View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Università di Cagliari

A semantic deconstruction of session types*

Massimo Bartoletti[1], Alceste Scalas[1], and Roberto Zunino[2]

[1] Università degli Studi di Cagliari, Italy — {bart , alceste.scalas}@unica.it
[2] Università degli Studi di Trento, Italy — roberto.zunino@unitn.it

Abstract We investigate the semantic foundations of session types, by revisiting them in the abstract setting of labelled transition systems. The crucial insight is a simulation relation which generalises the usual syntax-directed notions of typing and subtyping, and encompasses both synchronous and asynchronous binary session types. This allows us to extend the session types theory to some common programming patterns which are not typically considered in the session types literature.

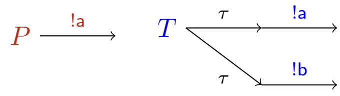
1 Introduction

Session typing is a well-established approach to the problem of correctly designing distributed applications [20,21,28]. In a nutshell, the application designer specifies the overall communication behaviour through a *choreography*, which enjoys some correctness properties (e.g. safety and progress). The overall application is the result of the composition of a set of *processes*, which are distributed over the network and interact through *sessions*. To ensure the correctness of this composition, the choreography is projected into a set of *session types*, which abstract the end-point communication behaviour of processes: if each process is type-checked against its session type, the composition of services preserves the properties enjoyed by the choreography.

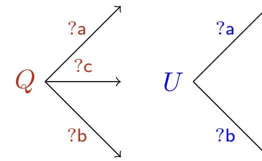
The usual technical tool used to prove the correctness of a behavioural type system is *subject reduction*. Say P is a process, and T is a session type. Roughly, subject reduction guarantees that, if we have a typing judgement $\vdash P : T$, then whenever P takes a computation step $P \xrightarrow{\ell} P'$, also the type can take a similar step, i.e. there exists some T' such that $T \xrightarrow{\ell} T'$ and $\vdash P' : T'$.

This relation between processes and types somehow resembles the *simulation* relation in labelled transition systems (LTSs): a state T simulates a state P iff, whenever $P \xrightarrow{\ell} P'$, then $T \xrightarrow{\ell} T'$, for some T' which still simulates P'. This seems to suggest that $\vdash P : T$ is rooted in some kind of "process-type simulation". To elaborate further on this insight, consider a session type $T = \mathsf{!a} \oplus \mathsf{!b}$, which models an *internal* choice between two outputs.

* Work partially supported by: Aut. Reg. Sardinia (L.R.7/07 *TRICS*, P.I.A.2010 *Social Glue*), MIUR PRIN 2010-11 *Security Horizons*, EU COST Action IC1201 *BETTY*.



We can refine this session type as the process P = !a which just wants to output !a. Intuitively, the process P respects the type T, because any client who can handle both choices in T will interact correctly with P. Now, let us consider the LTSs of P and T (on the left): we can observe that P is (weakly) simulated by T, in symbols $P \precsim T$, because each move of P is matched by a move of T.



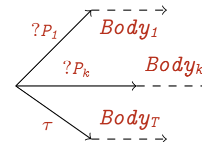
Let us now consider the type U = ?a & ?b, which models an *external* choice between two inputs, and let Q = ?a + ?b + ?c (where + is the standard CCS choice operator) which allows for an additional input ?c. Again, Q respects U: any client compatible with U will not exploit the additional choice, and will interact correctly with Q. But let us look at the LTSs of Q and U (on the right): differently from the previous case, now we have that Q is *not* weakly simulated by U (whereas the converse $U \precsim Q$ holds). This shows that the weak simulation relation does not faithfully capture the notion of session typing: indeed, the previous examples suggest that a hypothetical "process-type simulation" should treat input and output capabilities differently: intuitively, it should be *covariant* w.r.t. outputs and *contravariant* w.r.t. inputs.

A similar kind of co/contra-variance arises when dealing with *subtyping*. The intuition is that if a session type T is subtype of U, and we have two processes P, Q such that $\vdash P : T$ and $\vdash Q : U$, then P can safely "replace" Q: i.e., each process that interacts correctly with Q will also interact correctly with P. Again, the session subtyping relations (e.g. [18]) are *covariant w.r.t. outputs and contravariant w.r.t. inputs*; moreover, they are *coinductive*. This suggests a link between the subtyping relation and our hypothetical "process-type simulation".

Several papers have studied session typing relations (e.g. [7,8,11,19,20,21,23]) and subtype preorders (e.g. [1,2,9,10,12,15,18]). Despite the variety of aims and results, all these works share a common approach: fix some syntax for types and/or processes, and then characterise typing/subtyping through *syntax-driven* definitions, usually in the form of a type system, or coinductive definitions (for subtyping). This seems in slight contrast with a common principle in concurrency theory: keeping syntax separated from semantics. Indeed, behavioural equivalences (e.g. (bi)simulation, testing, *etc.*) are typically defined over arbitrary LTSs, and then applied to calculi by providing the latter with an LTS semantics [27].

Another drawback of these syntax-driven approaches is that they do not usually consider some common programming patterns for interactive applications. For example, let us think about a server waiting for client's input: typically, the server must handle the case where such inputs do not arrive. This can be achieved via signals/exceptions handling, or other programming language constructs. In Erlang , for instance, one can write:

```
receive  P1 -> Body1...
         Pk -> Bodyk
after    10 -> BodyT
```



This causes `receive` to be aborted if no messages matching the patterns $P_1,\ldots,P_k$ arrive within 10 milliseconds; in this case, $Body_T$is executed — where the program may e.g. do internal actions and start receiving again. Such a program blurs the distinction between internal/external choices: intuitively, its LTS (on the right) has a state with external inputs $?P_1,\ldots,?P_k$ and an internal τ-move abstracting the timeout. This eludes the notion of *"structured communication-based programming"* at the roots of the session types approach [19,20]; yet, it is a use case that one would like to somehow typecheck to ensure correct interaction.

In this work, we tackle these problems by revisiting the semantic foundations of session types, aiming for behavioural, syntax-independent relations and properties that can be later applied to specific process calculi and programming languages.

Contributions. We study a behavioural theory of session types, aimed at unifying the notions of typing and subtyping, including both synchronous/asynchronous semantics. We start in §2 by setting our framework, and giving a running example. In §3 we define *I/O compliance* as a notion of correct interaction between behaviours, stricter than progress, albeit coinciding with it on synchronous session types (Theorem 1). In §4 we introduce the *I/O simulation* $\ddot{\leqslant}$ between behaviours, which is an I/O compliance-preserving preorder (Theorems 3 and 4), is a Gay-Hole subtyping relation [18] (Theorem 5), and is preserved when passing from synchronous to asynchronous session types semantics (Theorem 6). In §5 we show that $\ddot{\leqslant}$ induces syntax-driven type systems, which guarantee correct interaction (Theorem 8). Due to space constraints, the proofs of all our statements, more examples and discussion are available in [5].

2 Behaviours

In this section we exploit the semantic model of labelled transition systems (LTSs) to provide a unifying ground for the notions developed later. We consider LTSs where labels are partitioned into internal, input, and output actions, and we call *behaviours* the states of such LTSs. Then, we exploit this model to embed three calculi for concurrency: binary session types with synchronous or asynchronous semantics, and asynchronous CCS. We will sometimes use these calculi to write examples and to discuss related work, but all the main technical notions and results do apply to the general class of behaviours.

