleright \Delta_Q \tag{5}$$

Proof. Let relation

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

We will show that the contextual properties hold in $\mathscr{S}$.

Typing: It should hold that $\mathscr{S}$ is a typed relation. From the definition of $\mathscr{S}$, we have that $\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta \approx (\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta', \Delta \leftrightharpoons \Delta'$. From here, by using typing rules (Nres), (Sres), (Conc), we get the required result.

Reduction Closedness: $\mathscr{S}$ is reduction closed by the freshness of succ. We cannot observe a reduction on succ or on $b(x).R$, so we conclude that if
$\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta \longrightarrow (\nu\ \tilde{a}, \tilde{s}, b)(P' \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta'$ implies
$\Gamma \vdash (\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta \twoheadrightarrow (\nu\ \tilde{a}, \tilde{s})(Q' \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \triangleright \Delta'$ then
$\Gamma \vdash P \triangleright \Delta_1 \longrightarrow P' \triangleright \Delta_P$ implies $\Gamma \vdash Q \triangleright \Delta_1 \twoheadrightarrow Q' \triangleright \Delta_Q$

Preserve Observation: We do a case analysis on the cases where $P \downarrow_m$.

If $P \downarrow_m$, $m \notin \tilde{a} \cdot \tilde{s}$ and $(\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \downarrow_m$ then $(\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathtt{o} : a] \mid b(x).R) \Downarrow_m$. From the definition of $\mathscr{S}$ and the freshness of succ, we conclude $Q \Downarrow_m$.

If $P \downarrow_m$, $m \notin \tilde{a} \cdot \tilde{s}$ and $(\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \not\downarrow_m$ then by the environment typing transition we have that m is a session occurring free in $\mathrm{succ}[\mathtt{o} : a'] \mid b(x).R$, and also $(\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathtt{o} : a] \mid b(x).R) \not\Downarrow_m$. The case where $Q \not\Downarrow_m$ does not hold, because it would be possible to have $(\nu\ \tilde{a}, \tilde{s}, b)(Q \mid \mathrm{succ}[\mathtt{o} : a] \mid b(x).R) \mid Q'$ with Q' having as a free name session m and have a typable process. But composition $(\nu\ \tilde{a}, \tilde{s}, b)(P \mid \mathrm{succ}[\mathtt{o} : a'] \mid b(x).R) \mid Q'$ is untypable because

$P\downarrow_m$, thus breaking reduction congruence. This results to the conclusion that $Q\downarrow_m$.

Context Property: The interesting case is the parallel composition. We will show that if $\Gamma\vdash P\triangleright\Delta_P \mathscr{S}\Gamma\vdash Q\triangleright\Delta_Q$. Then for arbitrary process R we have that $\Gamma\vdash P\mid P_1\triangleright\Delta'_P\ \mathscr{S}\ Q\mid P_1\triangleright\Delta'_Q$

To show this, it is enough to show that
$\Gamma\vdash(\nu\ \tilde{a},\tilde{s},b)(P\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta''_P\cong(\nu\ \tilde{a},\tilde{s},b)(Q\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta''_Q$, considering that succ may occur in P_1 and succ$'$ is fresh.

To prove this assume the process $T\langle\emptyset,\mathrm{succ}',\ell\rangle=\mathrm{succ}?(x);(\mathrm{succ}'!\langle x\rangle;\mathbf{0}\mid P'_1)\mid\mathrm{succ}'[\mathtt{i}:\varepsilon,\mathtt{o}:\varepsilon]$, where $P_1=P_1\{a'/x\}$.

From the contextual property of the theorem assumption and simple reductions, we have that:
$\Gamma\vdash(\nu\ \tilde{a},\tilde{s},b)(P\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta_1\cong\Gamma\vdash(\nu\ \tilde{a},\tilde{s},b)(Q\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta'_1$.
We need to verify that
$\Gamma\vdash(\nu\ \tilde{a},\tilde{s},b)(P\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta_1\approx(\nu\ \tilde{a},\tilde{s},b)(P\mid P_1\mid \mathrm{succ}'[\mathtt{o}:a']\mid R)\triangleright\Delta'_1$, which is simple because $R\approx\mathbf{0}$. By using Lemma C.1 we get the result. □

We are know ready to prove the completeness direction.

