Journal of Logical and Algebraic Methods in Programming 84 (2015) 684–707

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Reversible session-based pi-calculus ☆

Francesco Tiezzi [a,*], Nobuko Yoshida [b]

[a] University of Camerino, Italy
[b] Imperial College London, UK

ARTICLE INFO

Article history:
Received 31 August 2014
Received in revised form 31 March 2015
Accepted 31 March 2015
Available online 15 April 2015

Keywords:
Reversible computing
The pi-calculus
Session types
Session-based programming

ABSTRACT

In this work, we incorporate reversibility into structured communication-based programming, to allow parties of a session to automatically undo, in a rollback fashion, the effect of previously executed interactions. This permits to take different computation paths along the same session, as well as to revert the whole session and start a new one. Our aim is to define a theoretical basis for examining the interplay in concurrent systems between reversible computation and session-based interaction. We thus propose *ReSπ* a session-based variant of π-calculus using memory devices to keep track of the computation history of sessions in order to reverse it. We show how a session type discipline of π-calculus is extended to *ReSπ*, and illustrate its practical advantages for static verification of safe composition in communication-centric distributed software performing reversible computations. We also show how a fully reversible characterisation of the calculus extends to *committable* sessions, where computation can go forward and backward until the session is committed by means of a specific irreversible action.

1. Introduction

In the field of programming languages, *reversible computing* aims at providing a computational model that, besides the standard forward executions, also permits backward execution steps to undo the effect of previously performed forward computations. Despite being a subject of study for many years, reversible computing is recently experiencing a rise in popularity. This is mainly due to the fact that reversibility is a key ingredient in different application domains. In particular, for what specifically concerns our interest, many researchers have put forward exploiting this paradigm in the design of reliable concurrent systems. In fact, it permits us to understand existing patterns for programming reliable systems (e.g., compensations, checkpointing, transactions) and, possibly, to develop new ones.

A promising line of research on this topic advocates reversible variants of well-established process calculi, such as CCS [2] and π-calculus [3], as formalisms for studying reversibility mechanisms in concurrent systems. By pursing this line of research, in this work we incorporate reversibility into a variant of π-calculus equipped with *session* primitives supporting communication-based programming. A (binary) session consists in a series of reciprocal interactions between two parties, possibly with branching and recursion. Interactions on a session are performed via a dedicated private channel, which is generated when initiating the session. Session primitives come together with a session type discipline offering a simple

☆ This work is a revised and extended version of [1], presented in the Proceedings of the 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES). The work has been partially sponsored by EPSRC EP/K011715/1, EP/K034413/1, EP/L00058X/1, the COST Action BETTY (IC1201), the EU projects ASCENS (257414) and FP7-612985 UpScale, and the Italian MIUR PRIN project CINA (2010LHT4KM).
* Corresponding author.
E-mail addresses: francesco.tiezzi@unicam.it (F. Tiezzi), n.yoshida@imperial.ac.uk (N. Yoshida).

http://dx.doi.org/10.1016/j.jlamp.2015.03.004
2352-2208/

checking framework to statically guarantee the correctness of communication patterns. This prevents programs from interacting according to incompatible patterns.

Practically, combining reversibility and sessions paves the way for the development of session-based communication-centric distributed software intrinsically capable of performing reversible computations. In this way, without further coding effort by the application programmer, the interaction among session parties is relaxed so that, e.g., the computation can automatically go back, thus allowing to take different paths when the current one is not satisfactory. As an application example, used in this paper for illustrating our approach, we consider a simple scenario involving a client and multiple providers offering the same service (e.g., on-demand video streaming). The client connects to a provider to request a given service (specifying, e.g., title of a movie, video quality, etc.). The provider replies with a quote determined according to the requested quality of service and to the servers status (current load, available bandwidth, etc.). Then, the client can either accept, negotiate or reject the quote; in the first two cases, the interaction between the two parties shall continue. If a problem occurs during the interaction between the client and the provider for finalising the service agreement, the computation can be automatically reverted. This allows the client to partially undo the current session, in order to take a different computation path along the same session, or even start a new session with (possibly) another provider.

The proposed reversible session-based calculus, called *ReSπ* (*Reversible Session-based* π-calculus), relies on memories to store information about interactions and their effects on the system, which otherwise would be lost during forward computations. This data is used to enable backward computations that revert the effects of the corresponding forward ones. Each memory is devoted to record data concerning a single event, which can correspond to the taking place of a communication action, a choice or a thread forking. Memories are connected with one other, in order to keep track of the computation history, by using unique thread identifiers as links. Like all other formalisms for reversible computing in concurrent settings, forward computations are undone in a *causal-consistent* fashion [4,5]. This means that backtracking does not have to necessarily follow the exact order of forward computations in reverse, because independent actions can be undone in a different order. Thus, an action can be undone only after all the actions causally depending on it have already been undone.

Concerning the session type discipline, *ReSπ* inherits the notion of types and the typing system from π-calculus. Thus, the related results are mainly based on the ones stated for π-calculus. Besides the possibility of taking advantage of the theory already defined for π-calculus, this also allows our investigation to focus on a standard session type setting, rather than on an ad-hoc one specifically introduced for our calculus.

The resulting formalism offers a theoretical basis for examining the interplay between reversible computations and session-based structured interactions. We notice that reversibility enables session parties not only to partially undo the interactions performed along the current session, but also to automatically undo the whole session and restart it, possibly involving different parties. The advantage of the reversible approach is that this behaviour is realised without explicitly implementing loops, but simply relying on the reversibility mechanism available in the language semantics. On the other hand, the session type discipline affects reversibility as it forces concurrent interactions to follow structured communication patterns. If we would consider only a single session, due to linearity, a causal-consistent form of reversibility would not be necessary, i.e. concurrent interactions along the same session are forbidden and, hence, the rollback would follow a single path. Instead, in the general case, concurrent interactions along different sessions may take place, thus introducing causal dependences. In this case, a session execution has to be reverted in a causal-consistent fashion. Notably, interesting issues concerning reversibility and session types are still open questions, especially for what concerns the validity in the reversible setting of standard properties (e.g., progress enforcement) and possibly new properties (e.g., reversibility of ongoing session history, safe closure of subordinate sessions).

It is worth noticing that the proposed calculus is *fully* reversible, i.e. backward computations are always enabled. Full reversibility provides theoretical foundations for studying reversibility in session-based π-calculus, but it is not suitable for a practical use on structured communication-based programming. In fact, reverting a completed session might not be desirable. Therefore, we also propose an extension of the calculus with an *irreversible* action for committing the completion of sessions. In this way, computation would go backward and forward, allowing the parties to try different interactions, until the session is successfully completed and, hence, irreversibly closed.

Summary of the rest of the paper. Section 2 reviews strictly related work. Section 3 recalls syntax and semantics definitions of the considered session-based variants of π-calculus. Section 4 introduces *ReSπ*, our reversible session-based calculus. Section 5 shows the results concerning the reversibility properties of *ReSπ*. Section 6 describes the associated typing discipline. Section 7 presents the extension of *ReSπ* with irreversible commit actions. Section 8 concludes the paper by touching upon directions for future work. Proofs of results are collected in Appendix A.

2. Related work

Our proposal combines the notion of (causal-consistent) reversibility with (typed) primitives supporting session-based interactions in concurrent systems. We review here some of the closely related works concerning either reversibility or session types.

Forms of reversible computation can be found in different formalisms in the literature. For example, backward reductions are considered in the λ-calculus to define equality on expressions [6]. Similar notions are used in the definitions of back and

forth bisimulations [7] on Labelled Transition Systems, and of reversible steps in Petri nets [8]. More in practice, reversibility can be used for exploring different possibilities in a computation. For example, the Prolog language uses its backtracking capabilities to explore the state-space of a derivation to find a solution for a given goal. However, in our paper we mainly focus on the use of reversible computing as a suitable paradigm for designing and developing reliable concurrent systems, which came to prominence in recent years. Along this line of research, the works in the reversible computing field most closely related to ours are those concerning the definition of *reversible process calculi*. We briefly discuss the most relevant of them below, and refer the interested reader to [9] for a comprehensive survey and a larger perspective.

Reversible CCS (RCCS) [5] is the first proposal of reversible calculus, from which all subsequent works drew inspiration. The host calculus (i.e., the non-reversible calculus extended with capabilities for reversibility) is CCS without recursive definitions and relabelling. To each currently running thread is associated an individual memory stack keeping track of past actions, as well as forks and synchronisations. Information pushed on the memory stacks, upon doing a forward transition, can be then used for a rollback. The memories also serve as a naming scheme and yield unique identifiers for threads. When a process divides in two sub-threads, each sub-thread inherits the father memory together with a fork number (either $\langle 1 \rangle$ or $\langle 2 \rangle$) indicating which of the two sons the thread is. Then, in the case of a forward synchronisation, the synchronised threads exchange their names (i.e., memories) in order to allows the corresponding backward synchronisation to take place. A drawback of this approach for memorising fork actions is that the parallel operator does not satisfy usual structural congruence rules as commutativity, associativity and nil process as neutral element. It is proved that RCCS is causally consistent, i.e. the calculus allows backtrack along any causally equivalent past, where concurrent actions can be swapped and successive inverse actions can be cancelled. RCCS has been used for studying a general notion of transactions in [10].

CCS-R [11] is another reversible variant of CCS, which however mainly aims at formalising biological systems. Like the previous calculus, it relies on memory stacks for storing information needed for backtracking, which now also record events corresponding to the unfolding of process definitions. Instead, differently from RCCS, specific identifiers are used to label threads; in case of unfolding, sub-threads are labelled on-the-fly. Forking now does not exploit fork numbers, but it requires the memory stack of a given thread be empty before enabling the execution of its sons; this forces the sub-threads to share the memory that their father had before the forking. As in RCCS, in case of synchronisation, the communicating threads exchange their identifiers. The transition system of CCS-R is proved to be reversible and it is demonstrated that CCS-R is sound and complete w.r.t. CCS.

CCS with communication Keys (CCSK) [12] is a reversible process calculus obtained by applying a general procedure to produce reversible calculi. A relevant aspect of this approach is that it does not rely on memories for supporting backtracking. The idea is to maintain the structure of processes fixed throughout computations, thus avoiding to consume guards and alternative choices, which is the source of irreversibility. Past behaviour and discarded alternatives are then recorded in the syntax of terms. This is realised by transforming the dynamic rules of the SOS semantics into static-like rules. In this way, backward rules are obtained simply as symmetric versions of the forward ones. To ensure that synchronisations are properly reverted, two communicating threads have to agree on a communication key, which will uniquely identify that communication. In this way, the synchronising actions are locked together and can only be undone together. As usual, results showing that the method yields well-behaved transition relations are provided. The proposed converting procedure can be applied to other calculi without name passing, such as ACP [13] or CSP [14], but it is not suitable for calculi with name binders, as π-calculus, which we are interested in this work.

$\rho\pi$ [15] is a reversible variant of the higher-order π-calculus [16]. It borrows from RCCS the use of memories for keeping track of past actions. However, in $\rho\pi$ memories are not stacks syntactically associated to threads, but they simply are terms, each one dedicated to a single communication, in parallel with processes. The connection between memories and threads is kept by resorting to identifiers in a way similar to CCSK. Fork handling relies on specific structured tags connecting the identifier of the father thread with the identifiers of its sub-threads. Besides proving that $\rho\pi$ is causally consistent, it is also shown that it can be faithfully encoded into higher-order π-calculus. Notably, differently from the approaches mentioned before, the semantics of $\rho\pi$ is given in a reduction style. A variant of this calculus, called roll-π [17], has been defined to control backward computations by means of a rollback primitive. The approaches proposed in [15,17] have been applied in [18] for reversing a variant of Klaim [19].

Another reversible variant of π-calculus is Rπ [20]. Differently from $\rho\pi$, Rπ considers a standard π-calculus (without choice and replication) as host calculus, and its semantics is defined in terms of a labelled transition relation. This latter point requires to introduce some technicalities to properly deal with scope extrusion. Similarly to RCCS, this calculus relies on memory stacks, now recording events (i.e., consumed prefixes and related substitutions) and forking (by means of the fork symbol $\langle \uparrow \rangle$). Results about the notion of causality induced by the semantics of the calculus are provided.

Reversible structures [21] is a simple computational calculus, based on DSD [22], for modelling chemical systems. Since such systems are naturally reversible but have no backtracking memory, differently from most of the above proposals, reversible structures does not exploit memories. Instead, reversible structures maintain the structure of terms and use a special symbol $\hat{}$ to indicate the next operations (one forward and one backward) that a term can perform. Terms of the calculus are parallel compositions of signals and gates (i.e., terms that accept input signals and emit output signals), which interact according to a CCS-style model. When a forward synchronisation takes place, the executed gate input is labelled by the identifier of the consumed signal, and the pointer symbol inside the gate is moved forward. The backwards computation is realised by executing these operations in a reverse way, thus releasing the output signal. As usual, the interplay

between causal dependency and reversible structures is studied, with the novelty that in this setting signal identifiers are not necessary unique.

In our work we mainly take inspiration from the $\rho\pi$ approach. Indeed, all other approaches based on CCS and DSD cannot be directly applied to a calculus with name-passing. Moreover, the $\rho\pi$ approach is preferable to the Rπ one because the former proposes a reduction semantics, which we are interested in, while the latter proposes a labelled semantics, which would complicate our theoretical framework (in order to properly deal with scope extension of names). Specifically, we use unique (non-structured) tags for identifying threads and memories for recording taking place of actions, choices and forking. Each memory is devoted to storing the information needed to revert a single event, and memories are connected each other, in order to keep track of computation history, by using tags as links.

For what concerns the related works on session-based calculi, it is worth noticing that we consider a setting as standard and simple as possible, which is the one with synchronous binary sessions. In particular, our host calculus is the well-established variant of π-calculus introduced in [23], whose notation has been revised according to [24]. We leave for future investigation the application of our approach to formalisms relying on the other forms of sessions introduced in the literature, among which we would like to mention asynchronous binary sessions [25], multiparty asynchronous sessions [26], multiparty synchronous sessions [27], sessions with higher-order communication [24], sessions specifically devised for service-oriented computing [28–30].