We consider an LTS $(\mathcal{U}, \mathsf{A}_\tau, \{\xrightarrow{\ell_\tau} \mid \ell_\tau \in \mathsf{A}_\tau\})$, where $\mathcal{U} = \{p, q, \ldots\}$ is a set of *behaviours*, A_τ is a set of *labels*, and $\xrightarrow{\ell_\tau} \subseteq \mathcal{U} \times \mathcal{U}$ is a *transition relation*. A_τ is partitioned into *input actions* $\mathsf{A}^? = \{?\mathsf{a}, ?\mathsf{b}, \ldots\}$, *output actions* $\mathsf{A}^! = \{!\mathsf{a}, !\mathsf{b}, \ldots\}$, and the *internal action* τ. We use an involution $\mathrm{co}(\cdot)$ such that $\mathrm{co}(?\mathsf{a}) = !\mathsf{a}$ and $\mathrm{co}(!\mathsf{a}) = ?\mathsf{a}$. We let $\ell, \ell', \ldots$ range over $\mathsf{A} = \mathsf{A}^? \cup \mathsf{A}^!$. For a set $L \subseteq \mathsf{A}$, we define $L^? = L \cap \mathsf{A}^?$ and $L^! = L \cap \mathsf{A}^!$. For all $p, q \in \mathcal{U}$, we define the *parallel composition* $p \parallel q$ as the behaviour whose transitions are given by the (standard) rules:

$$\frac{p \xrightarrow{\ell_\tau} p'}{p \parallel q \xrightarrow{\ell_\tau} p' \parallel q} \qquad \frac{q \xrightarrow{\ell_\tau} q'}{p \parallel q \xrightarrow{\ell_\tau} p \parallel q'} \qquad \frac{p \xrightarrow{\ell} p' \quad q \xrightarrow{\mathrm{co}(\ell)} q'}{p \parallel q \xrightarrow{\tau} p' \parallel q'}$$

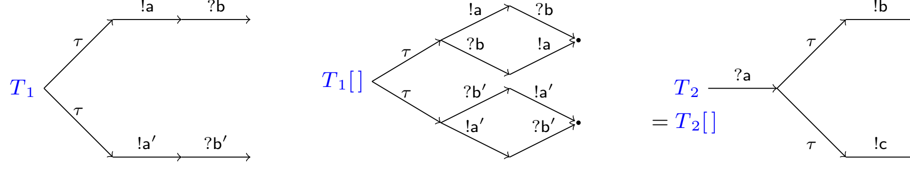


Figure 1: Three session behaviours.

We define the relation $\Rightarrow$ as the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\xRightarrow{\ell_\tau}$ as $\Rightarrow\xrightarrow{\ell_\tau}\Rightarrow$. We write $p\xrightarrow{\ell_\tau}$ when $\exists p' \,.\, p \xrightarrow{\ell_\tau} p'$; we write $p\rightarrow$ when $\exists \ell_\tau \,.\, p\xrightarrow{\ell_\tau}$. We write $\mathbf{0}$ to denote any p such that $p\not\rightarrow$. We define the *weak barbs of* p as $p\Downarrow \;=\; \{\ell \mid p\xRightarrow{\ell}\}$. Hereafter, we shall consider two behaviours equal iff their transition graphs are isomorphic (i.e. equal up-to node renaming).

Session types. A session type is an abstraction of the behaviour of a process interacting with its environment. Here, we use a simple version of session types by slightly adapting those studied in [1]. Session types comprise external choice ($\&$) among inputs ($?\mathsf{a}$), internal choice ($\oplus$) among outputs ($!\mathsf{a}$), and recursion. Empty choices (of any kind) represent successful termination.

Definition 1 (Session types). Session types *are terms with the syntax:*

$$T \;::=\; \&_{i\in I} ?\mathsf{a}_i . T_i \;\;\Big|\;\; \bigoplus_{i\in I} !\mathsf{a}_i . T_i \;\;\Big|\;\; \mathrm{rec}_X\, T \;\;\Big|\;\; X$$

where (i) the set I is finite, (ii) the actions in internal/external choices are pairwise distinct, and (iii) recursion is guarded. We write $\mathbf{0}$ for the empty choice.

We present two semantics for session types: one *synchronous* (Def. 2) and one *asynchronous* (Def. 3). In both, an internal choice first commits to one of the branches $!\mathsf{a}.T$, before enabling $!\mathsf{a}$. An external choice enables all its actions.

Definition 2 (Synchronous session behaviours). *We denote with $\mathbb{S}_s$ the set of behaviours of the form T (up-to unfolding), with transitions given by the rules:*

$$\&_{i\in I} ?\mathsf{a}_i . T_i \xrightarrow{?\mathsf{a}_j} T_j \;\; (j \in I) \qquad \bigoplus_{i\in I} T_i \xrightarrow{\tau} T_j \;\; (j \in I,\; |I| > 1) \qquad !\mathsf{a}.T \xrightarrow{!\mathsf{a}} T$$

For the asynchronous semantics, we consider behaviours of the form $T[\sigma]$ where σ is a sequence of output actions, modelling an unbounded *buffer*.

Definition 3 (Asynchronous session behaviours). *We denote with $\mathbb{S}_a$ the set of behaviours of the form $T[\sigma]$ (up-to unfolding), with transition rules:*

$$\Big(\bigoplus_{i\in I} !\mathsf{a}_i . T_i\Big)[\sigma] \xrightarrow{\tau} T_j\,[\sigma . !\mathsf{a}_j] \;\; (j \in I) \qquad T\,[!\mathsf{a}.\sigma] \xrightarrow{!\mathsf{a}} T\,[\sigma] \qquad \Big(\&_{i\in I} ?\mathsf{a}_i . T_i\Big)[\sigma] \xrightarrow{?\mathsf{a}_j} T_j\,[\sigma] \;\; (j \in I)$$

The async rule for $\oplus$ adds the selected output to the end of the buffer, with a τ-move. The 2nd rule says that an output $!\mathsf{a}$ at the head of the buffer is consumed with a $!\mathsf{a}$-transition. The async rule for $\&$ is similar to the sync one.

Example 1. Let $T_1 = !\mathsf{a}.?\mathsf{b} \oplus !\mathsf{a}'.?\mathsf{b}'$, and $T_2 = ?\mathsf{a}.(!\mathsf{b} \oplus !\mathsf{c})$. Their sync/async behaviours are shown in Figure 1. Note that T_2 has equal sync/async behaviours.

The following proposition shows that asynchronous session behaviours are not more general than synchronous ones, and *vice versa*: e.g., considering the session types in Example 1, we have that $T_1 \notin \mathcal{S}_a$, while $T_1[] \notin \mathcal{S}_s$.

Proposition 1. $\mathcal{S}_a \not\subseteq\not\supseteq \mathcal{S}_s$.

Definition 4 (CCS). CCS terms *have the following syntax:*

$$P, Q \ ::= \ \mathbf{0} \ \mid \ \ell_\tau.P \ \mid \ P + Q \ \mid \ P \mid Q \ \mid \ X \ \mid \ \mu_X P$$

where $+$ is non-deterministic choice, $|$ is parallel composition, and recursion $\mu_X P$ is guarded. Like async session behaviours, async CCS semantics use a buffer $[\sigma]$.

Definition 5 (Async CCS semantics). *We denote with* $\mathcal{P}_a$ *the set of behaviours of the form* $P[\sigma]$ *(up-to unfolding), with transitions given by the following rules (the symmetric ones for* $|$ *and* $+$ *are omitted):*

$$\frac{\ell_\tau \in \{\tau\} \cup \mathsf{A}^?}{\ell_\tau.P\,[\sigma] \xrightarrow{\ell_\tau} P[\sigma]} \qquad \begin{array}{c} !\mathsf{a}.P\,[\sigma] \xrightarrow{\tau} P[\sigma.!\mathsf{a}] \\ P\,[!\mathsf{a}.\sigma] \xrightarrow{!\mathsf{a}} P[\sigma] \end{array} \qquad \frac{P[\sigma] \xrightarrow{\ell_\tau} P'[\sigma']}{(P+Q)[\sigma] \xrightarrow{\ell_\tau} P'[\sigma']} \qquad \frac{P[\sigma] \xrightarrow{\ell_\tau} P'[\sigma']}{(P \mid Q)[\sigma] \xrightarrow{\ell_\tau} (P' \mid Q)[\sigma']}$$

As in async session behaviours, an output $!\mathsf{a}$ is first added at the end of the buffer, and can only be consumed from its head. Note that a behaviour *cannot* consume its own buffer: $|$ just allows for interleaving. Synchronization is obtained with $P[\sigma] \parallel Q[\sigma']$, i.e. using the parallel composition of LTS states: this allows P's input actions to consume Q's output buffer, and *vice versa.*

Example 2. The behaviour of the async process $!\mathsf{a}.\tau[]$ is shown as p_1 in Figure 2.