Lemma C.4 (Completeness). $P\cong Q$ implies $P\approx Q$

Proof. For the proof we show that if

$$\Gamma\vdash P\triangleright\Delta_P \quad\cong\quad Q\triangleright\Delta_Q \text{ and} \tag{6}$$

$$\Gamma\vdash P\triangleright\Delta_P \quad\xrightarrow{\ell}\quad P'\triangleright\Delta'_P \tag{7}$$

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

Suppose (6) and (7). Then there are two cases.
If $\ell=\tau$ then by reduction closeness of $\cong$ the result follows.
In the case where ℓ is an external action we can do a definability test for P by choosing the appropriate test $T\langle N,\mathrm{succ},l\rangle$.
Because $\cong$ is context preserving we have that $\Gamma\vdash P\mid T\langle N,\mathrm{succ},l\rangle\triangleright\Delta_{PT}\cong Q\mid T\langle N,\mathrm{succ},l\rangle\triangleright\Delta_{QT}$.
By Lemma C.2 we have that $\Gamma\vdash P\mid T\langle N,\mathrm{succ},l\rangle\triangleright\Delta_{PT}\Longrightarrow(\nu\ \mathtt{bn}(\ell))(\mathrm{succ}[\mathtt{o}:\mathtt{bn}(\ell)]|P')\triangleright\Delta$ thus by the definition of $\cong$ (Definition 4.5), we have that $\Gamma\vdash T\langle N,\mathrm{succ},l\rangle\mid Q\triangleright\Delta_{QT}\Longrightarrow R\triangleright\Delta'$.
According to the second part of the Definition C.1, we can write:

$$\Gamma\vdash Q'=(\nu\ \mathtt{bn}(\ell))(\mathrm{succ}[\mathtt{o}:\mathtt{bn}(\ell)]\mid b(x).R\mid Q'')\triangleright\Delta'' \tag{8}$$

$$\Gamma\vdash Q\triangleright\Delta_Q\stackrel{\ell}{\Longrightarrow}Q'\triangleright\Delta'_Q \tag{9}$$

Now we can derive
$\Gamma\vdash(\nu\ \mathtt{bn}(\ell),b)(\mathrm{succ}[\mathtt{o}:\mathtt{bn}(\ell)]\mid b(x).R\mid P')\cong(\nu\ \mathtt{bn}(\ell),b)(\mathrm{succ}[\mathtt{o}:\mathtt{bn}(\ell)]\mid b(x).R\mid Q')\triangleright\Delta''$.
By Lemma C.3 we conclude that:

$$\Gamma\vdash P'\triangleright\Delta'_P \quad\cong\quad Q'\triangleright\Delta'_Q \tag{10}$$

We began with the assumption that $\Gamma \vdash P \triangleright \Delta_P \cong \Gamma \vdash Q \triangleright \Delta_Q$ and we concluded to (9), (10). Thus $\cong$ implies $\approx$. □

C.1. *Bisimulation Properties*

We note that, in the following proofs, by Lemma B.4, we can always assume one of the queues of the same endpoint is always empty.

C.1.1. *Proof for Lemma 4.1* Before we proceed with the proof of Lemma 4.1 we prove the following useful Lemma.

Lemma C.5.

— If $\Gamma \vdash P \triangleright \Delta \longrightarrow \xrightarrow{\ell} \Gamma \vdash P' \triangleright \Delta'$ and ℓ is an input action then $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \longrightarrow \Gamma \vdash P' \triangleright \Delta'$.
— If $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \longrightarrow \Gamma \vdash P' \triangleright \Delta'$ and ℓ is an output action then $\Gamma \vdash P \triangleright \Delta \longrightarrow \xrightarrow{\ell} \Gamma \vdash P' \triangleright \Delta'$.

Proof. For the first part there are two cases
Case (1) P has the form $P = R \mid a[\vec{s}]$. $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \longrightarrow R' \mid a[\vec{s'}] \triangleright \Delta' \xrightarrow{a?\langle s\rangle} R' \mid a[\vec{s'} \cdot s] \triangleright \Delta'$ Now we can observe $\Gamma \vdash R \mid a[\vec{s}] \triangleright \Delta \xrightarrow{a?\langle s\rangle} \Gamma \vdash R \mid a[\vec{s} \cdot s] \triangleright \Delta \longrightarrow R' \mid a[\vec{s'} \cdot s] \triangleright \Delta'$ to conclude.
Case (2) Input communication takes place on a session channel. It is similar using a session queue.
For the second there are two cases.
Case (1) The action happens on a shared name. $\Gamma \vdash P \mid \overline{a}\langle s\rangle \triangleright \Delta \xrightarrow{a!\langle s\rangle} P \triangleright \Delta' \longrightarrow P' \triangleright \Delta''$.
From this we can always conclude that $\Gamma \vdash P\overline{a}\langle s\rangle \triangleright \Delta \longrightarrow P' \mid \overline{a}\langle s\rangle \triangleright \Delta''' \xrightarrow{a!\langle s\rangle} P' \triangleright \Delta''$. Hence we conclude the case.
Case (2) Output communication takes place on a session channel. Similar arguments by using a session queue. □

We are now ready to prove Lemma 4.1.