Finally, the paper with the aim closest to ours is [31], where a formalism combining the notions of reversibility and session is proposed. This calculus is simpler than *ReS*π, because it is an extension of the formalism of *session behaviours* [32] without delegation (i.e., it is a sub-language of CCS) with a checkpoint-based backtracking mechanism. In fact, neither message nor channel passing are considered in the host calculus. Concerning reversibility, only the behaviour prefixed by the lastly traversed checkpoint is recorded by a given party, that is each behaviour is simply paired with a one-size memory. Moreover, causal-consistency is not considered, because in this formalism parties just reduce in sequential way. Also committable sessions are not taken into account. On the other hand, this formalism enabled the study of an extension of the compliance notion to the reversible setting.

3. Session-based π-calculus

In this section we present the syntax and semantics definitions of the host language considered for our reversible calculus. This is a variant of π-calculus enriched with primitives for managing structured binary sessions.

3.1. Syntax

We use the following base sets: *shared channels*, used to initiate sessions; *session channels*, consisting on pairs of *endpoints* used by the two parties to exchange *values* within an established session; *variables*, used to store values; *labels*, used to select and offer branching choices; and *process variables*, used for recursion. The corresponding notation and terminology are as follows:

$$\text{Channels: } c, c', \ldots \begin{cases} \left.\begin{array}{l} \text{Variables: } x, y, \ldots \\ \text{Shared channels: } a, b, \ldots \end{array}\right\} \text{Shared identifiers: } u, u', \ldots \\ \text{Session channels } s, s' \ldots \end{cases}$$

$$\left.\begin{array}{l} \text{Variables: } x, y, \ldots \\ \text{Session endpoints: } s, \bar{s}, \ldots \end{array}\right\} \text{Session identifiers: } k, k', \ldots$$

$$\begin{array}{l} \text{Labels: } l, l', \ldots \\ \text{Process variables: } X, Y, \ldots \end{array}$$

$$\text{Values: } v, v', \ldots \begin{cases} \text{Booleans: } \texttt{true}, \texttt{false} \\ \text{Integers: } 0, 1, \ldots \\ \text{Shared channels: } a, b, \ldots \\ \text{Session endpoints: } s, \bar{s}, \ldots \end{cases}$$

Notably, each session channel s has two (dual) endpoints, denoted by s and $\bar{s}$, each one assigned to one session party to exchange values with the other. We define duality to be idempotent, i.e. $\bar{\bar{s}} = s$. The use of two separated endpoints is similar to that of *polarities* in [33,23]. Notation $\tilde{\cdot}$ stands for tuples, e.g. $\tilde{c}$ means $c_1, \ldots, c_n$.

Processes, ranged over by P, Q, ..., and *expressions*, ranged over by e, e', ... are given by the grammar in Fig. 1. We use $\mathrm{op}(\cdot, \ldots, \cdot)$ to denote a generic expression operator; we assume that expressions are equipped with standard operators on boolean and integer values (e.g., $\wedge$, $+$, ...).

The initiation of a session is triggered by the synchronisation on a shared channel a of two processes of the form $\bar{a}(x).P$ and $a(y).Q$. This causes the generation of a fresh session channel s, whose endpoints replace variables x and y, by means of a substitution application, in order to be used by P and Q, respectively, for later communications. Primitives $k!\langle e\rangle.P$ and $k'?(x).Q$ denote output and input via session endpoints identified by k and k', respectively. These communication primitives

Processes	P	$::=$	$\bar{u}(x).P$	connect
		$\mid$	$u(x).P$	connect dual
		$\mid$	$k!\langle e\rangle.P$	output
		$\mid$	$k?(x).P$	input
		$\mid$	$k \triangleleft l.P$	selection
		$\mid$	$k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$	branching
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional choice
		$\mid$	$P \mid Q$	parallel composition
		$\mid$	$(\nu c)\, P$	restriction
		$\mid$	X	process variable
		$\mid$	$\mu X.P$	recursion
		$\mid$	$\mathbf{0}$	inaction
Expressions	e	$::=$	v	value
		$\mid$	x	variable
		$\mid$	$\mathrm{op}(e_1, \ldots, e_n)$	composed expression

Fig. 1. Session-based π-calculus: syntax.

realise the standard synchronous message passing, where messages result from the evaluation of expressions. Notably, an exchanged value can be an endpoint that is being used in a session (this channel-passing modality is called *delegation*), thus allowing complex nested structured communications. Constructs $k \triangleleft l.P$ and $k' \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ denote label selection and label branching (where $l_1, \ldots, l_n$ are assumed to be pairwise distinct) via endpoints identified by k and k', respectively. They mime method invocation in object-based programming.

The above interaction primitives are then combined by standard process calculi constructs: conditional choice, parallel composition, restriction, recursion and the empty process (denoting inaction). It is worth noticing that restriction can have both shared and session channels as argument: $(\nu a)P$ states that a is a private shared channel of P; similarly, $(\nu s)P$ states that the two endpoints of the session channel, namely s and $\bar{s}$, are invisible from processes different from P (see the seventh law in Fig. 2), i.e. no external process can perform a session action on either of these endpoints (this ensures non-interference within a session). As a matter of notation, we will write $(\nu c_1, \ldots, c_n)P$ in place of $(\nu c_1) \ldots (\nu c_n)P$.

We adopt the following conventions about the operators precedence: prefixing, restriction, and recursion bind more tightly than parallel composition.

Bindings are defined as follows: $\bar{u}(x).P$, $u(x).P$ and $k?(x).P$ bind variable x in P; $(\nu a)\, P$ binds shared channel a in P; $(\nu s)\, P$ binds session channel s in P; finally, $\mu X.P$ binds process variable X in P. The derived notions of bound and free names, alpha-equivalence $\equiv_\alpha$, and substitution are standard. For P a process, $\mathrm{fv}(P)$ denotes the set of *free variables*, $\mathrm{fc}(P)$ denotes the set of *free shared channels*, and $\mathrm{fse}(P)$ the set of *free session endpoints*. For the sake of simplicity, we assume that free and bound variables are always chosen to be different, and that bound variables are pairwise distinct; the same applies to names. Of course, these conditions are not restrictive and can always be fulfilled by possibly using alpha-conversion.

3.2. Semantics

The operational semantics is given in terms of a structural congruence and of a reduction relation. Notably, the semantics is only defined for *closed* terms, i.e. terms without free variables. Indeed, we consider the binding of a variable as its declaration (and initialisation), therefore free occurrences of variables at the outset in a term must be prevented since they are similar to uses of variables before their declaration in programs (which are considered as programming errors).

The *structural congruence*, written $\equiv$, is defined as the smallest congruence relation on processes that includes the equational laws shown in Fig. 2. These are the standard laws of π-calculus. Reading the laws in Fig. 2 by row from left to right, and from top to bottom row, the first three are the monoid laws for $\mid$ (i.e., it is associative and commutative, and has $\mathbf{0}$ as identity element). The second four laws deal with restriction and enable garbage-collection of channels, scope extension and scope swap, respectively. The eighth law permits a recursion to be unfolded (notation $P[Q/X]$ denotes replacement of free occurrences of X in P by process Q). The last law equates alpha-equivalent processes, i.e. processes only differing in the identity of bound variables/channels.

To define the reduction relation, we use an auxiliary function $\cdot\downarrow$ for evaluating closed expressions: $e \downarrow v$ says that expression e evaluates to value v (where $v \downarrow v$, and $x\downarrow$ is undefined).

The *reduction relation*, written $\rightarrow$, is the smallest relation on closed processes generated by the rules in Fig. 3. We comment on salient points. A new session is established when two parallel processes synchronise via a shared channel a; this results in the generation of a fresh (private) session channel whose endpoints are assigned to the two session parties (rule [Con]). During a session, the two parties can exchange values (for data- and channel-passing, rule [Com]) and labels (for branching selection, rule [Lab]). The other rules are standard and state that: conditional choice evolves according to the evaluation of the expression argument (rules [If1] and [If2]); if a part of a larger process evolves, the whole process evolves accordingly (rules [Par] and [Res]); and structural congruent processes have the same reductions (rule [Str]).

$$
\begin{array}{ll}
(P \mid Q) \mid R \equiv P \mid (Q \mid R) & P \mid Q \equiv Q \mid P \\
P \mid \mathbf{0} \equiv P & (\nu c)\mathbf{0} \equiv \mathbf{0} \\
(\nu a)P \mid Q \equiv (\nu a)(P \mid Q) \ \text{ if } a \notin \mathrm{fc}(Q) & (\nu c_1)(\nu c_2)P \equiv (\nu c_2)(\nu c_1)P \\
(\nu s)P \mid Q \equiv (\nu s)(P \mid Q) \ \text{ if } s, \bar{s} \notin \mathrm{fse}(Q) & \mu X.P \equiv P[\mu X.P/X] \\
P \equiv Q \ \text{ if } P \equiv_\alpha Q &
\end{array}
$$

Fig. 2. Session-based π-calculus: structural congruence.

$$
\begin{array}{llll}
\bar{a}(x).P_1 \mid a(y).P_2 \ \rightarrow \ (\nu s)(P_1[\bar{s}/x] \mid P_2[s/y]) & s, \bar{s} \notin \mathrm{fse}(P_1, P_2) & & [\textsc{Con}] \\
\bar{k}!\langle e\rangle.P_1 \mid k?(x).P_2 \ \rightarrow \ P_1 \mid P_2[v/x] & (k = s \ \text{ or } \ k = \bar{s}),\ e \downarrow v & & [\textsc{Com}] \\
\bar{k} \lhd l_i.P \mid k \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \ \rightarrow \ P \mid P_i & (k = s \ \text{ or } \ k = \bar{s}),\ 1 \leq i \leq n & & [\textsc{Lab}] \\
\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \ \rightarrow \ P_1 & e \downarrow \texttt{true} & & [\textsc{If1}] \\
\texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \ \rightarrow \ P_2 & e \downarrow \texttt{false} & & [\textsc{If2}]
\end{array}
$$

$$
\frac{P \ \rightarrow \ P'}{P \mid Q \ \rightarrow \ P' \mid Q}\ [\textsc{Par}] \qquad \frac{P \ \rightarrow \ P'}{(\nu c)P \ \rightarrow \ (\nu c)P'}\ [\textsc{Res}] \qquad \frac{P \equiv P' \ \rightarrow \ Q' \equiv Q}{P \ \rightarrow \ Q}\ [\textsc{Str}]
$$

Fig. 3. Session-based π-calculus: reduction relation.

3.3. The multiple providers scenario in the session-based π-calculus

The scenario involving a client and multiple providers introduced in Section 1 can be rendered in π-calculus as follows (for the sake of simplicity, here we consider just two providers):

$$P_{client} \mid P_{provider1} \mid P_{provider2}$$

where the client process P_{client} is defined as

$$
\begin{array}{l}
\overline{a_{login}}(x).\, x!\langle \mathsf{srv\_req}\rangle.\, x?(y_{quote}). \\
\quad \texttt{if}\ accept(y_{quote})\ \texttt{then}\ x \lhd l_{acc}.\, P_{acc} \\
\quad \texttt{else}\ (\texttt{if}\ negotiate(y_{quote})\ \texttt{then}\ x \lhd l_{neg}.\, P_{neg}\ \texttt{else}\ x \lhd l_{rej}.\, \mathbf{0})
\end{array}
$$

while a provider process $P_{provider\,i}$ is as follows

$$a_{login}(y).\, y?(z_{req}).\, y!\langle quote_i(z_{req})\rangle.\, y \rhd \{l_{acc} : Q_{acc}\ ,\ l_{neg} : Q_{neg}\ ,\ l_{rej} : \mathbf{0}\}$$

We show below a possible evolution of the system, where the client contacts *provider*1 and accepts the proposed quote:

$$
\begin{array}{l}
P_{client} \mid P_{provider1} \mid P_{provider2} \\
\rightarrow \\
(\nu s)(\bar{s}!\langle \mathsf{srv\_req}\rangle.\, \bar{s}?(y_{quote}).\, \texttt{if}\ accept(y_{quote})\ \texttt{then}\ \bar{s} \lhd l_{acc}.\, P_{acc}[\bar{s}/x]\texttt{else}\ (\ldots) \\
\qquad \mid s?(z_{req}).\, s!\langle quote_i(z_{req})\rangle.\, s \rhd \{l_{acc} : Q_{acc}[s/y]\ ,\ l_{neg} : Q_{neg}[s/y]\ ,\ l_{rej} : \mathbf{0}\}\,) \\
\mid P_{provider2} \\
\rightarrow \\
(\nu s)(\bar{s}?(y_{quote}).\, \texttt{if}\ accept(y_{quote})\ \texttt{then}\ \bar{s} \lhd l_{acc}.\, P_{acc}[\bar{s}/x]\texttt{else}\ (\ldots) \\
\qquad \mid s!\langle quote_i(\mathsf{srv\_req})\rangle.\, s \rhd \{l_{acc} : Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}]\ ,\ \ldots\}\,) \\
\mid P_{provider2} \\
\rightarrow \\
(\nu s)(\,\texttt{if}\ accept(\mathsf{quote})\ \texttt{then}\ \bar{s} \lhd l_{acc}.\, P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}]\texttt{else}\ (\ldots) \\
\qquad \mid s \rhd \{l_{acc} : Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}]\ ,\ \ldots\}\,) \\
\mid P_{provider2} \\
\rightarrow
\end{array}
$$

$$
\begin{array}{l}
(\nu s)(\,\bar{s} \lhd l_{acc}.\, P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \mid s \rhd \{l_{acc} : Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}]\,,\ \ldots\}\,) \\
\mid\ P_{provider2} \\
\rightarrow \\
(\nu s)(\,P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \mid Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}]\,) \\
\mid\ P_{provider2}
\end{array}
$$

4. Reversible session-based π-calculus

In this section, we introduce *ReSπ*, a reversible extension of the calculus described in Section 3.

A reversible calculus is typically obtained from the corresponding host calculus by adding memory devices (see Section 2). They aim at storing information about interactions and their effects on the system, which otherwise would be lost during forward computations, as e.g. the discarded branch in a conditional choice. In doing this, we follow the approach of [15], which in its turn is inspired by [5] (for the use of memories) and by [12] (for the use of thread identifiers). Of course, since here we consider as host calculus a session-based variant of standard π-calculus, rather than an asynchronous higher-order variant, the technical development is different.