Definition 6. *We define an* encoding $[\![]\!]$ of session type terms into async CCS*:*

$$[\![\textstyle\bigoplus_I !\mathsf{a}_i.T_i]\!] = \sum_I !\mathsf{a}_i.[\![T_i]\!] \qquad [\![\&_I ?\mathsf{a}_i.T_i]\!] = \sum_I ?\mathsf{a}_i.[\![c_i]\!] \qquad [\![\mathrm{rec}_X\, T]\!] = \mu_X [\![T]\!] \qquad [\![X]\!] = X$$

By Lemma 1, an *async* session type and its encoding in async CCS are equivalent.

Lemma 1. $T[] = [\![T]\!][]$.

Proposition 2 relates async CCS behaviours with session behaviours.

Proposition 2. $\mathcal{S}_a \subsetneq \mathcal{P}_a \not\subseteq\not\supseteq \mathcal{S}_s$.

An example. The following types model a bartender (B) and a client Alice (A):

$$U_{\mathsf{B}} = \mathrm{rec}_X\,(?\mathsf{aCoffee}.!\mathsf{coffee}.X \;\;\&\;\; ?\mathsf{aBeer}.(!\mathsf{beer}.X \oplus !\mathsf{no}.X) \;\;\&\;\; ?\mathsf{pay})$$
$$T_{\mathsf{A}} = !\mathsf{aCoffee}.?\mathsf{coffee}.!\mathsf{pay} \;\;\oplus\;\; !\mathsf{aBeer}.(?\mathsf{beer}.!\mathsf{pay} \,\&\, ?\mathsf{no}.!\mathsf{pay})$$

The bartender presents an external choice $\&$, allowing a customer to order either coffee or beer, or to eventually pay; in the first case, he will serve the coffee and then recursively wait for more orders; in the second case, he uses the internal choice $\oplus$ to decide whether to serve the beer or not — and then waits for more orders; in the third case, after the due amount (possibly 0) is paid, the interaction ends. Alice internally chooses between coffee or beer; in the first case, she waits to get the coffee and then pays; in the second case, she lets the bartender choose between serving the beer, or saying no — and in both cases, she will check out.

Intuitively, U_{B} and T_{A} are compliant, and the following processes type-check:

$$Q_{\mathsf{B}} = \mu_Y(?\mathsf{aCoffee}.!\mathsf{coffee}.Y \;+\; ?\mathsf{aBeer}\,.\,(!\mathsf{beer}.Y + !\mathsf{no}.Y) \;+\; ?\mathsf{pay})$$
$$P_{\mathsf{A}} = !\mathsf{aCoffee}.?\mathsf{coffee}.!\mathsf{pay} \;+\; !\mathsf{aBeer}\,.\,(?\mathsf{beer}.!\mathsf{pay} + ?\mathsf{no}.!\mathsf{pay})$$

From typing and compliance, we can deduce that $P_{\mathsf{A}}[] \parallel Q_{\mathsf{B}}[]$ synchronize and reach the successful state $\mathbf{0}[] \parallel \mathbf{0}[]$, where they agree in stopping their interaction.

Alice may also implement a *subtype* of T_{A} only asking for coffee: $T'_{\mathsf{A}} = !\mathsf{aCoffee}.?\mathsf{coffee}.!\mathsf{pay}$, with a corresponding process $P'_{\mathsf{A}} = !\mathsf{aCoffee}.?\mathsf{coffee}.!\mathsf{pay}$. Note however that the subtyping step is not necessary: P'_{A} has also type T_{A}.

So far, the structures of A's and B's processes match the structure of their types. This is a common situation in the session types literature: processes are usually written using calculi inheriting the structured communication approach pioneered by Honda *et al.* [19,20], thus reflecting the internal/external choices of types. However, in some cases things may be more complex. The bartender might have other incumbencies, and may need to stop selling beer after a certain hour:

$$Q''_{\mathsf{B}} = \mu_Y\big(\,(?\mathsf{aCoffee}.!\mathsf{coffee}.Y \;+\; ?\mathsf{aBeer}\,.\,(!\mathsf{beer}.Y + !\mathsf{no}.Y) \;+\; ?\mathsf{pay}) \\ \quad + \tau.\mu_Z(?\mathsf{aCoffee}.!\mathsf{coffee}.Z \;+\; ?\mathsf{aBeer}.!\mathsf{no}.Z \;+\; ?\mathsf{pay})\,\big)$$

This reminds us of the small Erlang code sample given in §1: the τ branch represents the bartender's decision to stop waiting for customer orders, perform some internal duties (e.g. clean up the bar) and then serve again — this time, refusing to sell beer. Intuitively, we would like Q''_{B} to still have the type U_{B}, since compliant customer processes (e.g. Alice's one) will still be able to interact (either before or after the τ). A process like Q''_{B}, however, is usually impossible to write (and type) using classical session calculi: their grammar does not offer a τ prefix, since it would allow for processes where the distinction between internal/external choices is blurred (contrary to the expected program structure).

Let us consider another scenario: Alice is late for work. But she realises that the bartender-customer system is *asynchronous*: the counter is a bidirectional *buffer* where drinks and money can be placed. Thus, she tries to save time by implementing the following type and process:

$$T''_{\mathsf{A}} = !\mathsf{aCoffee}.!\mathsf{pay}.?\mathsf{coffee} \qquad\qquad P''_{\mathsf{A}} = !\mathsf{aCoffee}.(?\mathsf{coffee} \mid !\mathsf{pay})$$

i.e., in her type she plans to order a coffee, put her money on the counter while B prepares her drink, and take it as soon as it is ready; in her process, she orders a coffee, and tries to grab the coffee with one hand, while putting the money on the counter with the other. P''_{A} represents an optimised program exploiting buffered communication, thus diverging from the syntactic structure of T''_{A}. Therefore, is T''_{A} a type for P''_{A}? Is T''_{A} compliant with U_{B}, and will P''_{A} interact smoothly with Q_{B} and Q''_{B}? We shall answer these questions later on in §5.

3 I/O compliance

We now address the problem of defining a relation between behaviours to guarantee that, when combined together, they interact in a "correct" manner. Many different notions of correctness have been considered to this purpose in the literature, both for the binary [12,14,1,2] and for the multi-party settings [9,10,3,17].

We start by considering the classical, trace-based notion of compliance of [14,1], where correctness is interpreted as *progress* of the interaction. In Definition 7 we say that a behaviour p has progress with q (in symbols, $p \dashv q$) iff, whenever a τ-computation of the system $p \parallel q$ is stuck, then p has reached the final (success) state $\mathbf{0}$. Note that this notion is *asymmetric*, in the sense that p is allowed to terminate the interaction without the permission of q. This is intended to model the asymmetry between the role of a client p and that of a server q, as in [1].

Definition 7 (Progress). *We write* $p \dashv q$ *iff* $p \parallel q \Rightarrow p' \parallel q' \not\rightarrow$ *implies* $p' = \mathbf{0}$. *We write* $p \perp q$ *when* $p \dashv q$ *and* $p \vdash q$.

The following proposition states that, for session types, progress with the synchronous semantics implies progress with the async semantics. As we shall see, the main relations introduced in the rest of the paper will be preserved when passing from the synchronous to the asynchronous semantics of session types.

Proposition 3. *If* $T \dashv U$, *then* $T[] \dashv U[]$.