Proof. The proof in both parts is done by induction on the length of the silent transition. The base case is trivial.

For the first part of Lemma 4.1 we have:
$\Gamma \vdash P \triangleright \Delta \Longrightarrow \longrightarrow \xrightarrow{\ell} \Longrightarrow P' \triangleright \Delta'$. We use the first part of Lemma C.5 to permute actions $\longrightarrow$ and $\xrightarrow{\ell}$ and get $\Gamma \vdash P \triangleright \Delta \Longrightarrow \xrightarrow{\ell} \longrightarrow \Longrightarrow P' \triangleright \Delta'$. Then by the use of the induction hypothesis we get $\Gamma \vdash P \triangleright \Delta \xrightarrow{\ell} \Longrightarrow \longrightarrow \Longrightarrow P' \triangleright \Delta'$ as required.

The second part of the Lemma 4.1 follows similar arguments: $\Gamma \vdash P \triangleright \Delta \Longrightarrow \xrightarrow{\ell} \longrightarrow \Longrightarrow P' \triangleright \Delta'$. We use the second part of Lemma C.5 and then the induction hypothesis to permute as required: $\Gamma \vdash P \triangleright \Delta \Longrightarrow \longrightarrow \Longrightarrow \xrightarrow{\ell} P' \triangleright \Delta'$.

□

C.1.2. *Proof for Lemma 4.2*

Proof. The proof considers induction on the length of $\Longrightarrow_s$-transition. The base case is trivial. For the inductive step, we do the following case analysis.

Case: Receive.

By the typability of P, we have that $P' = s?(x);Q \mid s[\mathtt{i} : v \cdot \vec{h}] \mid R \longrightarrow_s P'' = Q\{v/x\} \mid s[\mathtt{i} : \vec{h}] \mid R$.

From the induction step, we have that $P \approx P'$. To show that $P \approx P''$ we need to show that $P' \approx P''$. We will use the fact that bisimulation is a congruence. Consider $R \approx R$ and $s?(x);Q \mid s[\mathtt{i} : v \cdot \vec{h}] \approx Q\{v/x\} \mid s[\mathtt{i} : \vec{h}]$. Due to $s \notin \mathtt{fn}(R)$ we can compose bisimilar processes in parallel and get that $P' \approx P''$ as required.

The rest of the cases follow similar arguments. □

C.1.3. *Proof for Lemma 4.3*

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

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

C.1.4. *Proof for Lemma 4.4*

Proof. There are two cases:

Case: τ:

Follow Lemma 4.2 to get $P \approx P'$ and $P \approx P''$. The result then follows.

Case: ℓ:

Suppose that $P \xrightarrow{\ell}_s P'$ and $P \overset{\ell}{\Longrightarrow}_s P''$ implies $P \Longrightarrow_s P_1 \xrightarrow{\ell}_s P_2 \Longrightarrow_s P''$. From Lemma 4.2, we can conclude that $P \approx P_1$ and because of the bisimulation definition, we have $P' \approx P_2$ to complete we call upon Lemma 4.2 once more to get $P' \approx P''$ as required. □

C.1.5. *Proof for Lemma 4.5*

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

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

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

By using Lemma 4.1, we have that $P \mid s_1[\mathtt{o} : \vec{h_1} \cdot v_1] \mid s_2[\mathtt{i} : \vec{h_2}] \Longrightarrow_s \xrightarrow{\ell_1}_s \xrightarrow{\ell_2}_s \Longrightarrow_s P' \mid s_1[\mathtt{o} : \vec{h_1''}] \mid s_2[\mathtt{i} : \vec{h_2''}]$ and $P \mid s_1[\mathtt{o} : \vec{h_1} \cdot v_1] \mid s_2[\mathtt{i} : \vec{h_2}] \xrightarrow{\ell_2}_s \Longrightarrow_s \xrightarrow{\ell_1}_s P'' \mid s_1[\mathtt{o} : \vec{h'}] \mid s_2[\mathtt{i} : \vec{h''}]$. We use Lemmas 4.3 and 4.1 to get $P \mid s_1[\mathtt{o} : \vec{h_1} \cdot v_1] \mid s_2[\mathtt{i} : \vec{h_2}] \xrightarrow{\ell_2}_s \Longrightarrow_s \xrightarrow{\ell_1}_s P' \mid s_1[\mathtt{o} : \vec{h_1''}] \mid s_2[\mathtt{i} : \vec{h_2''}]$.

The rest of the proof is similar to Lemma 4.2.

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

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

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

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

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

Fig. 18. Labelled Transition for Session Type System with Two Buffer Endpoint Without IO

□

Appendix D. Comparison with Asynchronous/Synchronous Calculi

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

Before we prove the relations in Section 5.2, we define a behavioural theory for the asynchronous session π-calculus with two end-point queues but without $\mathtt{i}/\mathtt{o}$-queues (Gay and Vasconcelos, 2010; Coppo et al., 2007; Mostrous and Yoshida, 2009).