Roughly, our approach to keep track of computation history is as follows: we tag processes with unique identifiers and use memories to store the information needed to reverse each single forward reduction. Thus, the history of a reduction sequence is stored in a number of small memories connected each other by using tags as links. In this way, *ReSπ* terms can perform, besides *forward reductions* (denoted by $\twoheadrightarrow$), also *backward reductions* (denoted by $\rightsquigarrow$) that undo the effect of the former ones. As in the reversible calculi discussed in Section 2, forward computations are reverted in a *causal-consistent* fashion. That is, independent (more precisely, *concurrent*) actions can be undone in an order possibly different from the exact order of forward reductions in reverse. Specifically, an action can be undone only after all the actions causally depending on it have already been undone. We will come back on causal-consistency in Section 5.

Before introducing the technicalities of *ReSπ*, we informally provide a basic intuition about its main features. Let us come back to the scenario introduced in Section 1 and specified in π-calculus in Section 3.3. We can obtain a *ReSπ* specification of the scenario by simply annotating the π-calculus term with the (unique) tags t_1, t_2 and t_3 as follows:

$$t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2}$$

Now, the computation described in Section 3.3 corresponds to a sequence of five forward reductions leading to the *ReSπ* process M having the following form:

$$
\begin{array}{l}
(\nu s, \tilde{t})\ (\,t'_1 : P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \mid t'_2 : Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}] \\
\qquad \mid \langle t_1 \mathrel{-}a_{login}(x)(y)(\nu s)P'_{client}P'_{provider1}\!\rightarrow t_2, t''_1, t''_2\rangle \\
\qquad \mid \langle\ldots\rangle \mid \langle\ldots\rangle \mid \langle\ldots\rangle \mid \langle\ldots\rangle\,) \\
\mid\ P_{provider2}
\end{array}
$$

The forward computation has created a tuple $\tilde{t}$ of fresh tags, which includes the tags t'_1 and t'_2 attached to the resulting processes of *client* and *provider*1, respectively. Moreover, each reduction has created a memory $\langle\ldots\rangle$, which is spawn in parallel with the two processes of the involved parties and devoted to store the information for reverting the corresponding forward reduction. Here, for the sake of presentation, we have omitted the content of such memories, except for the one generated by the first reduction: it records that the process tagged by t_1 (i.e., the client) initiates a session s along channel a_{login} with the process tagged by t_2 (i.e., the first provider); it also records the variables replaced by the session endpoints and the continuation processes together with their tags. Notably, process M cannot immediately use this memory to revert the interaction corresponding to the session initiation. Indeed, a memory can trigger a backward reduction only if two processes properly tagged with the continuation tags are available, which is not the case of the first memory in the process M. Therefore, as expected, all the other forward reductions must be previously reverted in order to revert the session initiation one.

4.1. Syntax

The syntax of *ReSπ* is given in Fig. 4. In addition to the base sets used for π-calculus processes in Section 3, here we use *tags*, ranged over by t, t', …, to uniquely identify *threads*. Letters h, h', … denote *names*, i.e. (shared and session) channels and tags together. Uniqueness of tags is ensured by using the restriction operator and by only considering *reachable* processes (see Definition 3).

ReSπ processes are built upon standard (session-based) π-calculus processes by labelling them with tags. Thus, the syntax of π-calculus processes P, as well as of expressions e, is the same of that shown in Fig. 1 and, hence, it is omitted here. It is worth noticing that only *ReSπ* processes can execute (i.e., π-calculus ones cannot).

ReSπ **Processes**	M	$::=$	$t : P$	thread
		$\mid$	$(\nu h)\, M$	channel/tag restriction
		$\mid$	$M \mid N$	parallel composition
		$\mid$	m	memory
		$\mid$	nil	empty process
Memories	m	$::=$	$\langle t_1 \xrightarrow{\mathsf{A}} t_2, t'_1, t'_2 \rangle$	action memory
		$\mid$	$\langle t, e?\, P{:}Q\,, t' \rangle$	choice memory
		$\mid$	$\langle t \rightrightarrows (t_1, t_2) \rangle$	fork memory
	A	$::=$	$a(x)(y)(\nu s)\, P\, Q \quad \mid \quad k\langle e \rangle (x)\, P\, Q$	action events
		$\mid$	$k \triangleleft l_i\, P\{l_1 : P_1, \ldots, l_n : P_n\}$	

Fig. 4. *ReSπ* syntax (π-calculus Processes P and Expressions e are in Fig. 1).

ReSπ also extends π-calculus with *memory* processes m. In particular, there are three kind of memories:

- *Action memory* $\langle t_1 \xrightarrow{\mathsf{A}} t_2, t'_1, t'_2 \rangle$, storing an action event A together with the tag t_1 of the active party of the action, the tag t_2 of the corresponding passive party, and the tags t'_1 and t'_2 of the two new threads activated by the corresponding reduction. An *action event* A, as we shall clarify later, records all information necessary to revert the corresponding interaction, which can be either a session initiation $a(x)(y)(\nu s)\, P\, Q$, a communication along an established session $k\langle e \rangle (x)\, P\, Q$, or a branch selection $k \triangleleft l_i\, P\{l_1 : P_1, \ldots, l_n : P_n\}$. Notably, in the latter two events, k can only be either s or $\bar{s}$ (i.e., it cannot be a variable).
- *Choice memory* $\langle t, e?\, P{:}Q\,, t' \rangle$, storing a choice event $e?\, P{:}Q$ together with the tag t of the conditional choice and the tag t' of the new activated thread. The choice event $e?\, P{:}Q$ records the evaluated expression e, and the processes P and Q of the then-branch and else-branch, respectively.
- *Fork memory* $\langle t \rightrightarrows (t_1, t_2) \rangle$, storing the tag t of a splitting thread, i.e. a thread of the form $t : (P \mid Q)$, together with the tags t_1 and t_2 of the two new activated threads, i.e. $t_1 : P$ and $t_2 : Q$. The use of fork memories is analogous to that of *connectors* in [18].

Threads and memories are composed by parallel composition and restriction operators. The latter, as well as the notion of bound and free identifiers, extend to names. In particular, for M a *ReSπ* process, $\mathrm{ft}(M)$ denotes the set of *free tags*; $\mathrm{fv}(\cdot)$, $\mathrm{fc}(\cdot)$ and $\mathrm{fse}(\cdot)$ extend naturally to *ReSπ* processes. Of course, we still rely on the same assumptions on free and bound variables/channels mentioned in Section 3.1.

Not all processes allowed by the syntax in Fig. 4 are semantically meaningful. Indeed, in a general term of the calculus, the history stored in the memories may not be consistent, due to the use of non-unique tags or broken connections between continuation tags within memories and corresponding threads. For example, given the choice memory $\langle t, e?\, P{:}Q\,, t' \rangle$, we have a broken connection when no thread tagged by t' exists in the *ReSπ* process and no memory of the form $\langle t' \xrightarrow{\mathsf{A}} t_2, t'_1, t'_2 \rangle$, $\langle t_1 \xrightarrow{\mathsf{A}} t', t'_1, t'_2 \rangle$, $\langle t', e?\, P_1{:}P_2, t_1 \rangle$, and $\langle t' \rightrightarrows (t_1, t_2) \rangle$ exist.

The class of meaningful *ReSπ* processes we are interested in consists of *programs* and *runtime processes*. The former are the terms that can be written by programmers, i.e. they are *ReSπ* processes with no memory. In fact, memories are not expected to occur in the source code written by programmers. We assume that the threads within a program have unique tags. The latter terms of the class are the *ReSπ* processes that can be obtained by means of forward reductions from programs; in this way, history consistency is ensured. Using the terminology from [20], the processes of the considered class are called *reachable*. We formalise their definition below.

Definition 1 *(Programs).* The set of *ReSπ programs* is the set of terms generated by the following grammar

$$M \ ::= \ t : P \ \mid \ (\nu c)\, M \ \mid \ M \mid N \ \mid \ \mathrm{nil}$$

and whose threads have distinct tags, where P is a π-calculus process as in Fig. 1.

Definition 2 *(Runtime processes).* The set of *ReSπ runtime processes* is the set of terms obtained by the transitive closure under $\twoheadrightarrow$ (see Section 4.2) of the set of *ReSπ* programs.

Definition 3 *(Reachable processes).* The set of *ReSπ reachable processes* is the union of the sets of programs and runtime processes.

Notice that in Definition 1 the restriction operator is defined on channels c rather than on names h. This because there is no need to restrict tags in a program. In fact, it is sufficient to use distinct tags, as required by the definition. In practice,

$$(M \mid N) \mid L \equiv M \mid (N \mid L) \qquad M \mid N \equiv N \mid M$$

$$M \mid \mathrm{nil} \equiv M \qquad (\nu h)\,\mathrm{nil} \equiv \mathrm{nil}$$

$$(\nu t)M \mid N \equiv (\nu t)(M \mid N) \quad \text{if } t \notin \mathrm{ft}(N) \qquad (\nu h_1)(\nu h_2)M \equiv (\nu h_2)(\nu h_1)M$$

$$(\nu a)M \mid N \equiv (\nu a)(M \mid N) \quad \text{if } a \notin \mathrm{fc}(N) \qquad t : (\nu c)P \equiv (\nu c)\,t : P$$

$$t : P \equiv t : Q \quad \text{if } P \equiv Q \qquad (\nu s)M \mid N \equiv (\nu s)(M \mid N) \quad \text{if } s, \bar{s} \notin \mathrm{fse}(N)$$

$$M \equiv N \quad \text{if } M \equiv_\alpha N \qquad t : (P \mid Q) \equiv (\nu t_1, t_2)(t_1 : P \mid t_2 : Q \mid \langle t \rightrightarrows (t_1, t_2)\rangle)$$

Fig. 5. *ReSπ* structural congruence (additional laws).

the programmer would have to write just a π-calculus term that can be then automatically annotated with unique tags to obtain a *ReSπ* program. At runtime, as shown in the next subsection, it is the operational semantics in charge of generating fresh tags for the new threads by means of the restriction operator.

4.2. Semantics

The operational semantics of *ReSπ* is given in terms of a structural congruence and of a reduction relation.

The *structural congruence* $\equiv$ extends that of π-calculus (Fig. 2) with the additional laws in Fig. 5. Most of the new laws simply deal with the parallel and restriction operators on *ReSπ* processes. Thus, we only focus on relevant laws (below, the laws are read by row from left to right, and from top to bottom row). The eighth law permits a restriction on a π-calculus process to be moved to the level of *ReSπ* processes. The ninth law lifts the congruence at π-calculus process level to the threads level. The last law is crucial for fork handling: it is used to split a single thread composed of two parallel processes into two threads with fresh tags; the tag of the original thread and the new tags are properly recorded in a fork memory.

The *reduction relation* of *ReSπ*, written $\rightarrowtail$, is given as the union of the forward and backward reduction relations defined by the rules in Fig. 6: $\rightarrowtail \;=\; \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are the smallest relations on closed *ReSπ* reachable processes generated by the corresponding rules in the figure.

We comment on salient points. When two parallel threads synchronise to establish a new session (rule [FWCON]), two fresh tags are created to uniquely identify the two continuations of the synchronising threads. Moreover, all relevant information is stored in the action memory $\langle t_1 \mathrel{-}a(x)(y)(\nu s)P_1P_2\mathrel{\rightarrow} t_2, t_1', t_2'\rangle$: the tag t_1 of the initiator (i.e., the thread executing a prefix of the form $\bar{a}(\cdot)$), the tag t_2 of the thread executing the dual action, the tags t_1' and t_2' of their continuations, the shared channel a used for the synchronisation, the replaced variables x and y, the generated session channel s, and the processes P_1 and P_2 to which substitutions are applied. All such information is exploited to revert this reduction (rule [BWCON]). In particular, the corresponding backward reduction is triggered by the coexistence of the memory described above with two threads tagged t_1' and t_2', all of them within the scope of the session channel s and tags t_1' and t_2' generated by the forward reduction (which, in fact, are removed by the backward one). Notice that, when considering reachable processes, due to tag uniqueness, the two processes P and Q must coincide with $P_1[\bar{s}/x]$ and $P_2[s/y]$; indeed, as registered in the memory, these latter processes have been tagged with t_1' and t_2' by the forward reduction. The fact that two threads tagged with t_1' and t_2' are available in parallel to the memory ensures that all actions possibly executed by the two continuations activated by the forward computation have been undone and, hence, we can safely proceed to undone the forward computation itself.

Rules [FWCOM], [BWCOM], [FWLAB], [BWLAB], [FWIF1], [FWIF2] and [BWIF] are similar. Notably, in the first two rules mentioned above, besides tags and continuation processes, the action memory stores the session endpoint k of the receiving party (the other endpoint $\bar{k}$ is obtained by duality), the expression e generating the sent value, and the replaced variable x. It is also worth noticing that, since all information about a choice event is stored in the corresponding memory, we need just one backward rule ([BWIF]) to revert the effect of the forward rules [FWIF1] and [FWIF2]. The meaning of the remaining rules, dealing with parallel composition, restriction and structural congruent terms, is straightforward.