Example 3. We have the following relations:

$$\begin{array}{llll}
\mathsf{!a.?b} \perp \mathsf{?a.!b} & \mathsf{!a.?b} \not\dashv\vdash \mathsf{?a} & \mathrm{rec}_X\, \mathsf{!a}.X \not\dashv\vdash \mathsf{?a} & (\mathrm{rec}_X\, \mathsf{!a}.X)[] \perp \mathsf{?b}[] \\
\mathsf{!a.?b} \not\dashv\not\vdash \mathsf{!b.?a} & (\mathsf{!a.?b})[] \perp (\mathsf{!b.?a})[] & \mathrm{rec}_X\, \mathsf{!a}.X \perp \mathrm{rec}_Y\, \mathsf{?a}.Y & (\mathrm{rec}_X\, \mathsf{?a}.X)[] \not\dashv\vdash \mathsf{!b}[]
\end{array}$$

The progress-based notion of correctness above also relates behaviours that allow arguably incorrect interactions. For instance, $(\mathrm{rec}_X\, \mathsf{!a}.X)[] \dashv \mathsf{?b}[]$ holds, because they produce an infinite τ-trace, even if they cannot synchronise. Ideally, we would like our notion of correct interaction to be stricter, avoiding "vacuous" progress where the client p exposes I/O capabilities, but the server q cannot interact, and $p \parallel q$ merely advances via internal τ-transitions (without synchronisations). We introduce a notion of compliance enjoying such a property on general behaviours (recall from §2 that $p{\Downarrow}^! = (p{\Downarrow})^! = p{\Downarrow} \cap \mathsf{A}^!$):

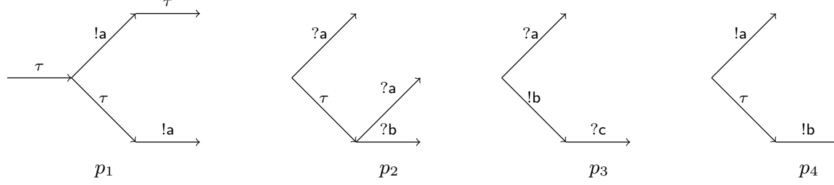


Figure 2: Four behaviours which are not session behaviours.

Definition 8. $\mathcal{R}$ *is an* I/O compliance relation *iff, when* $p \,\mathcal{R}\, q$*:*

a) $p{\Downarrow}^{!} \subseteq \mathrm{co}(q{\Downarrow}^{?}) \;\wedge\; \Big((p{\Downarrow}^{!} = \emptyset \wedge p{\Downarrow}^{?} \neq \emptyset) \implies \emptyset \neq q{\Downarrow}^{!} \subseteq \mathrm{co}(p{\Downarrow}^{?}) \Big)$;

b) $p \xrightarrow{\ell} p' \wedge q \xrightarrow{\mathrm{co}(\ell)} q' \implies p' \,\mathcal{R}\, q'$;

c) $p \xrightarrow{\tau} p' \implies p' \,\mathcal{R}\, q$;

d) $q \xrightarrow{\tau} q' \implies p \,\mathcal{R}\, q'$.

We write $\ddot{\lhd}$ *for the largest I/O compliance relation, and* $\ddot{\bowtie}$ *for the largest symmetric I/O compliance relation. When* $p \,\ddot{\bowtie}\, q$*, then we say that* p *and* q *are* I/O compliant.

Definition 8 can be interpreted with the game-theoretic metaphor. Let p and q be two players. Item *a*) has two conditions: by the leftmost constraint, if p wants to do some output (possibly after some τ-moves), then q must match it with its inputs; by the rightmost constraint, if p is *not* going to output, but wants to do some input, then q must be ready (possibly after some τ-moves) to do some output, and q cannot have outputs other than those accepted by p. I/O compliance must be preserved if p and q synchronise or do internal moves (items *b*), *c*), *d*)).

Lemma 2. $\ddot{\bowtie} \;=\; \ddot{\rhd} \cap \ddot{\lhd}$.

Example 4. Consider the behaviours in Figure 2. We have that $p_1 \,\ddot{\bowtie}\, p_2$, $p_2 \,\ddot{\bowtie}\, p_4$, $p_1 \,\ddot{\lhd}\, p_3$, and $p_2 \,\ddot{\lhd}\, p_3$, while all the other pairs of behaviours are not compliant.

Theorem 1 relates I/O compliance with Def. 7. If two behaviours are compliant, then they enjoy progress. The *vice versa* is not true: e.g., $(\mathrm{rec}_X\, \mathsf{!a}.X)\,[] \not\ddot{\lhd}\, \mathsf{?b}\,[]$, coherently with our *desideratum* that correct interactions must not progress vacuously. $\ddot{\lhd}$ can relate async session behaviours which intuitively interact correctly, e.g. $(\mathsf{!a}.\mathsf{?b})\,[] \,\ddot{\bowtie}\, (\mathsf{!b}.\mathsf{?a})\,[]$. Still, $\ddot{\lhd}$ and $\dashv$ coincide in $\mathcal{S}_s$.

Theorem 1. *If* $p \,\ddot{\lhd}\, q$ *then* $p \dashv q$*. Also, if* $p, q \in \mathcal{S}_s$ *then* $p \dashv q$ *implies* $p \,\ddot{\lhd}\, q$*.*

Example 5. Recall the example in §2. In the sync case, $U_{\mathsf{B}} \perp T_{\mathsf{A}}$, $U_{\mathsf{B}} \,\ddot{\bowtie}\, T_{\mathsf{A}}$, $U_{\mathsf{B}} \perp T'_{\mathsf{A}}$ and $U_{\mathsf{B}} \,\ddot{\bowtie}\, T'_{\mathsf{A}}$. The same holds for their async versions. When Alice is late for work, for the *sync* types $U_{\mathsf{B}} \not\perp T''_{\mathsf{A}}$ and $U_{\mathsf{B}} \not\ddot{\bowtie}\, T''_{\mathsf{A}}$, due to the wrong order of Alice's actions. In the *async* case, instead, $U_{\mathsf{B}}[] \perp T''_{\mathsf{A}}[]$ and $U_{\mathsf{B}}[] \,\ddot{\bowtie}\, T''_{\mathsf{A}}[]$.

Proposition 4 says that $\ddot{\lhd}$ is preserved when passing from sync to async session behaviour. It refines Proposition 3, that deals with the weaker notion of progress.

Proposition 4. *If* $T \,\ddot{\lhd}\, U$*, then* $T\,[] \,\ddot{\lhd}\, U\,[]$*.*

4 I/O simulation

In this section we introduce a simulation relation between behaviours. We start by adapting to our framework one of the classical notions of subtyping from the session types literature: the *strong subcontract relation* of [14]. A behaviour p is a subtype of p' iff, whenever p' is compliant with some (arbitrary) behaviour q, then p is compliant with q[3]. Thus, p can transparently replace p', in all contexts.

Definition 9 (Subtype). *$\sqsubseteq$ is the largest relation s.t. $p \sqsubseteq q$ implies $\forall r \,.\, q \ddot{\bowtie} r \implies p \ddot{\bowtie} r$. We write $p \sqsubseteq_{\mathbb{R}} q$ to restrict r to the set $\mathbb{R}$ (i.e., $\forall r \in \mathbb{R} \ldots$).*

Despite its elegance and generality, Def. 9 cannot be directly exploited to establish whether two behaviours are related, due to the universal quantification over all contexts. For session types, alternative characterisations of $\sqsubseteq$ have been defined, usually in the form of a syntax-driven coinductive relation [14,1]. This approach amounts to restricting p, q and r in Def. 9 to a process calculus with specific syntax and transition rules — e.g., $p, q, r \in \mathbb{S}_s$. In our semantic framework, however, behaviours are not syntax. We shall extend these characterisations from session behaviours to arbitrary ones, without resorting to a universal quantification over contexts. To do that, we define an *I/O simulation* relation on behaviours, denoted by $\ddot{\leqslant}$. We show that it is a preorder (Theorem 3), and it preserves I/O compliance (Theorem 4). $\ddot{\leqslant}$ is equivalent to the subtype relation on sync session behaviours (Theorem 5), albeit stricter on arbitrary behaviours.