A labelled transition system is given in Figure 18. 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.

D.2. *Proofs for Section 5.2*

We prove the results in Section 5.2 for the two asynchronous session typed π-calculi, by either giving the bisimulation closures when a bisimulation holds or giving the counterexample when bisimulation does not hold. The results for the synchronous and asynchronous π-calculi are well-known, hence we omit.

1 **Case:** $s!\langle v\rangle;s!\langle w\rangle;P \mid s[\mathtt{o}:\varepsilon] \not\approx s!\langle w\rangle;s!\langle v\rangle;P \mid s[\mathtt{o}:\varepsilon]$

On the left hand side process we can observe a τ transition and get $s!\langle w\rangle;P \mid s[\mathtt{o}:v] \xrightarrow{s!\langle v\rangle} s!\langle w\rangle;P \mid s[\mathtt{o}:\varepsilon]$ but $s!\langle w\rangle;s!\langle v\rangle;P \mid s[\mathtt{o}:\varepsilon] \not\xRightarrow{s!\langle v\rangle}$ 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:

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

gives the result.

3 **Case:** $s?(x);s?(y);P \mid s[\mathtt{i}:\varepsilon] \not\approx s?(y);s?(x);P \mid s[\mathtt{i}:\varepsilon]$
On both processes we can observe a $s?\langle v\rangle$ transition and get $s?(x);s?(y);P \mid s[\mathtt{i}:v] \xrightarrow{\tau} s?(y);P\{v/x\} \mid s[\mathtt{i}:\varepsilon]$ and $s?(w);s?(v);P \mid s[\mathtt{i}:v] \xrightarrow{\tau} s?(x);P\{v/y\} \mid s[\mathtt{i}:\varepsilon]$. From the substitution, we have that both processes are not bisimilar.

4 **Case:** $s_1?(x);s_2?(y);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon] \approx s_2?(y);s_1?(x);P \mid s_1[\mathtt{i}:\varepsilon] \mid s_2[\mathtt{i}:\varepsilon]$

Relation

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

gives the result.

D.3. *Arrived Operators in the π-Calculi*

In this subsection, we define the 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 `arrive` operator without a slight compromise of the non-blocking input property, due to the asynchronous nature of the `arrive` 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 π-Calculus with Arrive.

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

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

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

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

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

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

In the synchronous π-calculus with `arrive` 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

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

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 transitions, an arrive inspection can happen.

The asynchronous π-calculus with `arrive` 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.

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

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

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

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

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

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

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

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

Appendix E. Selector Properties

E.1. *Proof for Proposition 6.1 (1)*

Proof. We type left and right hand side of the selectors mapping.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash P \triangleright \Delta \cdot x_r : S \cdot x_{\overline{r}} : \overline{S}}{\Gamma \vdash b(x_r).P \triangleright \Delta \cdot x_{\overline{r}} : \overline{S}}}{\Gamma \vdash \overline{b}(x_{\overline{r}}).b(x_r).P \triangleright \Delta}}{\Gamma \vdash \overline{b}(x_{\overline{r}}).b(x_r).P \mid b[\varepsilon] \triangleright \Delta \cdot b}}{\Gamma \vdash (\nu\, b)(\overline{b}(x_{\overline{r}}).b(x_r).P \mid b[\varepsilon]) \triangleright \Delta}$$

The above result agrees with the typing rule [Selector].

$$\dfrac{\Gamma \vdash P \triangleright \Delta \cdot s : S \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}{\Gamma \vdash \overline{r}!\langle s\rangle;P \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}$$

The above result conisides with the typing rule [Reg].

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash P \triangleright \Delta \cdot s : S \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}{\Gamma \vdash \texttt{if arrive}\ x\ \texttt{then}\ P\ \texttt{else}\ \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}}{\Gamma \vdash r?(x);\texttt{if arrive}\ x\ \texttt{then}\ P\ \texttt{else}\ \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}}{\Gamma \vdash \mu Select.r?(x);\texttt{if arrive}\ x\ \texttt{then}\ P\ \texttt{else}\ \overline{r}!\langle x\rangle;Select \triangleright \Delta \cdot \overline{r} : \mu X.?(S);X \cdot \overline{r} : \mu X.!\langle S\rangle;X}$$

The above result conisides with the typing rule [Select].

□

E.2. *Selector Properties*

For the following proofs, we let $B_i = s_i[\mathtt{i} : h_i, \mathtt{o} : h_i']$.