4.3. The multiple providers scenario in ReSπ

Let us consider again the multiple providers scenario. We have shown at the beginning of this section that a *ReSπ* specification can be obtained by simply annotating the π-calculus term with unique tags as follows:

$$t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2}$$

$$t_1 : \bar{a}(x).P_1 \;\mid\; t_2 : a(y).P_2 \qquad s, \bar{s} \notin \mathrm{fse}(P_1, P_2) \qquad [\textsc{fwCon}]$$
$$\twoheadrightarrow \quad (\nu s, t_1', t_2')(t_1' : P_1[\bar{s}/x] \mid t_2' : P_2[s/y] \mid \langle t_1 \text{–}a(x)(y)(\nu s)P_1P_2 \rightarrow t_2, t_1', t_2'\rangle)$$

$$(\nu s, t_1', t_2')(t_1' : P \mid t_2' : Q \mid \langle t_1 \text{–}a(x)(y)(\nu s)P_1P_2 \rightarrow t_2, t_1', t_2'\rangle) \qquad [\textsc{bwCon}]$$
$$\rightsquigarrow \quad t_1 : \bar{a}(x).P_1 \;\mid\; t_2 : a(y).P_2$$

$$t_1 : \bar{k}!\langle e\rangle.P_1 \;\mid\; t_2 : k?(x).P_2 \qquad (k = s \text{ or } k = \bar{s}) \qquad [\textsc{fwCom}]$$
$$\twoheadrightarrow \quad (\nu t_1', t_2')(t_1' : P_1 \mid t_2' : P_2[v/x] \mid \langle t_1 \text{–}k\langle e\rangle(x)P_1P_2 \rightarrow t_2, t_1', t_2'\rangle) \qquad e \downarrow v$$

$$(\nu t_1', t_2')(t_1' : P \mid t_2' : Q \mid \langle t_1 \text{–}k\langle e\rangle(x)P_1P_2 \rightarrow t_2, t_1', t_2'\rangle) \qquad [\textsc{bwCom}]$$
$$\rightsquigarrow \quad t_1 : \bar{k}!\langle e\rangle.P_1 \;\mid\; t_2 : k?(x).P_2$$

$$t_1 : \bar{k} \triangleleft l_i.P \;\mid\; t_2 : k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \qquad (k = s \text{ or } k = \bar{s}) \qquad [\textsc{fwLab}]$$
$$\twoheadrightarrow \quad (\nu t_1', t_2')(t_1' : P \mid t_2' : P_i \mid \langle t_1 \text{–}k \triangleleft l_i\, P\{l_1 : P_1, \ldots, l_n : P_n\} \rightarrow t_2, t_1', t_2'\rangle) \qquad 1 \le i \le n$$

$$(\nu t_1', t_2')(t_1' : Q \mid t_2' : Q' \mid \langle t_1 \text{–}k \triangleleft l_i\, P\{l_1 : P_1, \ldots, l_n : P_n\} \rightarrow t_2, t_1', t_2'\rangle) \qquad [\textsc{bwLab}]$$
$$\rightsquigarrow \quad t_1 : \bar{k} \triangleleft l_i.P \;\mid\; t_2 : k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$$

$$t : \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \;\twoheadrightarrow\; (\nu t')(t' : P_1 \mid \langle t, e?\, P_1{:}P_2, t'\rangle) \qquad e \downarrow \texttt{true} \qquad [\textsc{fwIf1}]$$

$$t : \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \;\twoheadrightarrow\; (\nu t')(t' : P_2 \mid \langle t, e?\, P_1{:}P_2, t'\rangle) \qquad e \downarrow \texttt{false} \qquad [\textsc{fwIf2}]$$

$$(\nu t')(t' : P \mid \langle t, e?\, P_1{:}P_2, t'\rangle) \;\rightsquigarrow\; t : \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \qquad [\textsc{bwIf}]$$

$$\frac{M \twoheadrightarrow M'}{M \mid N \twoheadrightarrow M' \mid N}\ [\textsc{fwPar}] \qquad \frac{M \rightsquigarrow M'}{M \mid N \rightsquigarrow M' \mid N}\ [\textsc{bwPar}]$$

$$\frac{M \twoheadrightarrow M'}{(\nu h)M \twoheadrightarrow (\nu h)M'}\ [\textsc{fwRes}] \qquad \frac{M \rightsquigarrow M'}{(\nu h)M \rightsquigarrow (\nu h)M'}\ [\textsc{bwRes}]$$

$$\frac{M \equiv M' \twoheadrightarrow N' \equiv N}{M \twoheadrightarrow N}\ [\textsc{fwStr}] \qquad \frac{M \equiv M' \rightsquigarrow N' \equiv N}{M \rightsquigarrow N}\ [\textsc{bwStr}]$$

Fig. 6. *ReSπ* reduction relation.

Now, the computation described in Section 3.3 corresponds to the following sequence of forward reductions:

$$\begin{aligned}
& t_1 : P_{client} \;\mid\; t_2 : P_{provider1} \;\mid\; t_3 : P_{provider2} \\
& \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \\
& M \;=\; (\nu s, t_1^1, t_2^1, t_1^2, t_2^2, t_1^3, t_2^3, t_1^4, t_1^5, t_2^4) \\
& \qquad (t_1^5 : P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \;\mid\; t_2^4 : Q_{acc}[s/y][\mathsf{srv_req}/z_{req}] \\
& \qquad \mid\; \langle t_1 \text{–}a_{login}(x)(y)(\nu s)P'_{client}P'_{provider1} \rightarrow t_2, t_1^1, t_2^1\rangle \\
& \qquad \mid\; \langle t_1^1 \text{–}s\langle \mathsf{srv_req}\rangle(z_{req})P''_{client}P''_{provider1} \rightarrow t_2^1, t_1^2, t_2^2\rangle \\
& \qquad \mid\; \langle t_2^2 \text{–}\bar{s}\langle \mathsf{quote}\rangle(y_{quote})P'''_{provider1}P'''_{client} \rightarrow t_1^2, t_2^3, t_1^3\rangle \\
& \qquad \mid\; \langle t_1^3, accept(\mathsf{quote})?\, P_{client_t}{:}P_{client_e}, t_1^4\rangle \\
& \qquad \mid\; \langle t_1^4 \text{–}s \triangleleft l_{acc}\, P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \\
& \qquad\qquad \{l_{acc} : Q_{acc}[s/y][\mathsf{srv_req}/z_{req}], \ldots\} \rightarrow t_2^3, t_1^5, t_2^4\rangle\,) \\
& \qquad \mid\; P_{provider2}
\end{aligned}$$

Basically, five memories have been generated, each one dedicated to revert the effects of the corresponding forward reduction.

If a problem occurs during the subsequent interactions between the client and the provider for finalising the service agreement, the computation can be reverted to the initial state. In particular, the backward rules [bwCon], [bwCom], [bwLab] and [bwIf] can be applied only if the *ReSπ* term contains a memory in parallel with thread(s) appropriately tagged by the

continuation tag(s) stored in the memory. For example, to apply the rule [BWCON] in the process M, two threads tagged by t_1^1 and t_2^1 must be in parallel with the first memory, which actually is not the case. In fact, in M, only the last memory can trigger a backward step, by means of the application of rule [BWLAB]:

$$\begin{aligned} M \rightsquigarrow M' = (\nu s, t_1^1, t_2^1, t_1^2, t_2^2, t_1^3, t_2^3, t_1^4) \\ & (t_1^4 : \bar{s} \triangleleft l_{acc}.\, P_{acc}[\bar{s}/x][\mathsf{quote}/y_{quote}] \\ & \mid t_2^3 : s \triangleright \{l_{acc} : Q_{acc}[s/y][\mathsf{srv\_req}/z_{req}]\,,\ \ldots\} \\ & \mid \langle t_1 \xrightarrow{a_{login}(x)(y)(\nu s)} P'_{client}\, P'_{provider1} \rightarrow t_2, t_1^1, t_2^1\rangle \\ & \mid \langle t_1^1 \xrightarrow{s\langle \mathsf{srv\_req}\rangle(z_{req})} P''_{client}\, P''_{provider1} \rightarrow t_2^1, t_1^2, t_2^2\rangle \\ & \mid \langle t_2^2 \xrightarrow{\bar{s}\langle \mathsf{quote}\rangle(y_{quote})} P'''_{provider1}\, P'''_{client} \rightarrow t_1^2, t_2^3, t_1^3\rangle \\ & \mid \langle t_1^3, accept(\mathsf{quote})?\, P_{client\_t} : P_{client\_e}, t_1^4\rangle\,) \\ & \mid P_{provider2} \end{aligned}$$

In this way, the threads labelled by t_1^5 and t_2^4 are removed, while the threads performing the selection and offering the branching choice, labelled by t_1^4 and t_2^3 respectively, are restored.

Then, in the process M', only the last memory can trigger a backward reduction, which undoes the conditional choice performed by the client thread. Similarly, other backward reductions can be subsequently triggered by the other memories, consuming them from the bottom to the top of the term. In this way, the forward computation can be completely reverted:

$$M' \rightsquigarrow \rightsquigarrow \rightsquigarrow \rightsquigarrow\ t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2}$$

Now, the client can start a new session, possibly with *provider2*.

Notice that in *ReSπ* there is no need of explicitly implementing loops for enabling the client to undo and restart sessions. Notice also that here we do not consider specific primitives and techniques that avoid interacting again with the same provider. This would break the Loop Lemma (see Lemma 5) and complicate our theoretical framework; we refer to [34] for a definition of some of such controlled forms of reversibility.

5. Properties of the reversible calculus

We present in this section some properties of *ReSπ*, which are typically enjoyed by reversible calculi. We exploit terminology, arguments and proof techniques of previous works on reversible calculi (in particular, [15,5,20]). As a matter of notation, we will use $\mathcal{P}$ and $\mathcal{R}$ to denote the set of π-calculus processes and of *ReSπ* processes, respectively.

5.1. Correspondence with π-calculus

We first show that *ReSπ* is a conservative extension of the (session-based) π-calculus. In fact, as most reversible calculi, *ReSπ* is only a decoration of its host calculus. Such decoration can be erased by means of the *forgetful map* ϕ, which maps *ReSπ* terms into π-calculus ones by simply removing memories, tag annotations and tag restrictions.

Definition 4 *(Forgetful map).* The forgetful map $\phi : \mathcal{R} \rightarrow \mathcal{P}$, mapping a *ReSπ* process M into a π-calculus process P, is inductively defined on the structure of M as follows:

$$\begin{array}{rcl} \phi(t : P) = P & \qquad & \phi((\nu c)N) = (\nu c)\phi(N) \\ \phi((\nu t)N) = \phi(N) & & \phi(N_1 \mid N_2) = \phi(N_1) \mid \phi(N_2) \\ \phi(m) = \mathbf{0} & & \phi(\mathrm{nil}) = \mathbf{0} \end{array}$$

To prove the correspondence between *ReSπ* and π-calculus, we need the following auxiliary lemma relating structural congruence of *ReSπ* to that of π-calculus.

Lemma 1. *Let M and N be two ReSπ processes. If $M \equiv N$ then $\phi(M) \equiv \phi(N)$.*

Proof. We proceed by induction on the derivation of $M \equiv N$ (see Appendix A.1.1). □

Now, we can show that each forward reduction of a *ReSπ* process corresponds to a reduction of the corresponding π-calculus process.

Lemma 2. *Let M and N be two ReSπ processes. If $M \twoheadrightarrow N$ then $\phi(M) \rightarrow \phi(N)$.*

Proof. We proceed by induction on the derivation of $M \twoheadrightarrow N$ (see Appendix A.1.1). □

The correspondence between *ReSπ* and π-calculus reductions is completed by the following lemmas, which intuitively are the inverse of Lemma 1 and Lemma 2.

Lemma 3. *Let P and Q be two π-calculus processes. If $P \equiv Q$ then for any ReSπ process M such that $\phi(M) = P$ there exists a ReSπ process N such that $\phi(N) = Q$ and $M \equiv N$.*

Proof. The proof is straightforward (see Appendix A.1.1). □

Lemma 4. *Let P and Q be two π-calculus processes. If $P \to Q$ then for any ReSπ process M such that $\phi(M) = P$ there exists a ReSπ process N such that $\phi(N) = Q$ and $M \twoheadrightarrow N$.*

Proof. We proceed by induction on the derivation of $P \to Q$ (see Appendix A.1.1). □

5.2. Loop lemma

The following lemma shows that, in *ReSπ*, backward reductions are the inverse of the forward ones and vice versa.

Lemma 5 *(Loop lemma). Let M and N be two reachable ReSπ processes. $M \twoheadrightarrow N$ if and only if $N \rightsquigarrow M$.*

Proof. The proof for the *if* part is by induction on the derivation of $M \twoheadrightarrow N$, while the proof for the *only if* part is by induction on the derivation of $N \rightsquigarrow M$ (see Appendix A.1.2). □

5.3. Causal consistency

We show here that reversibility in *ReSπ* is causally consistent. Informally, this means that an action can be reverted only after all the actions causally depending on it have already been reverted. In this way, in the presence of independent actions, backward computations are not required to necessarily follow the exact execution order of forward computations in reverse. We formalise below the notions of independent (i.e., *concurrent*) actions and of causal consistency.

As in [15] and [5], we rely on the notion of transition. In *ReSπ*, a *transition* is a triplet of the form $M \xrightarrow{m,\mathcal{M},\twoheadrightarrow} N$ (resp. $M \xrightarrow{m,\mathcal{M},\rightsquigarrow} N$), where M and N are closed reachable *ReSπ* processes such that $M \twoheadrightarrow N$ (resp. $M \rightsquigarrow N$), m is the action or choice memory involved in the reduction, and $\mathcal{M}$ is the set of fork memories possibly involved in the reduction. A memory is *involved* in a reduction if it is created or removed by the reduction. We use η to denote transition *labels* $(m, \mathcal{M}, \twoheadrightarrow)$ and $(m, \mathcal{M}, \rightsquigarrow)$. If η is $(m, \mathcal{M}, \twoheadrightarrow)$, then its inverse is $\eta_\bullet = (m, \mathcal{M}, \rightsquigarrow)$ and vice versa. In a transition $M \xrightarrow{\eta} N$, we call M the *source* of the transition and N its *target*. We use τ to range over transitions; $\tau_\bullet$ denotes the inverse of transition τ.

Two transitions are *coinitial* if they have the same source, *cofinal* if they have the same target, *composable* if the target of one is the source of the other. A sequence of transitions, where each pair of sequential transitions is composable, is called a *trace*; we use σ to range over traces. Notions of target, source and composability extend naturally to traces. We use ϵ_M to denote the empty trace with source M and $\sigma_1;\sigma_2$ the composition of two composable traces σ_1 and σ_2.

We consider only transitions $M \xrightarrow{\eta} N$ where M and N do not contain threads of the form $t : (P \mid Q)$. This condition can be always satisfied by splitting all threads of this kind into sub-threads, until their disappearance in the considered terms, using the structural law $t : (P \mid Q) \equiv (\nu t_1, t_2)(t_1 : P \mid t_2 : Q \mid \langle t \rightrightarrows (t_1, t_2)\rangle)$. Moreover, since conflicts between transitions are identified by means of tag identifiers (see Definition 5 below), we only consider transitions that do not use α-conversion on tags, and that generates fork memories in a deterministic way, e.g. given a memory $\langle t \rightrightarrows (t_1, t_2)\rangle$ tags t_1 and t_2 are generated by applying an injective function to t.