Let $\mathbb{Q}$ be a set of behaviours. We write $q \Rrightarrow \mathbb{Q}$ iff $\emptyset \neq \mathbb{Q} \subseteq \{q' \mid q \Rightarrow q'\}$. By extension, we write $\mathbb{Q} \xRightarrow{\ell_\tau} q''$ iff $\exists q' \in \mathbb{Q} \,.\, q' \xRightarrow{\ell_\tau} q''$, and similarly for $\mathbb{Q} \Rightarrow q''$. We write $\mathbb{Q}\Downarrow$ for $\bigcup_{q' \in \mathbb{Q}} q'\Downarrow$, and similarly for $\mathbb{Q}\Downarrow^{?}$ and $\mathbb{Q}\Downarrow^{!}$.

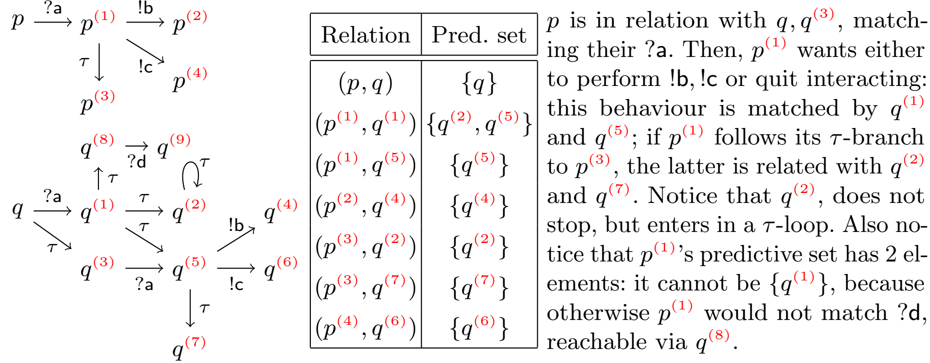
Definition 10 (I/O simulation). *$\ddot{\mathcal{R}}$ is a* I/O simulation relation *iff, whenever $p \,\ddot{\mathcal{R}}\, q$, then $\exists \mathbb{Q}$ (called* predictive set*) such that $q \Rrightarrow \mathbb{Q}$, and:*

a) $p\Downarrow^{!} = \emptyset \implies \mathbb{Q}\Downarrow^{!} = \emptyset$;
b) $\mathbb{Q}\Downarrow^{?} \subseteq p\Downarrow^{?} \;\wedge\; (\mathbb{Q}\Downarrow^{?} = \emptyset \implies p\Downarrow^{?} = \emptyset)$;
c) $p \xrightarrow{\tau} p' \implies \exists q' \,.\, \mathbb{Q} \Rightarrow q' \,\wedge\, p' \,\ddot{\mathcal{R}}\, q'$;
d) $p \xrightarrow{!\mathsf{a}} p' \implies \exists q' \,.\, \mathbb{Q} \xRightarrow{!\mathsf{a}} q' \,\wedge\, p' \,\ddot{\mathcal{R}}\, q'$;
e) $p \xrightarrow{?\mathsf{a}} p' \wedge \mathbb{Q} \xRightarrow{?\mathsf{a}} \implies \exists q' \,.\, \mathbb{Q} \xRightarrow{?\mathsf{a}} q' \,\wedge\, p' \,\ddot{\mathcal{R}}\, q'$.

We write $\ddot{\leqslant}$ for the largest I/O simulation, $\ddot{\approx}$ for the largest symmetric I/O simulation, and $\ddot{=}$ for $\ddot{\leqslant} \cap \ddot{\geqslant}$.

Definition 10 can be explained in terms of a sort of simulation game between players p and q. At the first step, q predicts a suitable choice of its internal moves, via a set $\mathbb{Q}$ of states reachable from q. The outputs of $\mathbb{Q}$ must include those of p (item *d*)), and the inputs of $\mathbb{Q}$ must be included in those of p (item *b*)). Moreover, if p has no outputs, then also $\mathbb{Q}$ cannot have outputs, and if $\mathbb{Q}$ has no inputs, then also p cannot have inputs (items *a*),*b*)). Intuitively, these constraints reflect the usual subtyping in session types: inputs (external choices) can be enlarged (if not empty), while outputs (internal choices) can be narrowed (but not emptied). The

[3] In this paper the direction of $\sqsubseteq$ is opposite w.r.t. the subcontract relation in [14]. Moreover, we require I/O compliance in each context, while [14] only requires progress.



Relation	Pred. set
(p, q)	$\{q\}$
$(p^{(1)}, q^{(1)})$	$\{q^{(2)}, q^{(5)}\}$
$(p^{(1)}, q^{(5)})$	$\{q^{(5)}\}$
$(p^{(2)}, q^{(4)})$	$\{q^{(4)}\}$
$(p^{(3)}, q^{(2)})$	$\{q^{(2)}\}$
$(p^{(3)}, q^{(7)})$	$\{q^{(7)}\}$
$(p^{(4)}, q^{(6)})$	$\{q^{(6)}\}$

p is in relation with $q, q^{(3)}$, matching their ?a. Then, $p^{(1)}$ wants either to perform !b, !c or quit interacting: this behaviour is matched by $q^{(1)}$ and $q^{(5)}$; if $p^{(1)}$ follows its τ-branch to $p^{(3)}$, the latter is related with $q^{(2)}$ and $q^{(7)}$. Notice that $q^{(2)}$, does not stop, but enters in a τ-loop. Also notice that $p^{(1)}$'s predictive set has 2 elements: it cannot be $\{q^{(1)}\}$, because otherwise $p^{(1)}$ would not match ?d, reachable via $q^{(8)}$.

Table 1: Example of I/O simulation.

requirements above must be preserved by the moves of p. τ-moves and outputs of p must be (weakly) simulated by some process in $\mathbb{Q}$ (items c)–d)). The same holds for inputs (item e)), but only moves shared by p and $\mathbb{Q}$ are considered.

Example 6. Detailed examples of $\mathrel{\ddot{\lesseqgtr}}$ are shown in Table 1, and in [5].

Example 7. Consider Figure 3. To assess $p \mathrel{\ddot{\lesseqgtr}} q$, we choose a predictive set $\mathbb{Q}$ that mandates the inputs of p, and includes its outputs (note that p has an additional input ?c′ *not* offered by $\mathbb{Q}$). The same happens with the predictive set $\mathbb{R}$, assessing $q \mathrel{\ddot{\lesseqgtr}} r$ — but $\mathbb{R}$ must be chosen carefully: it *must* include the lower τ-branch of r, matching the branch of q with a τ-loop and no further I/O; however, it *must not* include the upper τ-branch of r, which requires ?d (not matched by q). Note that $\mathbb{R}$ and the small set inside are predictive sets for $p \mathrel{\ddot{\lesseqgtr}} r$.

We now study some properties of $\mathrel{\ddot{\lesseqgtr}}$. Lemma 3 ensures that Def. 10 is well-formed.

Lemma 3. *Let $\ddot{\mathbb{R}}$ be a set of I/O simulations. Then, $\bigcup \ddot{\mathbb{R}}$ is an I/O simulation.*

The following result relates I/O simulation with weak moves. When $p \mathrel{\ddot{\lesseqgtr}} q$, the relation $\mathrel{\ddot{\lesseqgtr}}$ is preserved by forward τ-moves of p and backward τ-moves of q.