Definition E.1. $s[\mathtt{i} : \vec{h_i'} \cdot \vec{h_i}, \mathtt{o} : \vec{h_o'} \cdot \vec{h_o}] \succ s[\mathtt{i} : \vec{h_i}, \mathtt{o} : \vec{h_o'} \cdot \vec{h_o}]$ wher $\exists P \cdot \Gamma \vdash P \mid B_i \triangleright \Delta \Longrightarrow \Gamma \vdash Q \mid B_j \triangleright \Delta$

Lemma E.1. Let $B_i \succ B_j$ and assume ℓ is a visible action. Then $\Gamma \vdash P \mid B_i \triangleright \Delta \xrightarrow{\ell} P' \mid B_i' \triangleright \Delta'$ iff $\Gamma \vdash P \mid B_j' \triangleright \Delta \xrightarrow{\ell} P' \mid B_j' \triangleright \Delta'$.

Proof. The lemma is proved by the definitions of the label transition system and environment transition. □

Definition E.2.

$$\begin{array}{lrcl} \mathsf{IfSel}_i^n = \texttt{def} & X_1 & = & \texttt{if arrive}\ s_1\ \texttt{then}\ C_1[X_2]\ \texttt{else}\ X_2 \\ & & \vdots & \\ & X_n & = & \texttt{if arrive}\ s_n\ \texttt{then}\ C_n[X_1]\ \texttt{else}\ X_1 \quad \texttt{in}\ X_i \end{array}$$

with $C_i = \texttt{typecase}\ s_i\ \texttt{of}\ \{(x_i : S_i) : R_{ij}; -\}_{1\le i\le n, 1\le j\le m}$ where $R_{ij}\{s_i/x_i\}$ is a blocking prefixed sequential series of actions with no blocking terms other than its prefix. Furthermore $R_{ij}\{s_i/x_i\}$ is session determinate.

The next definition is used in the proofs.

Definition E.3. We define $\mathsf{IfSel}_i^{'n} \mid \prod_{1\le i\le n} B_i'$ and $\mathsf{Sel}_i^{'n} \mid \prod_{1\le i\le n} B_i'$ as

1 $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \Longrightarrow \mathsf{IfSel}_i^{'n} \mid \prod_{1\le i\le n} B_i' \triangleright \Delta' \Longrightarrow \mathsf{IfSel}_{i+1}^n \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta''$

2 $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \Longrightarrow \mathsf{Sel}_i^{'n} \mid \prod_{1\le i\le n} B_i' \triangleright \Delta' \Longrightarrow \mathsf{Sel}_{i+1}^n \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta''$.

Lemma E.2. $\mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i \ \approx\ \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i$

Proof. By unfolding Sel_i^n n times we can see the bisimulation relation between the two processes. Consider relation $\mathscr{R}$, such that:

$$\begin{array}{rcl}\mathscr{R} = \{(P,Q) & \mid & P = \mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i,\ Q = \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i \\ & & P = \mathsf{IfSel}_i^{'n} \mid \prod_{1\le i\le n} B_i',\ Q = \mathsf{Sel}_{i+1}^n \mid \prod_{1\le i\le n} B_i, \\ & & P = \mathsf{IfSel}_{i+1}^n \mid \prod_{1\le i\le n} B_i,\ Q = \mathsf{Sel}_i^{'n} \prod_{1\le i\le n} B_i'\}\end{array}$$

where $B_i \succ B_i'$. For visible actions, $\ell \neq \tau$ we can use part 1 of Lemma E.1 to obtain

$\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta_{if} \xrightarrow{\ell} \mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B_j' \mid \ldots \mid B_n \triangleright \Delta_{if}'$ if and only if

$\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta_{sel} \xrightarrow{\ell} \mathsf{Sel}_i^n \mid B_1 \mid \ldots \mid B_j' \mid \ldots \mid B_n \triangleright \Delta_{sel}'$ and the resulting pair of processes to be in $\mathscr{R}$ as required.

The result is the same for the other two defining pairs of $\mathscr{R}$.

For $\ell = \tau$ we obtain if $\Gamma \vdash \mathsf{IfSel}_i^{'n} \mid \prod_{1\le i\le n} B_i' \triangleright \Delta_{if} \xrightarrow{\tau} \mathsf{IfSel}_i^{''n} \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta_{if}'$ then $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta_{sel} \Longrightarrow \mathsf{Sel}_{i+1}^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta_{sel}$ and the resulting pair of processes to be in $\mathscr{R}$ as required.

For the symmetric direction, we obtain if $\Gamma \vdash \mathsf{Sel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta_{sel} \xrightarrow{\tau} \mathsf{Sel}_i^{'n} \mid \prod_{1\le i\le n} B_i' \triangleright \Delta_{sel}$ then $\Gamma \vdash \mathsf{IfSel}_i' \prod_{1\le i\le n} B_i' \triangleright \Delta_{if} \stackrel{\tau}{\Longrightarrow} \mathsf{IfSel}_{i+1}^n \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta_{if}'$ and the resulting pair of processes to be in $\mathscr{R}$ as required.