The *stamp* $\lambda(\eta)$ of a transition label η identifies the threads involved in the corresponding transition, and is defined as follows (we use T to denote a set of tags $\{t_i\}_{i\in I}$):

$$\lambda(m, \mathcal{M}, \twoheadrightarrow) = \lambda(m, \mathcal{M}, \rightsquigarrow) = \lambda(m) \cup \lambda_{\lambda(m)}(\mathcal{M})$$

$$\lambda(\langle t_1 \text{ —A→ } t_2, t_1', t_2'\rangle) = \{t_1, t_2, t_1', t_2'\}$$

$$\lambda(\langle t, e?\, P{:}Q\,, t'\rangle) = \{t, t'\}$$

$$\lambda_T(\{m_i\}_{i\in I}) = \bigcup_{i\in I} \lambda_T(m_i)$$

$$\lambda_T(\langle t \rightrightarrows (t_1, t_2)\rangle) = \begin{cases} \{t_1, t_2\} & \text{if } t \in T \\ \emptyset & \text{otherwise} \end{cases}$$

The stamp of fork memories permits us to take into account the relationships between a thread and its sub-threads. This is similar to the closure over tags used in [18]. Notably, as in [15], the tags of continuation processes are inserted into a stamp, in order to take into account possible conflicts between a forward transition and a backward one. Notice also that it is instead not necessary to include in the stamp the fresh session channel created or used by a reduction. In fact, this would allow to detect conflicts between the transitions involving the memory corresponding to the creation of the channel and the transitions where such channel is used. Such conflicts, however, are already implicitly considered, since after its creation the channel is only known by the threads corresponding to the continuation processes, which are already considered in the stamp as discussed above.

We can now define when two transitions are concurrent.

Definition 5 *(Concurrent transitions).* Two coinitial transitions $M \xrightarrow{\eta_1} M_1$ and $M \xrightarrow{\eta_2} M_2$ are *in conflict* if $\lambda(\eta_1) \cap \lambda(\eta_2) \neq \emptyset$. Two coinitial transitions are *concurrent* if they are not in conflict.

Intuitively, two transitions are concurrent when they do not involve a common thread.

The following lemma characterises the causally independence among concurrent transitions.

Lemma 6 *(Square lemma). If $\tau_1 = M \xrightarrow{\eta_1} M_1$ and $\tau_2 = M \xrightarrow{\eta_2} M_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $\tau_2/\tau_1 = M_1 \xrightarrow{\eta_2} N$ and $\tau_1/\tau_2 = M_2 \xrightarrow{\eta_1} N$.*

Proof. By case analysis on the form of transitions τ_1 and τ_2 (see Appendix A.1.3). □

In order to study the causality of *ReSπ* reversibility, we introduce the notion of *causal equivalence* [4,5] between traces, denoted $\asymp$. This is defined as the least equivalence relation between traces closed under composition that obeys the following rules:

$$\tau_1 \,;\, \tau_2/\tau_1 \asymp \tau_2 \,;\, \tau_1/\tau_2 \qquad \tau \,;\, \tau_\bullet \asymp \epsilon_{\mathtt{source}(\tau)} \qquad \tau_\bullet \,;\, \tau \asymp \epsilon_{\mathtt{target}(\tau)}$$

Intuitively, the first rule states that the execution order of two concurrent transitions can be swapped, while the other rules state that the composition of a trace with its inverse is equivalent to the empty transition.

Now, we conclude with the result stating that two coinitial causally equivalent traces lead to the same final state. Thus, in such case, we can rollback to the initial state by reversing any of the two traces.

Theorem 1 *(Causal consistency). Let σ_1 and σ_2 be coinitial traces. Then, $\sigma_1 \asymp \sigma_2$ if and only if σ_1 and σ_2 are cofinal.*

Proof. By construction of $\asymp$ and by applying Lemma 6 and other two auxiliary lemmas (see Appendix A.1.3). □

6. Type discipline

In this section, first we recall the session type discipline of session-based π-calculus then we discuss how we could exploit it to type *ReSπ* processes.

6.1. Typing session-based π-calculus

The type discipline presented here is basically the one proposed in [23], except for the notation of the calculus that has been revised according to [24].

6.1.1. Types

The syntax of *sorts*, ranged over by S, S', ..., and *types*, ranged over by α, β, ..., is defined in Fig. 7. The type $![S].\alpha$ represents the behaviour of first outputting a value of sort S, then performing the actions prescribed by type α. Type $![\beta].\alpha$ represents a similar behaviour, which starts with session output (throw) instead. Types $?[S].\alpha$ and $?[\beta].\alpha$ are the dual ones, receiving values instead of sending. Type $\&[l_1 : \alpha_1, \ldots, l_n : \alpha_n]$ describes a branching behaviour: it waits with n options, and behave as type α_i if the i-th action is selected (external choice). Type $\oplus[l_1 : \alpha_1, \ldots, l_n : \alpha_n]$ represents the behaviour which would select one of l_i and then behaves as α_i, according to the selected l_i (internal choice). Type end represents inaction, acting as the unit of sequential composition. Type $\mu t.\alpha$ denotes a recursive behaviour, representing the behaviour that starts by doing α and, when variable t is encountered, recurs to α again. As in [23], we take an *equi-recursive* view of types, not distinguishing between a type $\mu t.\alpha$ and its unfolding $\alpha[\mu t.\alpha/t]$, and we are interested on *contractive* types only, i.e. for each of sub-expressions $\mu t.\mu t_1 \ldots \mu t_n.\alpha$ the body α is not t. The result is that, in a typing derivation, types $\mu t.\alpha$ and $\alpha[\mu t.\alpha/t]$ can be used interchangeably.

For each type α, we define $\overline{\alpha}$, the *dual type* of α, by exchanging ! and ?, and & and $\oplus$. The inductive definition is in Fig. 8.

Sorts	S	$::=$	bool	boolean
		$\mid$	int	integer
		$\mid$	$\langle\alpha\rangle$	shared channel
Types	α, β	$::=$	$![S].\alpha$	output
		$\mid$	$?[S].\alpha$	input
		$\mid$	$![\beta].\alpha$	throw
		$\mid$	$?[\beta].\alpha$	catch
		$\mid$	$\oplus[l_1 : \alpha_1, \ldots, l_n : \alpha_n]$	selection
		$\mid$	$\&[l_1 : \alpha_1, \ldots, l_n : \alpha_n]$	branching
		$\mid$	end	end
		$\mid$	t	type variable
		$\mid$	$\mu t.\alpha$	recursion

Fig. 7. Syntax of sorts and types.

$$\overline{![S].\alpha} = ?[S].\overline{\alpha} \qquad \overline{![\beta].\alpha} = ?[\beta].\overline{\alpha} \qquad \overline{\oplus[l_i : \alpha_i]_{i\in I}} = \&[l_i : \overline{\alpha_i}]_{i\in I}$$

$$\overline{?[S].\alpha} = ![S].\overline{\alpha} \qquad \overline{?[\beta].\alpha} = ![\beta].\overline{\alpha} \qquad \overline{\&[l_i : \alpha_i]_{i\in I}} = \oplus[l_i : \overline{\alpha_i}]_{i\in I}$$

$$\overline{\mathsf{end}} = \mathsf{end} \qquad \overline{\mu t.\alpha} = \mu t.\overline{\alpha} \qquad \overline{t} = t$$

Fig. 8. Dual types.

6.1.2. Typing system

A *sorting* (resp. a *typing*, resp. a *basis*) is a finite partial map from shared identifiers to sorts (resp. from session identifiers to types, resp. from process variables to typings). We let Γ, Γ', ...(resp. Δ, Δ', ..., resp. Θ, Θ', ...) range over sortings (resp. typings, resp. bases). We write $\Delta \cdot k : \alpha$ when $k \notin dom(\Delta)$; this notation is then extended to $\Delta \cdot \Delta'$.

Typing judgements are of the form $\Theta; \Gamma \vdash P \triangleright \Delta$ which stands for "under the environment $\Theta; \Gamma$, process P has typing Δ". The typing system is defined by the axioms and rules in Fig. 9. We call a typing *completed* when it contains only end types. A typing Δ is called *balanced* if whenever $s : \alpha, \bar{s} : \beta \in \Delta$, then $\alpha = \overline{\beta}$. We refer the interested reader to [23] for detailed comments on the rules.

6.1.3. Results

We report here the main results concerning the type discipline, namely Subject Reduction and Type Safety, borrowed from [23]. The former result states that well-typedness is preserved along computations, while the latter states that no interaction errors occur on well-typed processes.

Theorem 2 *(Subject reduction). If $\Theta; \Gamma \vdash P \triangleright \Delta$ with Δ balanced and $P \rightarrow^* Q$, then $\Theta; \Gamma \vdash Q \triangleright \Delta'$ and Δ' balanced.*

Proof. See proof of Theorem 3.3 in [23]. □

The notion of error, necessary to formalise Type Safety, is also taken from [23]. A *k-process* is a process prefixed by subject k, while a *k-redex* is the parallel composition of two k-processes either of the form $(\bar{k}!\langle e\rangle.P_1 \mid k?(x).P_2)$, or $(\bar{k} \triangleleft l_i.P \mid k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\})$. Then, P is an *error* if $P \equiv (\nu c)(Q \mid R)$ where Q is, for some k, the parallel composition of either two k-processes that do not form a k-redex, or of three or more k-processes.

Theorem 3 *(Type Safety). A program typable under a balanced channel environment never reduces to an error.*

Proof. See proof of Theorem 3.4 in [23]. □

6.2. Typing ReSπ

We show here how the notion of types, the typing system and the related results given for the π-calculus (Section 6.1) can be reused for typing *ReSπ*. The key point is that we only consider reachable *ReSπ* processes originated from *ReSπ* programs that are well-typed (according to the typing discipline of π-calculus). In fact, by statically type checking *ReSπ* programs, we already check all possible interactions that they will perform. More specifically, Subject Reduction and Type Safety ensure that all runtime processes obtained from a program by means of (forward) reductions are interaction safe. Thus, since backward computations cannot lead to new runtime processes, but just go back to terms reachable from the program via forward reductions, there is no need of type checking the content of the memories in runtime processes.

$$\Gamma \vdash \texttt{true} \triangleright \mathsf{bool} \quad [\textsc{Bool}_{tt}] \qquad \Gamma \vdash \texttt{false} \triangleright \mathsf{bool} \quad [\textsc{Bool}_{ff}] \qquad \Gamma \vdash 1 \triangleright \mathsf{int} \quad [\textsc{Int}] \quad \ldots$$

$$\frac{\Gamma \vdash e_1 \triangleright \mathsf{int} \qquad \Gamma \vdash e_2 \triangleright \mathsf{int}}{\Gamma \vdash +(e_1, e_2) \triangleright \mathsf{int}} \; [\textsc{Sum}] \qquad \frac{\Gamma \vdash e_1 \triangleright \mathsf{bool} \qquad \Gamma \vdash e_2 \triangleright \mathsf{bool}}{\Gamma \vdash \wedge(e_1, e_2) \triangleright \mathsf{bool}} \; [\textsc{And}] \quad \ldots$$

$$\Gamma \cdot u : S \vdash u \triangleright S \quad [\textsc{Id}] \qquad \frac{\Delta \;\; \text{completed}}{\Theta; \Gamma \vdash \mathbf{0} \triangleright \Delta} \; [\textsc{Inact}]$$

$$\frac{\Gamma \vdash u \triangleright \langle \alpha \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \overline{\alpha}}{\Theta; \Gamma \vdash \bar{u}(x).P \triangleright \Delta} \; [\textsc{Req}] \qquad \frac{\Gamma \vdash u \triangleright \langle \alpha \rangle \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot x : \alpha}{\Theta; \Gamma \vdash u(x).P \triangleright \Delta} \; [\textsc{Acc}]$$

$$\frac{\Gamma \vdash e \triangleright S \qquad \Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k!\langle e \rangle.P \triangleright \Delta \cdot k :![S].\alpha} \; [\textsc{Send}] \qquad \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta}{\Theta; \Gamma \vdash k!\langle k' \rangle.P \triangleright \Delta \cdot k :![\alpha].\beta \cdot k' : \alpha} \; [\textsc{Thr}]$$

$$\frac{\Theta; \Gamma \cdot x : S \vdash P \triangleright \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k?(x).P \triangleright \Delta \cdot k :?[S].\alpha} \; [\textsc{Rcv}] \qquad \frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \beta \cdot x : \alpha}{\Theta; \Gamma \vdash k?(x).P \triangleright \Delta \cdot k :?[\alpha].\beta} \; [\textsc{Cat}]$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \alpha_j}{\Theta; \Gamma \vdash k \triangleleft l_j.P \triangleright \Delta \cdot k : \oplus[l_1 : \alpha_1, \ldots, l_n : \alpha_n]} \; (1 \leq j \leq n) \; [\textsc{Sel}]$$

$$\frac{\Theta; \Gamma \vdash P_1 \triangleright \Delta \cdot k : \alpha_1 \qquad \ldots \qquad \Theta; \Gamma \vdash P_n \triangleright \Delta \cdot k : \alpha_n}{\Theta; \Gamma \vdash k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \triangleright \Delta \cdot k : \&[l_1 : \alpha_1, \ldots, l_n : \alpha_n]} \; [\textsc{Br}]$$

$$\frac{\Gamma \vdash e \triangleright \mathsf{bool} \qquad \Theta; \Gamma \vdash P \triangleright \Delta \qquad \Theta; \Gamma \vdash Q \triangleright \Delta}{\Theta; \Gamma \vdash \texttt{if}\; e \;\texttt{then}\; P \;\texttt{else}\; Q \triangleright \Delta} \; [\textsc{If}]$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \qquad \Theta; \Gamma \vdash Q \triangleright \Delta'}{\Theta; \Gamma \vdash P \mid Q \triangleright \Delta \cdot \Delta'} \; [\textsc{Conc}] \qquad \frac{\Theta; \Gamma \cdot a : S \vdash P \triangleright \Delta}{\Theta; \Gamma \vdash (\nu a)P \triangleright \Delta} \; [\textsc{Res1}]$$

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot s : \alpha \cdot \bar{s} : \overline{\alpha}}{\Theta; \Gamma \vdash (\nu s)P \triangleright \Delta} \; [\textsc{Res2}] \qquad \frac{\Theta; \Gamma \vdash P \triangleright \Delta \qquad s \;\text{not in}\; \Delta}{\Theta; \Gamma \vdash (\nu s)P \triangleright \Delta} \; [\textsc{Res3}]$$

$$\Theta \cdot X : \Delta; \Gamma \vdash X \triangleright \Delta \quad [\textsc{Var}] \qquad \frac{\Theta \cdot X : \Delta; \Gamma \vdash P \triangleright \Delta}{\Theta; \Gamma \vdash \mu X.P \triangleright \Delta} \; [\textsc{Rec}]$$

Fig. 9. Typing system for π-calculus.