Lemma 4. *If $p \mathrel{\ddot{\lesseqgtr}} q$, with $p \Rightarrow p'$ and $q' \Rightarrow q$, then $p' \mathrel{\ddot{\lesseqgtr}} q'$.*

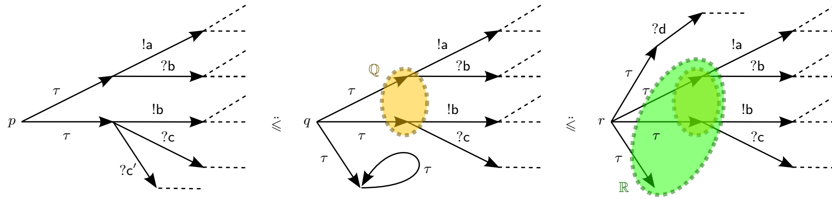


Figure 3: I/O simulation. $\mathbb{Q}$, $\mathbb{R}$ are the predictive sets resp. for $p \mathrel{\ddot{\lesseqgtr}} q$ and $q \mathrel{\ddot{\lesseqgtr}} r$.

Weak simulation ($\precapprox$) and I/O simulations are unrelated, i.e. $\ddot{\leqslant} \not\subseteq \precapprox \not\subseteq \ddot{\leqslant}$. However, weak *bisimulation* ($\approx$) is strictly stronger than I/O bisimulation.

Theorem 2. $\approx \subsetneq \ddot{\approx}$

By Theorem 3, $\ddot{\leqslant}$ is a preorder, as the subtype relation. This is not quite straightforward, due to the existential quantification on the predictive set $\mathbb{Q}$.

Theorem 3. $(\mathcal{U}, \ddot{\leqslant})$ *is a preorder.*

Quite surprisingly, on general behaviours *progress* is *not* preserved by $\ddot{\leqslant}$: if $p \mathrel{\ddot{\leqslant}} q \dashv r$, then it is not always the case that $p \dashv r$. For instance, consider the behaviours in Figure 4. It is easy to check that $p_5 \mathrel{\ddot{\leqslant}} p_6$ and $p_6 \dashv p_7$. However, $p_5 \not\dashv p_7$: indeed, if p_7 chooses the branch !b, then p_5 is stuck waiting for ?c.

Theorem 4 is one of our main results: it states that $\ddot{\leqslant}$ preserves (symmetric/asymmetric) *I/O compliance*. This is a further motivation for using $\ddot{\bowtie}$ instead of $\perp$, when dealing with behaviours where these two notions do not coincide. In the example above, p_7 is not a sync session behaviour: were all behaviours in Figure 4 elements of $\mathbb{S}_s$, we would also have preserved progress (by Theorem 1).

Theorem 4. $p \mathrel{\ddot{\leqslant}} q \circ r \implies p \circ r$, *for* $\circ \in \{\ddot{\rhd}, \ddot{\bowtie}, \ddot{\lhd}\}$.

I/O simulation can be seen as a subtyping relation on general behaviours, that is $p \mathrel{\ddot{\leqslant}} q$ allows p to be always used in place of q. For instance, assume that p is an asynchronous CCS process typed with a session type q, which in turn complies with the session type r. Then, Theorem 4 states that I/O compliance is preserved by $\ddot{\leqslant}$, i.e. p is also I/O compliant with r, notwithstanding with the fact that p and r are specified in different calculi (actually, our statement is even more general, as it applies to *arbitrary* behaviours). Summing up, the process p will interact correctly with any process with type r (Theorem 8).

Theorem 5 below states that I/O simulation is stricter than Definition 9. However, the two notions coincide on synchronous session behaviours. Hence, $\ddot{\leqslant}$ can be interpreted as a subtyping relation in $\mathbb{S}_s$, according to [18].

Theorem 5. *If* $p \mathrel{\ddot{\leqslant}} q$, *then* $p \sqsubseteq q$. *Also, if* $p, q \in \mathbb{S}_s$, *then* $p \sqsubseteq_{\mathbb{S}_s} q \implies p \mathrel{\ddot{\leqslant}} q$.

Theorem 6 generalises Propositions 3 and 4, extending to I/O simulation the set of properties preserved when passing from a sync to an async semantics.

Theorem 6. *If* $T \circ T'$, *then* $T[] \circ T'[]$, *for* $\circ \in \{\ddot{\leqslant}, \vdash, \perp, \dashv, \ddot{\rhd}, \ddot{\bowtie}, \ddot{\lhd}\}$.

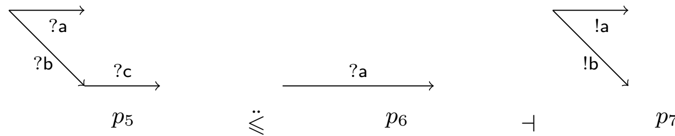


Figure 4: Progress is not preserved by I/O simulation (on general behaviours).

$$
\begin{aligned} P &= P + \mathbf{0} \\ P + Q &= Q + P \\ P + (Q + R) &= (P + Q) + R \\ P &= P + P \end{aligned}
\qquad
\begin{aligned} P &= P \mid \mathbf{0} \\ P \mid Q &= Q \mid P \\ P \mid (Q \mid R) &= (P \mid Q) \mid R \\ \mu_X P &= P[{}^{\mu_X P}/_X] \end{aligned}
\qquad
\begin{gathered} P \ddot{\leqslant} P \\ \mathsf{!a} \mid \mathsf{?b} \ddot{\leqslant} \mathsf{!a}.\mathsf{?b} \end{gathered}
\qquad
\frac{P \ddot{\leqslant} Q}{\mathsf{!a}.\mathsf{?b}.P \ddot{\leqslant} \mathsf{!a} \mid \mathsf{?b}.Q}\;[|^0]
\qquad
\frac{\forall i \in I \,.\, P_i \ddot{\leqslant} Q}{\sum_{i \in I} \tau.P_i \;\ddot{\leqslant}\; Q}\;[\tau]
$$

$$
\frac{\forall i \in I \,.\, P_i \ddot{\leqslant} Q_i}{\sum_{i \in I} \ell_{\tau i}.P_i \;\ddot{\leqslant}\; \sum_{i \in I} \ell_{\tau i}.Q_i}\;[\ell_\tau]
\qquad
\frac{\begin{array}{l} Q \doteqdot \sum_{i \in I} \mathsf{!a_i}.Q_i \\ \forall i \in I \,.\, P_i \ddot{\leqslant} Q_i \qquad I \neq \emptyset \end{array}}{\sum_{i \in I} \mathsf{!a_i}.P_i \;\ddot{\leqslant}\; Q + \mathsf{!b}.Q'}\;[\textsc{Int}]
\qquad
\frac{\begin{array}{ll} P \ddot{\leqslant} Q & \\ R \ddot{\leqslant} S & \mathrm{ins}(P) \cup \mathrm{ins}(Q) = \emptyset \end{array}}{P \mid R \;\ddot{\leqslant}\; Q \mid S}\;[|]
$$

$$
\frac{\begin{array}{lll} Q \doteqdot \sum_{i \in I} \mathsf{?a_i}.Q_i & & \forall k \in K \,.\, P_k \ddot{\leqslant} Q \\ \forall i \in I \,.\, P_i \ddot{\leqslant} Q_i & \emptyset \neq I \subseteq J & \forall j \in J \setminus I \,.\, \mathsf{a}_j \notin \{\mathsf{a}_i\}_{i \in I} \end{array}}{\sum_{j \in J} (\mathsf{?a_j}.P_j) + \sum_{k \in K} \tau.P_k \;\ddot{\leqslant}\; Q}\;[\textsc{Ext}]
$$

Figure 5: Axioms for $\ddot{\leqslant}$ in $\mathcal{P}_a^-$. $\mathrm{ins}(P)$ gives the set of inputs in P's body.

5 Session types without types

Our treatment so far does not depend on a syntactic representation of behaviours in $\mathcal{U}$. In the resulting unifying view, there are no inherent distinctions between processes and types: they are just states of an LTS. This allows us to define relations between objects which morally belong to different realms: e.g. $p \ddot{\leqslant} q$ may relate, say, an async CCS process with a (sync or async) session type.