The result is the same for the other two defining pairs of $\mathscr{R}$. □

The selectors enjoy the confluence property.

Lemma E.3.

1 $\mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B_n$ is confluent.

2 $\mathsf{Sel}_i^n \mid B_1 \mid \ldots \mid B_n$ is confluent.

Proof. We prove the first part. The second part is a direct consequence from Lemma E.2 and the fact that bisimulation preserves confluence.

We apply the confluence definition on $\mathsf{IfSel}_i^n \mid B_1 \mid \ldots \mid B_n$ on all possible pairs of ℓ_1 and ℓ_2. Then we have: $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xrightarrow{\ell_1} \stackrel{\widehat{\ell_2 \lfloor \ell_1}}{\Longrightarrow} \mathsf{IfSel}_j^{'n} \mid \prod_{1\le i\le n} B_i' \triangleright \Delta'$ and $\Gamma \vdash \mathsf{IfSel}_i^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \stackrel{\ell_2}{\Longrightarrow} \stackrel{\widehat{\ell_1 \lfloor \ell_2}}{\Longrightarrow} \mathsf{IfSel}_k^{'n} \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta''$.

Hence we need to show that $\mathsf{IfSel}_j^{'n} \mid \prod_{1\le i\le n} B_i' \approx \mathsf{IfSel}_k^{'n} \mid \prod_{1\le i\le n} B_i''$.

Consider relation $\mathscr{R} = \mathscr{S} \cup \{(\mathsf{IfSel}_j^{'n} \mid \prod_{1\le i\le n} B_i', \mathsf{IfSel}_k^{'n} \mid \prod_{1\le i\le n} B_i'')\}$, where

$$\mathscr{S} \quad = \quad \{(P,Q),(Q,P) \mid P = \mathsf{IfSel}_i^{'n} \mid \prod_{1\le i\le n} B_i', Q = \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i,\ B_i \succ B_i'\}$$

If $\Gamma \vdash \mathsf{IfSel}_i'^n \mid \prod_{1\le i\le n} B_i' \triangleright \Delta' \xrightarrow{\ell} \mathsf{IfSel}_i''^n \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta''$ then $\Gamma \vdash \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xRightarrow{\hat{\ell}} \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i''' \triangleright \Delta'''$ and the resulting process are related by $\mathscr{S}$.

For the symmetric case, $\Gamma \vdash \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xrightarrow{\ell} \mathsf{IfSel}_1'^n \mid \prod_{1\le i\le n} B_i''' \triangleright \Delta'''$ then $\Gamma \vdash \mathsf{IfSel}_i'^n \mid \prod_{1\le i\le n} B_i' \triangleright \Delta' \xRightarrow{\ell} \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i'' \triangleright \Delta''$ and the resulting process are related by $\mathscr{S}$. □

E.3. *Proof of Lemma 6.1*

Proof. By Lemma E.2, consider the equivalences,
$\mathsf{IfSel}_k^n \mid \prod_{1\le i\le n} B_i \approx \mathsf{Sel}_k^n \mid \prod_{1\le i\le n} B_i$ and $\mathsf{PermIfSel}_k^n \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermSel}_k^n \mid \prod_{1\le i\le n} B_i$. We will show that $\mathsf{IfSel}_k^n \mid \prod_{1\le i\le n} B_i \approx \mathsf{PermIfSel}_k^n \mid \prod_{1\le i\le n} B_i$, by exploiting Lemma E.3 to build a confluent up-to relation.

Consider the relation $\mathscr{R} = \mathscr{S} \cup \{(\mathsf{IfSel}_k^n \mid \prod_{1\le i\le n} B_i, \mathsf{PermIfSel}_k^n \mid \prod_{1\le i\le n} B_i)\}$ such that $\mathscr{S} = \{(\mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i, \mathsf{PermIfSel}_1^n \mid \prod_{1\le i\le n} B_i)\}$.
If $\Gamma \vdash \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xrightarrow{\ell} \Longrightarrow \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i' \triangleright \Delta'$ then $\Gamma \vdash \mathsf{PermIfSel}_1^n \mid \prod_{1\le i\le n} B_i \triangleright \Delta \xRightarrow{\ell} \mathsf{IfSel}_1^n \mid \prod_{1\le i\le n} B_i' \triangleright \Delta'$ and both resulting processes are in $\mathscr{S}$.

The symmetric case is similar. Then the proof is complete with Lemma E.2. □

E.4. *Proof of Lemma 6.2*

First by the similar technique as the static selector, we prove:

Lemma E.4. $\mathsf{DSel}_i^n \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i$ is confluent.