Now, before formally showing how the typing discipline of π-calculus extends to *ReS*π, we introduce a few auxiliary definitions and results.

Definition 6 *(Reachable processes for typed ReSπ).* The set of *reachable processes* for the typed *ReS*π only contains: *(i)* programs M such that $\Theta; \Gamma \vdash \phi(M) \triangleright \Delta$ with Δ balanced, and (ii) runtime processes M obtained by forward reductions from the above programs.

Property 1. *Let M be a reachable process. If $M \rightarrowtail M'$ then M' is a reachable process.*

Proof. The proof follows from Definition 6 and Lemma 5 (see Appendix A.2). □

The notion of well-typedness for *ReS*π, expressed by the judgement $\Theta; \Gamma \vdash_r M \triangleright \Delta$, is defined in terms of the well-typedness notion introduced for the π-calculus.

Definition 7 *(Well-typedness).* $\Theta; \Gamma \vdash_r M \triangleright \Delta$ if and only if $\Theta; \Gamma \vdash \phi(M) \triangleright \Delta$, with Δ balanced.

Thus, Subject Reduction extends to *ReS*π terms as follows.

Theorem 4 *(Subject Reduction). Let M be a reachable process. If $\Theta; \Gamma \vdash_r M \triangleright \Delta$ with Δ balanced and $M \rightarrowtail M'$, then $\Theta; \Gamma \vdash_r M' \triangleright \Delta'$ and Δ' balanced.*

Proof. The proof relies on Theorem 2 (see Appendix A.2). □

We conclude by showing how the notion of error and Type Safety of π-calculus extends to *ReS*π. A *ReS*π process M is an *error* if and only if $\phi(M)$ is an error.

Theorem 5 *(Type Safety). A ReSπ program typable under a balanced channel environment never reduces to an error.*

Proof. By the notion of *ReS*π error and by Theorem 3, we have that typable programs are not errors. Then, by Theorem 4 we have the thesis. □

6.3. Typing the multiple providers scenario

Coming back to the scenario introduced in Section 3.3 and specified in *ReSπ* in Section 4.3, we can easily verify that the process

$$t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2}$$

is well-typed (assuming that the unspecified processes P_{acc}, P_{neg}, Q_{acc} and Q_{neg} are properly typable). In particular, the channel a_{login} can be typed by the shared channel type

$$\langle\, ?[\mathsf{Request}].\,![\mathsf{Quote}].\,\&[l_{acc} : \alpha_{acc}\,,\, l_{neg} : \alpha_{neg}\,,\, l_{rej} : \mathsf{end}]\,\rangle$$

where we use sorts Request and Quote to type requests and quotes, respectively.

Let us consider now a scenario where the client wishes to concurrently submit two different requests to the same provider, which would be able to concurrently serve them. Consider in particular the following specification of the client (the provider one is dual):

$$\overline{a_{login}}(x).\,(\,x!\langle \mathsf{srv\_req\_1}\rangle.\,P_1 \mid x!\langle \mathsf{srv\_req\_2}\rangle.\,P_2\,)$$

The new specification of the scenario is clearly not well-typed, due to the use of parallel threads within the same session. This forbids us from mixing up messages related to different requests and wrongly delivering them. In order to properly concurrently submit separate requests, the client must instantiate separate sessions with the provider, one for each request.

The session type discipline, indeed, forces concurrent interactions to follow structured patterns that guarantee the correctness of communication. For what concerns reversibility, linear use of session channels limits the effect of causal consistency, since concurrent interactions along the same session are prevented and, hence, the backtracking of a given session follows a single path. Of course, interactions along different sessions are still concurrent and, therefore, it is important to use a causal-consistent rollback to revert them.

7. Committable sessions

The calculus *ReSπ* discussed so far is *fully* reversible, i.e. backward computations are always enabled. Full reversibility provides theoretical foundations for studying reversibility in session-based π-calculus, but it is not suitable for a practical use on structured communication-based programming. Therefore, in this section, we enrich the framework to allow computation to go backward and forward along a session, allowing the involved parties to try different interactions, until the session is successfully completed. This is achieved by adding a specific action to the calculus for irreversibly committing the closure of sessions.

It is worth noticing that the fully reversible characterisation of the calculus permits us to prove that its machinery for reversibility (i.e., memories and their usage) soundly works with respect to the expected properties of a reversible calculus. This remains valid also for the extension proposed here. In fact, as clarified below, the extended calculus basically prunes some computations allowed in *ReSπ*, which corresponds to backward and forward actions that are undesired after a session closure.

7.1. ReSπ with commit

The syntax of *ReSπC* (*Reversible Session-based π-calculus with Commit*) is obtained from that of *ReSπ* (given in Fig. 4) by extending the syntactic category of processes P with process $\mathtt{commit}(k).P$, and by extending the syntactic category of memories m with the *commit memory* $\langle t_1 - \surd(s) \rightarrow t_2\rangle$. This new memory simply registers the closing event of the session identified by s due to an agreement of threads tagged by t_1 and t_2.

The irreversible closure of a session is achieved by the synchronisation on its session channel s of two threads of the form $t_1 : \mathtt{commit}(\bar{s}).P_1$ and $t_2 : \mathtt{commit}(s).P_2$. This synchronisation acts similarly to the 'cut' operator in Prolog, as both mechanisms are used to prevent unwanted backtracking. After the synchronisation, since the session s is closed, the continuations P_1 and P_2 can no longer use the session channel s; this check is statically enforced by the type system for *ReSπC* (presented later on).

Formally, the semantics of *ReSπC* is obtained by adding the following rule to those defining the reduction relation of *ReSπ* (Fig. 6):

$$\begin{array}{ll} t_1 : \mathtt{commit}(\bar{k}).P_1 \mid t_2 : \mathtt{commit}(k).P_2 & (k = s \text{ or } k = \bar{s}) \ [\textsc{commit}] \\ \twoheadrightarrow \ (\nu t_1', t_2')(t_1' : P_1 \mid t_2' : P_2 \mid \langle t_1 - \surd(s) \rightarrow t_2\rangle) & \end{array}$$

Since `commit` is an irreversible action that will never be backtracked, there is no need to remember information about the continuation processes in the generated memory. For the same reason, there is no backward rule inverse to [COMMIT].

For what concerns the type discipline, types α (defined in Fig. 7) are extended with type $\surd$, while the typing system is extended with the following rule:

$$\frac{\Theta; \Gamma \vdash P \triangleright \Delta \cdot k : \mathsf{end}}{\Theta; \Gamma \vdash \mathtt{commit}(k).P \triangleright \Delta \cdot k : \surd} \ [\textsc{Commit}]$$

which ensures that after the commit the session is closed.

7.2. Irreversibility propagation

When a `commit` action is executed, all actions that caused it became unbacktrackable, although they were themselves reversible. In other words, a `commit` action creates a *domino effect* that disables the possibility of reversing the session actions previously performed.

To formalise the domino effect caused by the `commit` irreversible action, we have to introduce the following notions of *head* and *tail* of memories:

$$\begin{array}{ll}
\mathsf{head}(\langle t_1 \text{ –A}\!\!\rightarrow t_2, t'_1, t'_2\rangle) = \{t_1, t_2\} & \mathsf{tail}(\langle t_1 \text{ –A}\!\!\rightarrow t_2, t'_1, t'_2\rangle) = \{t'_1, t'_2\} \\
\mathsf{head}(\langle t, e?\, P{:}Q\,, t'\rangle) = \{t\} & \mathsf{tail}(\langle t, e?\, P{:}Q\,, t'\rangle) = \{t'\} \\
\mathsf{head}(\langle t \rightrightarrows (t_1, t_2)\rangle) = \{t\} & \mathsf{tail}(\langle t \rightrightarrows (t_1, t_2)\rangle) = \{t_1, t_2\} \\
\mathsf{head}(\langle t_1 - \surd(s) \rightarrow t_2\rangle) = \{t_1, t_2\} & \mathsf{tail}(\langle t_1 - \surd(s) \rightarrow t_2\rangle) = \emptyset
\end{array}$$

Using the terminology of [5], we say that a memory is *locked* when the event stored inside can never be reverted, because the conditions triggering the corresponding backward reduction will never be satisfied. Specifically, to perform a backward reduction is required the coexistence of a memory with threads properly tagged, and the latter will never be available due to an irreversible action. Let us now formalise, given a process M, its set $\mathcal{L}_M$ of locked memories.

Definition 8 *(Locked memories).* Let M be a *ReSπC* process and $\mathcal{M}_M$ stand for the set of memories occurring in M. The set $\mathcal{L}_M$ of locked memories of M is defined as follows:

- $\langle t_1 - \surd(s) \rightarrow t_2\rangle \in \mathcal{M}_M \quad \Rightarrow \quad \langle t_1 - \surd(s) \rightarrow t_2\rangle \in \mathcal{L}_M$
- $m \in \mathcal{L}_M,\ m' \in \mathcal{M}_M,\ t \in \mathsf{head}(m),\ t \in \mathsf{tail}(m') \quad \Rightarrow \quad m' \in \mathcal{L}_M$

The first point says that a commit memory is locked, while the second point describes the propagation of the locking effect, i.e. the event within m depends on the event within m' (because the latter generates a thread involved in the former) and hence also m' is locked. Of course, $\mathcal{L}_M \subseteq \mathcal{M}_M$.

Now, we can demonstrate the main result about *ReSπC*, stating that committed sessions cannot be reverted (Theorem 6). This result is based on the notion of reversible memory (Definition 9) and on Lemma 7, ensuring that locked memories cannot be reverted. We use $\rightsquigarrow^+$ to denote the transitive closure of $\rightsquigarrow$.

Definition 9. Let M be a *ReSπC* process. A memory $m \in \mathcal{M}_M$ is *reversible* if there exists a process M' such that $M \rightsquigarrow^+ M'$ and $m \notin \mathcal{M}_{M'}$.

Intuitively, a memory can be reverted if there exists a backward computation that consumes it to restore the threads it memorises.

Lemma 7. *Let M be a ReSπC process. If $m \in \mathcal{L}_M$ then m is not reversible.*

Proof. The proof proceeds by contradiction (see Appendix A.3). □

Theorem 6. *Let M be a ReSπC process and s a session committed in M. Then, all interactions performed along s cannot be reverted.*

Proof. The proof relies on Lemma 7 (see Appendix A.3). □

7.3. The multiple providers scenario in ReSπC

Let us consider again the multiple providers scenario specified in *ReSπ* in Section 4.3.

Suppose now that, for the sake of simplicity, the client and the first provider commit the session immediately after the acceptance of the quote. That is, P_{acc} and Q_{acc} stand for $\mathtt{commit}(x)$ and $\mathtt{commit}(y)$, respectively. Thus, the computation described in Section 3.3 in *ReSπC* corresponds to:

$$
\begin{aligned}
& t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2} \\
& \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \twoheadrightarrow \\
& M = (\nu s, t_1^1, t_2^1, \ldots, t_1^5, t_2^4) \\
& \qquad (t_1^5 : \mathtt{commit}(\bar{s}) \mid t_2^4 : \mathtt{commit}(s) \\
& \qquad \mid \langle t_1 \text{--}a_{login} \ldots\!\rightarrow t_2, t_1^1, t_2^1 \rangle \mid \ldots \mid \langle t_1^4 \text{--}s \triangleleft l_{acc} \ldots\!\rightarrow t_2^3, t_1^5, t_2^4 \rangle) \\
& \qquad \mid P_{provider2}
\end{aligned}
$$

M can now evolve as follows:

$$
\begin{aligned}
M \twoheadrightarrow M' = & (\nu s, t_1^1, t_2^1, \ldots, t_1^5, t_2^4, t_1^6, t_2^5) \\
& (t_1^6 : \mathbf{0} \mid t_2^5 : \mathbf{0} \mid \langle t_1^5 - \surd(s) \rightarrow t_2^4 \rangle \\
& \mid \langle t_1 \text{--}a_{login} \ldots\!\rightarrow t_2, t_1^1, t_2^1 \rangle \mid \ldots \mid \langle t_1^4 \text{--}s \triangleleft l_{acc} \ldots\!\rightarrow t_2^3, t_1^5, t_2^4 \rangle) \\
& \mid P_{provider2}
\end{aligned}
$$

The memory $\langle t_1^5 - \surd(s) \rightarrow t_2^4 \rangle$ generated by this last forward reduction is locked (first point of Definition 8). Thus, also memory $\langle t_1^4 \text{--}s \triangleleft l_{acc} \ldots\!\rightarrow t_2^3, t_1^5, t_2^4 \rangle$ is locked, since its tail contains tags belonging to the head of the commit memory (second point of Definition 8). Repeatedly applying the second point of Definition 8, we obtain that $\mathcal{L}_{M'} = \mathcal{M}_{M'}$, i.e. all memory of M' are locked. This means that $M' \not\rightarrow$ and hence, as desired, the computation along session s cannot be reverted.

8. Concluding remarks

To bring the benefits of reversible computing to structured communication-based programming, we have defined a theoretical framework based on π-calculus that can be used as formal basis for studying the interplay between (causal-consistent) reversibility and session-based structured interaction. We conclude the paper by discussing the main directions of ongoing and future work.

Concerning the reversible calculus, we plan to investigate the definition of a syntactic characterisation of consistent terms, which statically enforces history consistency in memories (as in [15]), rather than using the current definition of reachable process (as in [20,18]). In line with [17], we also plan to enrich the language with primitives and mechanisms to control reversibility.

For what concerns the typing discipline of *ReSπ*, we intend to investigate the definition of a typing system capable of type checking contents of memories, in order to identify interaction errors that could be caused by terms restored from memories. This permits us to extend the class of *ReSπ* programs to include also processes with memories, thus allowing programmers to write also this kind of reversible code. We think that the required checks would resemble the semantics of the *roll* primitive in [17], which can be then used as a source of inspiration.

Coming to the extension with committable sessions, it is worth noticing that action `commit` must be carefully used in case of subordinate sessions. For example, let us consider the typical three-party scenario where a *Customer* sends an order to a *Seller* that, in his own turn, contacts a *Shipper* for the delivery. We have that the session between the *Seller* and the *Shipper* is subordinated to the session between the *Customer* and the *Seller*. Now, when the main session is committed, also the subordinate session is involved in the commit. This is usually desirable, because the commit acts on the whole transaction and, hence, after the commit the interaction with the *Shipper* cannot be reverted. However, if the subordinate session is previously committed, the main session is affected, because interaction performed before the commit cannot be reverted. This latter situation is typically undesirable; therefore, as a best practice, `commit` should be not used by subordinate sessions. We plan to devise a static analysis technique supporting this disciplined use of `commit` in presence of subordinate sessions.