The price for this generalisation is (seemingly) the loss of a useful feature: using syntax-based reasoning to check whether a process has a certain type, without having to deal with the semantic level. In this section, we show how this possibility can be restored in four steps: *(i)* choosing a process language and a type language (with their corresponding semantics); *(ii)* encoding the former in the latter; *(iii)* devising a sound set of axioms for $\ddot{\leqslant}$; and *(iv)* using these axioms to induce syntax-based typing rules that imply (i.e., safely approximate) $\ddot{\leqslant}$.

In this section we give a proof-of-concept of this methodology for the case of async CCS ($\mathcal{P}_a$) for processes, and async session behaviours ($\mathcal{S}_a$) as types. The encoding from types to processes for step *(ii)* is the one given in Definition 6. Proceeding to step *(iii)*, we now introduce a set of $\ddot{\leqslant}$-based relations for $\mathcal{P}_a$. We shall sometimes omit generic buffers $[\sigma]$ appearing in processes.

Lemma 5 ("Axioms" for $\ddot{\leqslant}$). *The relations in Figure 5 hold.*

The axioms in Lemma 5 are mostly straightforward. [Int] and [Ext] (with $K = \emptyset$) model the typical session typing rules for internal/external choices (resp. with outputs and inputs), allowing to add inputs and remove outputs according to $\ddot{\leqslant}$. [Ext] with $K \neq \emptyset$ handles an external choice that is interrupted (with τ-moves) and later reprised (i.e., a simple case of Erlang-style `receive`... `after`... behaviour, seen in §2). [|] allows the parallel composition of behaviours, provided that they cannot interfere badly (i.e., compete on the same inputs) along their reductions.

To ease the presentation, we focus on a fragment of async CCS (called $\mathcal{P}_a^-$) where *(a)* choices are guarded, *(b)* $|$ cannot appear within recursion, and *(c)*

in $P \mid Q$, either P's or Q's body cannot contain inputs. Conditions *(b)* and *(c)* globally enforce the premises of $[\mid]$, allowing us to simplify Def. 11 below.

Definition 11. *Let Γ be a mapping from recursion variables to pairs of $\mathcal{P}_a^-$ terms. We define $\ddot{\preccurlyeq}_\Gamma$ as the least relation between $\mathcal{P}_a^-$ terms closed under the rules obtained by replacing $\ddot{\leqslant}$ with $\ddot{\preccurlyeq}_\Gamma$ in Figure 5 — plus the following:*

$$\frac{\Gamma(X) = (P, Q)}{X \ddot{\preccurlyeq}_\Gamma Q} \text{ [S-VAR]} \qquad\qquad \frac{P \ddot{\preccurlyeq}_{\Gamma, X:(\mu_X P, Q)} Q}{\mu_X P \ddot{\preccurlyeq}_\Gamma Q} \text{ [S-REC]}$$

We treat the $=$-based relations in Figure 5 as structural congruence rules.

The rules in Def. 11 are straightforward: [S-VAR] enriches the environment by "guessing" that $P \ddot{\leqslant} Q$; [S-REC] consumes such a guess, introducing recursion.

Theorem 7 states that $(\mathcal{P}_a^-, \ddot{\preccurlyeq})$ is a preorder stricter than $(\mathcal{P}_a^-, \ddot{\leqslant})$, and it is preserved by all the operators of $\mathcal{P}_a^-$, that is $.$, $+$, and $\mid$. This enables us to use the syntactic rules $\ddot{\preccurlyeq}_\Gamma$ as a basis for a type system for $\mathcal{P}_a^-$ (as we will see in Def. 12).

Theorem 7. *$\ddot{\preccurlyeq}$ is a precongruence for $\mathcal{P}_a^-$, and $P \ddot{\preccurlyeq} Q \implies P[\sigma] \ddot{\leqslant} Q[\sigma]$.*

A non-obvious aspect of Definition 11 and Theorem 7 is that, by requiring guarded choices in $\mathcal{P}_a^-$, $\ddot{\preccurlyeq}$ is preserved by $+$ (rule $[\ell_\tau]$). This is *not* directly matched by a corresponding property for $\ddot{\leqslant}$ in $\mathcal{P}_a$ without guarded choices, i.e. $P \ddot{\leqslant} Q \implies P + R \ddot{\leqslant} Q + R$. Indeed, the latter implication is false in general, because $\tau.?\mathsf{a}.P \ddot{\leqslant} ?\mathsf{a}.P$, but $?\mathsf{b} + \tau.?\mathsf{a}.P \ddot{\not\leqslant} ?\mathsf{b} + ?\mathsf{a}.P$. A similar argument holds for $\mid$, when arbitrary terms are put in parallel. This shows that $\ddot{\leqslant}$ is *not* a precongruence for $\mathcal{P}_a$, and gives reason for having $\ddot{\preccurlyeq}$ *stricter* than $\ddot{\leqslant}$.

We can now define a *syntax-directed* typing judgement relating $\mathcal{P}_a^-$ processes with session types. To this purpose, we exploit the encoding in Definition 6.

Definition 12. *We write $\Gamma \vdash P : T$ iff $P \ddot{\preccurlyeq}_\Gamma [\![T]\!]$.*

Theorem 8 states the correctness of our "typing" discipline. Suppose you have a process P with type T, and a process Q with type U. If T and U are I/O compliant, then we have that P and Q are I/O compliant, too. Thus, by Theorem 1 we have that the behaviour $P[] \parallel Q[]$ enjoys progress.

Theorem 8. *If $\vdash P : T$, $\vdash Q : U$ with $T[] \circ U[]$, then $P[] \circ Q[]$, for $\circ \in \{\ddot{\triangleright}, \ddot{\bowtie}, \ddot{\triangleleft}\}$.*

Proof. From Def. 12 we have $P \ddot{\preccurlyeq} [\![T]\!]$; by Lemma 1, Def. 11 and Theorem 7 it follows $P[] \ddot{\leqslant} T[]$. Similarly, $Q[] \ddot{\leqslant} U[]$. Since $P \ddot{\leqslant} T[] \circ U[]$, by Theorem 4 it follows $P[] \circ U[]$. Since $Q[] \ddot{\leqslant} U[] \circ P[]$, then by Theorem 4 we conclude $Q[] \circ P[]$.

Note that $\vdash P : T$ and $\vdash Q : U$ can be inferred by a syntax-driven analysis, by the rules in Definition 11. If T and U are interpreted as *synchronous* session types, than we can use syntax-driven techniques (e.g. those in [1]) to deduce $T \circ U$ in the synchronous case; then, by Theorem 6, this result is lifted "for free" to the async case. We stress that the above result is obtained just by exploiting the properties of I/O simulation, without explicitly proving subject reduction.

Example 8. From §2, recall Alice's type T''_{A} and process P''_{A} when she is late for work. We have the following type encoding in $\mathcal{P}^{-}_{a}$: $P_{T''_{\mathsf{A}}} = [\![T''_{\mathsf{A}}]\!] = \mathsf{!aCoffee.!pay.?coffee}$. Then, by Definition 11, we can derive $\vdash P''_{\mathsf{A}} : T''_{\mathsf{A}}$.

Let us now consider the bartender processes and types in §2. Since in Example 5 we determined that $U_{\mathsf{B}} \,\ddot{\bowtie}\, T''_{\mathsf{A}}$, by Theorem 4 we have $Q_{\mathsf{B}} \,\ddot{\bowtie}\, P''_{\mathsf{A}}$. Also, since in Example 9 we show that $\vdash Q''_{\mathsf{B}} : U_{\mathsf{B}}$, by Theorem 8 we have $Q''_{\mathsf{B}} \,\ddot{\bowtie}\, P''_{\mathsf{A}}$.

Example 9. From §2, recall Q''_{B}, U_{B}. We can derive $\vdash Q''_{\mathsf{B}} : U_{\mathsf{B}}$.