Then the rest is proved by constructing the up to relation of

$$\mathscr{R} = \mathscr{S} \cup \{\mathsf{DSel}_k^n \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i,\ \mathsf{PermDSel}_k^n \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i\}$$

where $\mathscr{S} = \{(\mathsf{DSel}_1^n \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i,\ \mathsf{PermDSel}_1^n \mid a[\vec{s}] \mid \prod_{1\le i\le n} B_i\}$, using a similar confluence property as done in the proof of Lemma 6.1.

Appendix F. Thread Elimination Transform Properties

We establish an equivalence result between recursive and the replicated processes.

Lemma F.1. $\mathtt{def}\ X = C[X]\ \mathtt{in}\ X \approx *(c.C[\overline{c}]) \mid \overline{c}$, where C does not contain X.

Proof. $*P$ is defined to be $\mu Y.(P \mid Y)$, so we rewrite $*c.C[\overline{c}]$ to $\mu Y.(c.C[\overline{c}] \mid Y)$. $\mu Y.P$ is defined as $\mathtt{def}\ Y \stackrel{\mathtt{def}}{=} P\ \mathtt{in}\ Y$. So $\mu Y.(c.C[\overline{c}] \mid Y)$ can be written as $\mathtt{def}\ Y \stackrel{\mathtt{def}}{=} c.C[\overline{c}] \mid Y\ \mathtt{in}\ Y$.

We can build a bisimulation relation on the transitions of context C.

$$\begin{aligned}\mathscr{R} \quad = \quad & \{(P,Q),(Q,P) \mid \\ & P = \mathtt{def}\ X \stackrel{\mathtt{def}}{=} C[X]\ \mathtt{in}\ C'[X], Q = \mathtt{def}\ Y \stackrel{\mathtt{def}}{=} c.C[\overline{c}] \mid Y\ \mathtt{in}\ C'[\overline{c}] \mid Y \\ & P = \mathtt{def}\ X \stackrel{\mathtt{def}}{=} C[X]\ \mathtt{in}\ X, Q = \mathtt{def}\ Y \stackrel{\mathtt{def}}{=} c.C[\overline{c}] \mid Y\ \mathtt{in}\ Y \mid \overline{c}\}\end{aligned}$$

If $\texttt{def}\ X = C[X]\ \texttt{in}\ C'[X] \xrightarrow{\ell} \texttt{def}\ X = C[X]\ \texttt{in}\ C''[X]$ then $\texttt{def}\ Y = c.C[\vec{c}] \mid Y\ \texttt{in}\ C'[\vec{c}] \mid Y \xrightarrow{\hat{\ell}} \texttt{def}\ Y = c.C[\vec{c}] \mid Y\ \texttt{in}\ C''[\vec{c}] \mid Y$.

For the second pair the transition is obvious. □

A usefull definition is that of the LN-transform in recursive programming style.

Definition F.1 (LN transform-recursive programming style).

$$
\begin{array}{rcl}
LNR[\![*a(x).P]\!] & \stackrel{\texttt{def}}{=} & (\nu\ q)(\mathsf{Loop}\langle q\rangle \mid q\langle(\texttt{sd}, a, \emptyset)\rangle) \\
\mathsf{Loop}\langle q\rangle & \stackrel{\texttt{def}}{=} & \texttt{select}\ (x_s, x_a, y)\ \texttt{from}\ q\ \texttt{in}\ \texttt{if}\ x_a = a\ \texttt{then}\ \texttt{new}\ \texttt{env}\ y\ \texttt{in}\ \mathscr{B}[\![*a(x).P]\!]\ \texttt{else} \\
 & & \texttt{typecase}\ x\ \texttt{of}\ \{x_1 : S_1 : \mathscr{B}[\![P_1]\!], \ldots x_{n-m} : S_{n-m} : \mathscr{B}[\![P_{n-m}]\!]\} \\
\mathscr{B}[\![*a(x).P]\!] & \stackrel{\texttt{def}}{=} & a(w).\texttt{update}\ (y, w, w')\ \texttt{in}\ \texttt{register}\ (x_a, a, \emptyset)\ \texttt{to}\ q\ \texttt{in}\ [\![P, y]\!] \\
\mathscr{B}[\![x^{(i)}?(z:T); Q]\!] & \stackrel{\texttt{def}}{=} & x'?(z');\texttt{update}\ (y, z, z')\ \texttt{in}\ \texttt{update}\ (y, x, x')\ \texttt{in}\ [\![Q, y]\!] \\
\mathscr{B}[\![x^{(i)}\&\{l_j : Q_j\}_j]\!] & \stackrel{\texttt{def}}{=} & x'\&\{l_j :\ \texttt{update}\ (y, x, x')\ \texttt{in}\ [\![Q_j, y]\!]\}_j \\
[\![Q, y]\!] & \stackrel{\texttt{def}}{=} & \texttt{let}\ x' = [\![x]\!]_y\ \texttt{in}\ \texttt{register}\ (x', \texttt{shd}, y)\ \texttt{to}\ q\ \texttt{in}\ \mathsf{Loop}\langle q\rangle\ \ (Q \text{ is blocking at } x^{(i)}) \\
[\![\mathbf{0}, y]\!] & \stackrel{\texttt{def}}{=} & \mathsf{Loop}\langle q\rangle
\end{array}
$$