As longer-term goal, we intend to apply the proposed approach to other session-based formalisms, which consider, e.g., asynchronous sessions and multiparty sessions. We also plan to investigate implementation issues that may arise when incorporating the approach into standard programming languages, in particular in case of a distributed setting. The rollback mechanism incorporated in the semantics of the language would require low-level synchronisations between the involved parties.

Appendix A. Proofs

A.1. Proofs of Section 5

A.1.1. Correspondence with π-calculus

Lemma 1. *Let M and N be two ReSπ processes. If $M \equiv N$ then $\phi(M) \equiv \phi(N)$.*

Proof. We proceed by induction on the derivation of $M \equiv N$. For most of the laws in Fig. 5 the conclusion is trivial because $\phi(M) \equiv \phi(N)$ directly corresponds to a law for π-calculus in Fig. 2. Instead, for the eighth and twelve laws, we easily conclude because by applying ϕ we obtain the identity. □

Lemma 2. *Let M and N be two ReSπ processes. If $M \twoheadrightarrow N$ then $\phi(M) \rightarrow \phi(N)$.*

Proof. We proceed by induction on the derivation of $M \twoheadrightarrow N$. Base cases:

- [FWCON]: We have that $M = t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2$ and $N = (\nu s, t'_1, t'_2)(t'_1 : P_1[\bar{s}/x] \mid t'_2 : P_2[s/y] \mid \langle t_1 - a(x)(y)(\nu s)P_1P_2 \rightarrow t_2, t'_1, t'_2\rangle)$ with $s, \bar{s} \notin \mathrm{fse}(P_1, P_2)$. By definition of ϕ, we obtain $\phi(M) = \bar{a}(x).P_1 \mid a(y).P_2$. Now, by applying rules [CON] and [STR], we have $\phi(M) \rightarrow (\nu s)(P_1[\bar{s}/x] \mid P_2[s/y] \mid \mathbf{0}) = P$. By definition of ϕ, we have $\phi(N) = P$ that permits us to conclude.
- [FWCOM], [FWLAB], [FWIF1], [FWIF2]: These cases are similar to the previous one.

Inductive cases:

- [FWPAR]: We have that $M = M_1 \mid M_2$ and $N = M'_1 \mid M_2$. By the premise of rule [FWPAR], we also have $M_1 \twoheadrightarrow M'_1$ from which, by induction, we obtain $\phi(M_1) \rightarrow \phi(M'_1)$. By definition of ϕ, we get $\phi(M) = \phi(M_1) \mid \phi(M_2)$. By applying rule [PAR], we have $\phi(M) \rightarrow \phi(M'_1) \mid \phi(M_2) = P$. Thus, by definition of ϕ, we have $\phi(N) = P$ that directly permits us to conclude.
- [FWRES]: This case is similar to the previous one. In particular, when the restricted name is a tag, it is not even necessary to apply rule [RES], because the forgetful map erases the restriction.
- [FWSTR]: By the premise of rule [FWSTR], we have $M \equiv M'$, $M' \twoheadrightarrow N'$ and $N' \equiv N$. By induction, we obtain $\phi(M') \rightarrow \phi(N')$. By applying Lemma 1, we have $\phi(M) \equiv \phi(M')$ and $\phi(N') \equiv \phi(N)$ that allow us to conclude. □

Lemma 3. *Let P and Q be two π-calculus processes. If $P \equiv Q$ then for any ReSπ process M such that $\phi(M) = P$ there exists a ReSπ process N such that $\phi(N) = Q$ and $M \equiv N$.*

Proof. The proof is straightforward. Indeed, given a *ReSπ* process M such that $\phi(M) = P$, it must have the form $(\nu\tilde{t})(t : P \mid \prod_{i\in I} m_i)$ up to $\equiv$. Thus, the process N, such that $\phi(N) = Q$, can be defined accordingly: $N \equiv (\nu\tilde{t})(t : Q \mid \prod_{i\in I} m_i)$. Now, we can conclude by exploiting the ninth law in Fig. 5, i.e. $t : P \equiv t : Q$ if $P \equiv Q$, and the fact that relation $\equiv$ on *ReSπ* processes is a congruence. □

Lemma 4. *Let P and Q be two π-calculus processes. If $P \rightarrow Q$ then for any ReSπ process M such that $\phi(M) = P$ there exists a ReSπ process N such that $\phi(N) = Q$ and $M \twoheadrightarrow N$.*

Proof. We proceed by induction on the derivation of $P \rightarrow Q$. Base cases:

- [CON]: We have that $P = \bar{a}(x).P_1 \mid a(y).P_2$ and $Q = (\nu s)(P_1[\bar{s}/x] \mid P_2[s/y])$ with $s, \bar{s} \notin \mathrm{fse}(P_1, P_2)$. Let M be a *ReSπ* process such that $\phi(M) = P$, it must have the form $(\nu\tilde{t})(t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2 \mid \prod_{i\in I} m_i)$ up to $\equiv$. Thus, by applying rules [FWCON], [FWPAR], [FWRES] and [FWSTR], we get $M \twoheadrightarrow (\nu\tilde{t}, s, t'_1, t'_2)(t'_1 : P_1[\bar{s}/x] \mid t'_2 : P_2[s/y] \mid \langle t_1 - a(x)(y)(\nu s)P_1P_2 \rightarrow t_2, t'_1, t'_2\rangle \mid \prod_{i\in I} m_i) = N$. We conclude by applying ϕ to N, since we obtain $\phi(N) = Q$.
- [COM], [LAB], [IF1], [IF2]: These cases are similar to the previous one.

Inductive cases:

- [PAR]: We have that $P = P_1 \mid P_2$ and $Q = P'_1 \mid P_2$. Let M be a *ReSπ* process such that $\phi(M) = P_1 \mid P_2$. We have $M \equiv (\nu\tilde{t})(t_1 : P_1 \mid t_2 : P_2 \mid \prod_{i\in I} m_i) \equiv (\nu\tilde{t'})(M_1 \mid t_2 : P_2 \mid \prod_{j\in J} m_j)$ with $M_1 \equiv (\nu\tilde{t''})(t_1 : P_1 \mid \prod_{k\in K} m_k)$, $\tilde{t} = \tilde{t'}, \tilde{t''}$ and $J \cup K = I$. By the premise of rule [PAR], we also have $P_1 \rightarrow P'_1$ from which, by induction, since $\phi(M_1) = P_1$, there exists M'_1 such that $\phi(M'_1) = P'_1$ and $M_1 \twoheadrightarrow M'_1$. Thus, by applying rules [FWPAR], [FWRES] and [FWSTR], we get $M \twoheadrightarrow (\nu\tilde{t'})(M'_1 \mid t_2 : P_2 \mid \prod_{j\in J} m_j) = N$. We conclude by applying ϕ to N, because $\phi(N) = \phi(M'_1) \mid P_2 = P'_1 \mid P_2 = Q$.
- [RES]: This case is similar to the previous one.
- [STR]: We have that $P \equiv P'$, $Q \equiv Q'$ and $P' \rightarrow Q'$. Let M be a process such that $\phi(M) = P$. By applying Lemma 3, there exists M' such that $\phi(M') = P'$ and $M \equiv M'$. By induction, there is N' such that $\phi(N') = Q'$ and $M' \twoheadrightarrow N'$. By applying Lemma 3 again, there exists N such that $\phi(N) = Q$ and $N \equiv N'$. By applying rule [FWSTR], we conclude $M \twoheadrightarrow N$. □

A.1.2. Loop lemma

Lemma 5. *Let M and N be two reachable ReSπ processes. $M \twoheadrightarrow N$ if and only if $N \rightsquigarrow M$.*

Proof. The proof for the *if* part is by induction on the derivation of $M \twoheadrightarrow N$. Base cases:

- [FWCON]: We have that $M = t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2$ and $N = (\nu s, t'_1, t'_2)(t'_1 : P_1[\bar{s}/x] \mid t'_2 : P_2[s/y] \mid \langle t_1 - a(x)(y)(\nu s)P_1P_2 \rightarrow t_2, t'_1, t'_2\rangle)$ with $s, \bar{s} \notin \mathrm{fse}(P_1, P_2)$. By applying rule [BWCON], we can directly conclude $N \rightsquigarrow M$.

- [FWCOM], [FWLAB], [FWIF1], [FWIF2]: These cases are similar to the previous one.

Inductive cases:

- [FWPAR]: We have that $M = N_1 \mid N_2$, $N = N_1' \mid N_2$ and $N_1 \twoheadrightarrow N_1'$. By induction $N_1' \rightsquigarrow N_1$. Thus, we conclude by applying rule [BWPAR], since we get $N = N_1' \mid N_2 \rightsquigarrow N_1 \mid N_2 = M$.
- [FWRES]: We have that $M = (\nu h)M_1$, $N = (\nu h)M_1'$ and $M_1 \twoheadrightarrow M_1'$. By induction $M_1' \rightsquigarrow M_1$. Thus, we conclude by applying rule [BWRES], since we get $N = (\nu h)M_1' \rightsquigarrow (\nu h)M_1 = M$.
- [FWSTR]: We have that $M \equiv M'$, $N \equiv N'$ and $M' \twoheadrightarrow N'$. By induction $N' \rightsquigarrow M'$. Thus, we conclude by applying rule [BWSTR], since we directly get $N \rightsquigarrow M$.

The proof for the *only if* part is by induction on the derivation of $N \rightsquigarrow M$. Base cases:

- [BWCON]: We have that $N = (\nu s, t_1', t_2')(t_1' : P \mid t_2' : Q \mid \langle t_1 \text{–}a(x)(y)(\nu s)P_1P_2\rightarrow t_2, t_1', t_2'\rangle)$ and $M = t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2$. Since N is a reachable process, memory $\langle t_1 \text{–}a(x)(y)(\nu s)P_1P_2\rightarrow t_2, t_1', t_2'\rangle$ has been generated by a synchronisation between threads $t_1 : \bar{a}(x).P_1$ and $t_2 : a(y).P_2$, producing a session channel s and two continuation processes $P_1[\bar{s}/x]$ and $P_2[s/y]$ tagged by t_1' and t_2', respectively. Now, by tag uniqueness implied by reachability and use of restriction in tag generation, P and Q must coincide with $P_1[\bar{s}/x]$ and $P_2[s/y]$, respectively. Therefore, by applying rule [FWCON], we can directly conclude $M \twoheadrightarrow N$.
- [BWCOM], [BWLAB], [BWIF]: These cases are similar to the previous one.

Inductive cases:

- [BWPAR]: We have that $N = N_1 \mid N_2$, $M = N_1' \mid N_2$ and $N_1 \rightsquigarrow N_1'$. By induction $N_1' \twoheadrightarrow N_1$. Thus, we conclude by applying rule [FWPAR], since we get $M = N_1' \mid N_2 \twoheadrightarrow N_1 \mid N_2 = N$.
- [BWRES]: We have that $N = (\nu h)N_1$, $M = (\nu h)N_1'$ and $N_1 \rightsquigarrow N_1'$. By induction $N_1' \twoheadrightarrow N_1$. Thus, we conclude by applying rule [FWRES], since we get $M = (\nu h)N_1' \twoheadrightarrow (\nu h)N_1 = N$.
- [BWSTR]: We have that $N \equiv N'$, $M \equiv M'$ and $N' \rightsquigarrow M'$. By induction $M' \twoheadrightarrow N'$. Thus, we conclude by applying rule [FWSTR], since we directly get $M \twoheadrightarrow N$. □

A.1.3. Causal consistency

Lemma 6. *If $\tau_1 = M \xrightarrow{\eta_1} M_1$ and $\tau_2 = M \xrightarrow{\eta_2} M_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $\tau_2/\tau_1 = M_1 \xrightarrow{\eta_2} N$ and $\tau_1/\tau_2 = M_2 \xrightarrow{\eta_1} N$.*

Proof. By case analysis on the form of transitions τ_1 and τ_2.

- $M \xrightarrow{m_1, \mathcal{M}_1, \twoheadrightarrow} M_1$ and $M \xrightarrow{m_2, \mathcal{M}_2, \twoheadrightarrow} M_2$. The two transitions can be any combination of forward reductions. Let us consider the case of two communication (the other cases are similar). Since the two transitions are concurrent, the involved threads are four distinct threads (two sending threads and two receiving ones). In particular, we consider below two communications along different sessions; in fact, the type discipline in Section 6 forbids concurrent communications along the same session (although, in this proof, this kind of concurrent communications would not cause any problem). Thus, in the considered case, the source of the two transitions is as follows:

$$\begin{aligned} M \equiv (\nu s_1, s_2, \tilde{t})(\, & t_1 : \bar{s_1}!\langle e_1\rangle.P_1 \mid t_2 : s_1?(x_1).P_2 \\ & \mid t_3 : \bar{s_2}!\langle e_2\rangle.P_3 \mid t_4 : s_2?(x_2).P_4 \mid M' \,) \end{aligned}$$

where t_1, t_2, t_3, t_4 are in $\tilde{t}$. Then, $M \twoheadrightarrow M_1$ with

$$\begin{aligned} M_1 \equiv (\nu s_1, s_2, \tilde{t}, t_1', t_2')(\, & t_1' : P_1 \mid t_2' : P_2[v_1/x_1] \mid m_1 \\ & \mid t_3 : \bar{s_2}!\langle e_2\rangle.P_3 \mid t_4 : s_2?(x_2).P_4 \mid M' \,) \end{aligned}$$

where $e_1 \downarrow v_1$ and $m_1 = \langle t_1 \text{–}\bar{s_1}\langle e_1\rangle(x_1)P_1P_2\rightarrow t_2, t_1', t_2'\rangle$. Similarly, $M \twoheadrightarrow M_2$ with

$$\begin{aligned} M_2 \equiv (\nu s_1, s_2, \tilde{t}, t_3', t_4')(\, & t_1 : \bar{s_1}!\langle e_1\rangle.P_1 \mid t_2 : s_1?(x_1).P_2 \\ & \mid t_3' : P_3 \mid t_4' : P_4[v_2/x_2] \mid m_2 \mid M' \,) \end{aligned}$$

where $e_2 \downarrow v_2$ and $m_2 = \langle t_3 \text{–}\bar{s_2}\langle e_2\rangle(x_2)P_3P_4\rightarrow t_4, t_3', t_4'\rangle$. Now, we have that $M_1 \twoheadrightarrow N$ with

$$\begin{aligned} N \equiv (\nu s_1, s_2, \tilde{t}, t_1', t_2', t_3', t_4')(\, & t_1' : P_1 \mid t_2' : P_2[v_1/x_1] \mid m_1 \\ & \mid t_3' : P_3 \mid t_4' : P_4[v_2/x_2] \mid m_2 \mid M' \,) \end{aligned}$$

As desired, we also have that $M_2 \twoheadrightarrow N$.