The previous examples show that our syntax-driven rules allow to type an Erlang-style `receive`... `after`... behaviour, featured in the bartender process.

6 Conclusions and related work

We have revisited the theory of session types from a purely semantic perspective. We have defined a preorder $\ddot{\leqslant}$ between generic behaviours, which unifies the notions of typing and subtyping for session types, as well as their synchronous and asynchronous interpretations. In this work we mostly focused on behaviours arising from session types and async CCS; however, it seems that our framework can be easily exploited to analyse the properties of other behaviours populating $\mathcal{U}$ — e.g. the LTS semantics of other process calculi and programming languages.

Session types were introduced by Honda *et al.* in [19,28,20], as a type system for communication channels in a variant of the π-calculus. The resulting concept of *structured communication-based programming* has been the cornerstone of the subsequent research. In [23], session types are coupled with a "featherweight" Erlang-like language that, however, omits the problematic `receive`... `after`... construct described in §1. While adapting the type system of [23] to cope with such construct should be feasible, our approach allows the construction of the type system (in our case, the rules for $\ddot{\preccurlyeq}$) to be driven by an explicit underlying semantic notion (the I/O simulation). In particular, we think that the syntax-based reasoning in §5 can be extended to deal with other language constructs, beyond the Erlang-style `receive`... `after`... (which is treated in §5).

Some recent results extend the session types discipline to the multiparty case, starting from [21]. We expect that our approach can be extended to this setting: some insights come from the streamlined approach of [13], where the authors *"take a step back . . . defining global descriptions whose restrictions are semantically justified"*. The plan is to extend the $\ddot{\leqslant}$ relation to capture the *role* of each type/process, and then to produce the syntax-based typing rules via (partial) axiomatisation for a given calculus. We also plan to address the orthogonal problem of multiple *interleaved* sessions. Two starting points are [4,26], which both introduce type systems for ensuring liveness in this setting.

[18] studies subtyping for (dyadic) session types. This topic is reprised in [1,2] with different notions of client-server compliance (e.g., allowing the client to terminate interaction or to skip messages). We took inspiration from these works, aiming at a framework general enough to replicate their notions and results.

Asynchronous dyadic session types have been addressed in [24], where type equivalence up-to buffering was defined over traces, and then approximated via syntax-based rules. A notion of compliance among services with buffers has been studied in [10], which extends [9] (albeit the setting is quite different from session types). Also [25] addresses the problem of defining compliance between service contracts. In their *weak compliance* relation, finite-state orchestrators can resolve external choices or rearrange messages in order to guarantee progress. Weak compliance is unrelated to our I/O compliance: on the one hand, the latter cannot rearrange messages; on the other hand, I/O compliance has no fixed bound on the size of the buffers. For instance, let $!\mathsf{a}^m$ be a sequence of m $!\mathsf{a}$; the async behaviours $!\mathsf{a}.?\mathsf{b}.!\mathsf{a}^2.?\mathsf{b}^2 \cdots !\mathsf{a}^n.?\mathsf{b}^n \cdots$ and $!\mathsf{b}.?\mathsf{a}.!\mathsf{b}^2.?\mathsf{a}^2 \cdots !\mathsf{b}^n.?\mathsf{a}^n \cdots$ are I/O compliant, but they are not weakly compliant, as orchestrators must have a finite rank. In [22] a bisimulation is defined to relate processes communicating via unbounded buffers. The aim of Theorem 6 is to provide for a unifying approach to these issues, by tranferring properties from the sync to the async setting.

Several works denote the successful termination of a behaviour with a specific transition label (e.g. ✓) and/or a specific state (e.g. $\mathbf{1}$ or End). In this paper, we consider two behaviours to be I/O compliant when they synchronise until the client (in the asymmetric case) simply stops interacting. It is easy to extend our framework with a success label/state, thus allowing e.g. to study a testing theory (as in [6]). For simplicity, we chose not to include it in the present work.

Our approach shares some common ground with [14,12]: the inspiration to [16] for the (synchronous) session types semantics, the idea of representing processes and contracts/types in the same LTS, thus allowing for easy reasoning about their progress/compliance properties, and the will to overcome the rigid internal/external choices dichotomy required by session types. In [14], it is assumed that some type system can abstract processes P, Q (expressed in *any* calculus) into contracts. This type system must be *"consistent"* and *"informative"*, by preserving some essential properties like e.g. visible actions and internal non-determinism. A result in [14] is that if the abstractions of P, Q are (strongly) compliant, then P, Q will be (strongly) compliant as well. We believe that the concept of consistent/informative abstraction could be adapted to our framework: it would allow, e.g., to abstract rich process calculi (e.g. with value passing and delegation) into an LTS populated with I/O sorts (like the one adopted in this work). Beyond these general ideas, the technical developments are different: in the strong subcontract relation of [14] there is no input/output distinction, and some desirable subtypings do not hold, e.g. $?\mathsf{a} \,\&\, ?\mathsf{b} \not\sqsubseteq ?\mathsf{a}$. These are restored through a "weak" subcontract relation, exploiting *filters* to suitably resolve external non-determinism. A challenging task would be that of using filters to enforce the I/O co/contra-variance typical of session types (and embodied in $\ddot{\lessgtr}$).

References

1. Barbanera, F., de' Liguoro, U.: Two Notions of Sub-behaviour for Session-based Client/Server Systems. In: PPDP. ACM SIGPLAN, ACM (2010)

2. Barbanera, F., de' Liguoro, U.: Loosening the notions of compliance and sub-behaviour in client/server systems. In: ICE (2014)
3. Bartoletti, M., Cimoli, T., Zunino, R.: A theory of agreements and protection. In: POST (2013)
4. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. In: FORTE (2013)
5. Bartoletti, M., Scalas, A., Zunino, R.: A semantic deconstruction of session types. Tech. rep. (2014), `http://tcs.unica.it/publications`
6. Bernardi, G., Hennessy, M.: Compliance and testing preorders differ. In: SEFM Workshops (2013)
7. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: CONCUR (2008)
8. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR (2010)
9. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Software Composition (2007)
10. Bravetti, M., Zavattaro, G.: Contract compliance and choreography conformance in the presence of message queues. In: WS-FM (2008)
11. Caires, L., Vieira, H.T.: Conversation types. Theor. Comput. Sci. 411(51-52) (2010)
12. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for Web Services. In: WS-FM (2006)
13. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. Logical Methods in Computer Science 8(1) (2012)
14. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for Web services. ACM TOPLAS 31(5) (2009)
15. Castagna, G., Padovani, L.: Contracts for mobile processes. In: CONCUR (2009)
16. De Nicola, R., Hennessy, M.: CCS without tau's. In: TAPSOFT, Vol.1 (1987)
17. Deniélou, P.M., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: ICALP (2013)
18. Gay, S., Hole, M.: Subtyping for session types in the Pi calculus. Acta Inf. 42(2) (2005)
19. Honda, K.: Types for dyadic interaction. In: CONCUR (1993)
20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: ESOP (1998)
21. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
22. Kouzapas, D., Yoshida, N., Honda, K.: On asynchronous session semantics. In: FMOODS/FORTE (2011)
23. Mostrous, D., Vasconcelos, V.T.: Session typing for a featherweight Erlang. In: COORDINATION (2011)
24. Neubauer, M., Thiemann, P.: Session types for asynchronous communication. Universität Freiburg (2004)
25. Padovani, L.: Contract-based discovery of web services modulo simple orchestrators. Theor. Comput. Sci. 411(37) (2010)
26. Padovani, L., Vasconcelos, V.T., Vieira, H.T.: Typing liveness in multiparty communicating systems. In: COORDINATION (2014)
27. Sangiorgi, D.: An introduction to bisimulation and coinduction. Cambridge University Press, Cambridge, UK New York (2012)
28. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: PARLE (1994)