Lemma F.2. $LN[\![*(a(x).P) \mid a[\varepsilon]]\!] \approx LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$

Proof. The proof is an application of lemma F.1. Since definition $LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$ uses process variable using lemma F.1 we can substitute process variables with names c_i and their definition with $*c_i...$ to get $LN[\![*(a(x).P) \mid a[\varepsilon]]\!]$. □

The LN-transformed process is essentially a sequential process with session endpoints composed in parallel. Hence we can also establish:

Lemma F.3. $LN[\![*(a(x).P) \mid a[\varepsilon]]\!]$ is confluent.

Proof. We use lemma E.4 to show that $LNR[\![*(a(x).P) \mid a[\varepsilon]]\!]$ is confluent then by lemma F.2 and the fact that bisimulation preserves confluence, we get the required result. □

We can know study the behaviour of the LN-transform.

Lemma F.4 (Event Server Permutation).
Let

$$
\begin{array}{rcl}
P_1 & = & (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle \ldots, (s_i, a_i, y_i, c_i), (s_j, a_i, y_j, c_j), \ldots\rangle \mid \\
 & & a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m[\texttt{i} : \vec{h_{im}}, \texttt{o} : \vec{h_{om}}])
\end{array}
$$

and

$$
\begin{array}{rcl}
P_2 & = & (\nu\ \vec{c}or)(\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle \ldots, (s_j, a_i y_j, c_j), (s_i, a_i, y_i, c_i), \ldots\rangle \mid \\
 & & a[\vec{s}] \mid \textstyle\prod_{m\in I} s_m[\texttt{i} : \vec{h_{im}}, \texttt{o} : \vec{h_{om}}])
\end{array}
$$

Then $P_1 \approx P_2$.

Proof. The first step is by Definition F.1. We then apply Lemmas 6.2 and F.2. □

We finaly prove our main theorem

Theorem F.1. $*a(x).P \mid a[\varepsilon] \approx LN[\![*a(x).P \mid a[\varepsilon]]\!]$

Proof.
Since both processes are confluent we can develop a confluent up-to relation along with lemma F.4 to prove bisimulation closure.
Let relation $\mathscr{R}$ such that

$$\begin{array}{rcll} \mathscr{R} & = & \{(P_1,P_2),(P_2,P_1) \mid & P_1 = *a(x).P \mid \prod_{1\le i\le n} R_i \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}] \\ & & & R_1,\ldots,R_n \text{ blocking subterms of } P \\ & & & P_2 = \mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle s_j,\ldots,s_{j-1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]\} \end{array}$$

Then we prove that $\mathscr{R}$ is a bisimulation up-to confluence. For observable actions, the bisimulation holds trivially since if $P_1 \xrightarrow{\ell} P_1'$ then $P_2 \xrightarrow{\ell} P_2'$ and $P_1'\mathscr{R}P_2'$.

Let $P_2 \longrightarrow Q' \mid \mathsf{CodeBlocks} \mid r\langle s_j,\ldots,s_{j-1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$ then
$P_1 \Longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_j' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$, where $R_j \Longrightarrow R_j'$ and R_j' is a blocking server subterm of P and $Q' \mid \mathsf{CodeBlocks} \mid r\langle s_j,\ldots,s_{j-1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}] \Longrightarrow$
$\mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle s_{j+1},\ldots,s_j\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$.

For the symmetric case, if $P_1 \longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_i' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$ then we choose a processe $P_2' \approx P_2$ (from Lemma 6.2) such that
$P_2' = \mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle s_i, s_j \ldots, s_{j+1}\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$.

Now we can observe $P_2' \Longrightarrow \mathsf{Loop} \mid \mathsf{CodeBlocks} \mid r\langle s_j,\ldots,s_i\rangle \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$ and
$*a(x).P \mid R_1 \mid \ldots \mid R_i' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}] \Longrightarrow *a(x).P \mid R_1 \mid \ldots \mid R_i'' \mid \ldots \mid R_n \mid \prod_{1\le i\le n} B_i \mid a[\vec{s}]$
where R_i'' is a blocking subterm of P.

This completes the proof.

□