- $M \xrightarrow{m_1, \mathcal{M}_1, \twoheadrightarrow} M_1$ and $M \xrightarrow{m_2, \mathcal{M}_2, \rightsquigarrow} M_2$. The two transitions can be any combination of a forward rule and a backward one, respectively. Let us consider the case of a forward communication and the undo of a choice (again, the other cases are similar). Thus, in the considered case, the source of the two transitions is as follows:

$$M \equiv (\nu s, \tilde{t}, t'_3)(t_1 : \bar{s}!\langle e_1 \rangle.P_1 \mid t_2 : s?(x_1).P_2 \\ \mid t'_3 : P_3 \mid m_2 \mid M')$$

where $m_2 = \langle t_3, e_2?\, P_3{:}P_4, t'_3 \rangle$, $e \downarrow \texttt{true}$, and t_1, t_2, t_3 are in $\tilde{t}$. Notably, since the two transitions are concurrent, the continuation tag in m_2 can be neither t_1 nor t_2 (indeed, in the above process M this tag is t'_3). Then, $M \twoheadrightarrow M_1$ with

$$M_1 \equiv (\nu s, \tilde{t}, t'_3, t'_1, t'_2)(t'_1 : P_1 \mid t'_2 : P_2[v_1/x_1] \mid m_1 \\ \mid t'_3 : P_3 \mid m_2 \mid M')$$

where $e_1 \downarrow v_1$ and $m_1 = \langle t_1 \, {-}\bar{s}\langle e_1 \rangle(x_1) P_1 P_2 {\rightarrow}\, t_2, t'_1, t'_2 \rangle$. Now, we also have that $M \rightsquigarrow M_2$ with

$$M_2 \equiv (\nu s, \tilde{t})(t_1 : \bar{s}!\langle e_1 \rangle.P_1 \mid t_2 : s?(x_1).P_2 \\ \mid t_3 : \texttt{if}\ e_2\ \texttt{then}\ P_3\ \texttt{else}\ P_4 \mid M')$$

As desired, both M_1 and M_2 can then evolve with a backward and forward reduction, respectively, to N:

$$N \equiv (\nu s, \tilde{t}, t'_1, t'_2)(t'_1 : P_1 \mid t'_2 : P_2[v_1/x_1] \mid m_1 \\ \mid t_3 : \texttt{if}\ e_2\ \texttt{then}\ P_3\ \texttt{else}\ P_4 \mid M')$$

- $M \xrightarrow{m_1, \mathcal{M}_1, \rightsquigarrow} M_1$ and $M \xrightarrow{m_2, \mathcal{M}_2, \rightsquigarrow} M_2$. Similar to the first case.
- $M \xrightarrow{m_1, \mathcal{M}_1, \rightsquigarrow} M_1$ and $M \xrightarrow{m_2, \mathcal{M}_2, \twoheadrightarrow} M_2$. Similar to the second case. □

The proof of the Causal Consistency theorem follows the (standard) pattern used in [15,5]. In particular, the proof relies on two auxiliary lemmas. The first lemma permits us to rearrange a trace as a composition of a backward trace and a forward one. The second lemma permits a forward trace to be shortened.

Lemma 8. *Let σ be a trace. There exist σ' and σ'' both forward traces such that $\sigma \asymp \sigma'_\bullet; \sigma''$.*

Proof. We prove this by lexicographic induction on the length of σ, and the distance to the beginning of σ of the earliest pair of transitions in σ contradicting the property. If there is no such contradicting pair, then we are done. If there is one, say a pair of the form $\tau; \tau'_\bullet$ with τ and τ' forward transitions, we have two possibilities: either τ and τ' are concurrent, or they are in conflict. In the first case, τ and $\tau'_\bullet$ can be swapped by using Lemma 6, resulting in a later earliest contradicting pair. Then, the result follows by induction, since swapping transitions keeps the total length constant. In the second case, there is a conflict on a tag, because it belongs to the stamps of both transitions. Again, we have two sub-cases: either the memory involved in the two transitions is the same or not. In the first sub-case we have $\tau = \tau'$, and then we can apply Lemma 5 to remove $\tau; \tau_\bullet$. Hence, the total length of σ decreases and, again, by induction we obtain the thesis. Instead, the second sub-case never happens. Indeed, let τ generate a memory $m_1 = \langle t, e?\, P{:}Q\,, t' \rangle$ (the case with the action memory is similar). A conflict with τ' would be caused by the presence of t or t' in the memory m_2 removed by $\tau'_\bullet$ (and, by hypothesis, different from m_1). However, t cannot occur in m_2, because the transition τ consumed the thread uniquely tagged by t, which then cannot be involved in the other transition. Also t' cannot occur in m_2, because the thread uniquely tagged by t' has been generated by τ; thus, another forward transition must take place before $\tau'_\bullet$ to involve this thread so that t' could occur in m_2. □

Lemma 9. *Let σ_1 and σ_2 be coinitial and cofinal traces, with σ_2 forward. Then, there exists a forward trace σ'_1 of length at most that of σ_1 such that $\sigma'_1 \asymp \sigma_1$.*

Proof. The proof is by induction on the length of σ_1. If σ_1 is a forward trace we are already done. Otherwise, by Lemma 8 we can write σ_1 as $\sigma_\bullet; \sigma'$ (with σ and σ' forward). Due to its form, σ_1 contains only one sequence of transitions with opposite direction, say $\tau_\bullet; \tau'$. Let m_1 be the memory removed by $\tau_\bullet$. Then, in σ' there is a forward transition generating m_1; otherwise there would be a difference with respect to σ_2, since the latter is a forward trace. Let τ_1 be the earliest such transition in σ_1. Since τ_1 is able to put back m_1, it has to be the opposite of $\tau_\bullet$, i.e. $\tau_1 = \tau$. Now, we can swap τ_1 with all the transitions between τ_1 and $\tau_\bullet$, in order to obtain a trace in which τ_1 and $\tau_\bullet$ are adjacent. To do so, we use Lemma 6, since all the transitions in between are concurrent. Assume in fact that there is a transition involving memory m_2 which is not concurrent to τ_1. A possible conflict could be caused by the presence of a continuation tag, say t, of m_1 in m_2. But this case can never happen, since t is freshly generated by the forward rule used to produce τ_1 and thus, thanks to tag uniqueness, t cannot coincide with any tag of a previously executed transition. The other possible conflict could be caused by the presence of a continuation tag of m_2 in m_1. Since $\tau_\bullet$ removes m_1, this memory cannot contain a fresh tag generated by a subsequent transition when m_2 is created. Thus, also this case can never happen. Now, when $\tau_\bullet$ and τ are adjacent, we can remove both of them using $\asymp$. The resulting trace is shorter, thus the thesis follows by inductive hypothesis. □

Theorem 1. *Let σ_1 and σ_2 be coinitial traces. Then, $\sigma_1 \asymp \sigma_2$ if and only if σ_1 and σ_2 are cofinal.*

Proof. By construction of $\asymp$, if $\sigma_1 \asymp \sigma_2$ then σ_1 and σ_2 must be coinitial and cofinal, so this direction of the theorem is verified. Thus, it just remains to prove that σ_1 and σ_2 being coinitial and cofinal implies that $\sigma_1 \asymp \sigma_2$. By Lemma 8, we know that the two traces can be written as composition of a backward trace and a forward one. The proof is by lexicographic induction on the sum of the lengths of σ_1 and σ_2 and on the distance between the end of σ_1 and the earliest pair of transitions τ_1 in σ_1 and τ_2 in σ_2 which are not equal. If all the transitions are equal then we are done. Otherwise, we have to consider four cases, depending on whether the two transitions are forward or backward.

- *τ_1 forward and τ_2 backward.* One has $\sigma_1 = \sigma_\bullet; \tau_1; \sigma'$ and $\sigma_2 = \sigma_\bullet; \tau_2; \sigma''$ for some σ, σ', and σ''. Lemma 9 applies to $\tau_1; \sigma'$, since it is a forward trace, and to $\tau_2; \sigma''$; indeed, σ_1 and σ_2 are coinitial and cofinal by hypothesis, thus also $\tau_1; \sigma'$ and $\tau_2; \sigma''$ are coinitial and cofinal. We then have that $\tau_2; \sigma''$ has a shorter equivalent forward trace, and so σ_2 has a shorter equivalent forward trace. We can conclude by induction.
- *τ_1 backward and τ_2 forward.* This case is similar to the previous one.
- *τ_1 and τ_2 forward.* We have two possibilities: they are concurrent or are not. In the latter case, they should conflict on a thread, say $t : P$, that they both consume and store in different memories. Since the two traces are cofinal, there should be a transition τ_2' in σ_2 creating the same memory as τ_1. However no other thread $t : P$ is ever created in σ_2; hence, this is not possible. Therefore, we can assume that τ_1 and τ_2 are concurrent. Let τ_2' be the transition in σ_2 creating the same memory of τ_1. We have to prove that τ_2' is concurrent to all the previous transitions. This holds since no previous transition can remove one of the processes needed for triggering τ_2' and since forward transitions can never conflict on t. Thus we can repetitively apply Lemma 6 to derive a trace equivalent to σ_2 where τ_2 and τ_2' are consecutive. We can apply a similar transformation to σ_1. Now, we can apply Lemma 6 to τ_1 and τ_2 to have two traces of the same length as before but where the first pair of different transitions is closer to the end. We then conclude by inductive hypothesis.
- *τ_1 and τ_2 backward.* Let m_1 be the memory removed by τ_1, which is surely different from the memory removed by τ_2 (indeed, the two backward transitions cannot remove the same memory). Since the two traces are cofinal, either there is another transition in σ_1 putting back the memory or there is a transition τ_1' in σ_2 removing the same memory. In the first case, τ_1 is concurrent to all the backward transitions following it, but the ones that consume processes generated by it. Thus, all such transitions have to be undone by corresponding forward transitions (since they are not possible in σ_2). Consider the last such transition: we can use Lemma 6 to make it the last backward transition. The forward transition undoing it should be concurrent to all the previous forward transitions (the reason is the same as in the previous case). We can then use Lemma 6 to make it the first forward transition. Finally, we can apply $\tau_\bullet; \tau \asymp \epsilon_{\mathtt{target}(\tau)}$ to remove the two transitions, thus shortening the trace. In this way, we obtain the thesis by inductive hypothesis. □

A.2. Proofs of Section 6

Property 1. *Let M be a reachable process. If $M \rightarrowtail M'$ then M' is a reachable process.*

Proof. The proof follows from Definition 6. Indeed, since M is a reachable process, there exists a typed program N such that $N \twoheadrightarrow^* M$, where $\twoheadrightarrow^*$ denotes the reflexive and transitive closure of $\twoheadrightarrow$. Now, we distinguish two cases:

- $M \twoheadrightarrow M'$. In this case, we have $N \twoheadrightarrow^* M \twoheadrightarrow M'$, that is $N \twoheadrightarrow^* M'$.
- $M \rightsquigarrow M'$. Here we have to sub-cases:
  - M' is a process encountered in the computation $N \twoheadrightarrow^* M$. Thus, $N \twoheadrightarrow^* M' \twoheadrightarrow^* M$. In particular, since from $M \rightsquigarrow M'$ by Lemma 5 we have $M' \twoheadrightarrow M$, we obtain $N \twoheadrightarrow^* M' \twoheadrightarrow M$.
  - M' is not encountered in $N \twoheadrightarrow^* M$. Since M' is obtained from M by a backward reduction, the memories of M' are the same of M except for the one consumed by the backward reduction (which, of course, is missing in M'). Being M a reachable process, the content of such memories is consistent. Thus, consuming all memories in M' inevitably leads to the program N, i.e. $M' \rightsquigarrow^* N$. By applying Lemma 5 to all reductions in this backward computation, we obtain $N \twoheadrightarrow^* M'$.

In all cases, M' is originated from the same program of M, hence it is a reachable process too. □

Theorem 4. *Let M be a reachable process. If $\Theta; \Gamma \vdash_r M \triangleright \Delta$ with Δ balanced and $M \rightarrowtail M'$, then $\Theta; \Gamma \vdash_r M' \triangleright \Delta'$ and Δ' balanced.*

Proof. We distinguish two cases:

1. ($\rightarrowtail = \twoheadrightarrow$). By Definition 7, from $\Theta; \Gamma \vdash_r M \triangleright \Delta$ we obtain $\Theta; \Gamma \vdash \phi(M) \triangleright \Delta$. By applying Lemma 2 to the hypothesis $M \twoheadrightarrow M'$, we have $\phi(M) \rightarrow \phi(M')$. Thus, by Theorem 2, we get $\Theta; \Gamma \vdash \phi(M') \triangleright \Delta'$ with Δ' balanced. Finally, by Definition 7, we have the thesis $\Theta; \Gamma \vdash_r M' \triangleright \Delta'$.

2. ($\rightarrowtail = \rightsquigarrow$). By Property 1, we get that process M' is reachable. Thus, by Definition 6, there exists a program N such that $\Theta; \Gamma \vdash \phi(N) \triangleright \Delta''$ with Δ'' balanced, and $N \twoheadrightarrow^* M'$. Now, we can proceed as in case 1, by applying Lemma 2 and Theorem 2 to obtain $\Theta; \Gamma \vdash \phi(M') \triangleright \Delta'$ with Δ' balanced. Again, by Definition 7, we get the thesis. □

A.3. Proofs of Section 7

Lemma 7. *Let M be a ReSπC process. If $m \in \mathcal{L}_M$ then m is not reversible.*

Proof. The proof proceeds by contradiction. Suppose that there exists a memory m such that $m \in \mathcal{L}_M$ and m is reversible. By Definition 9, there exists M' such that $M \rightsquigarrow^+ M'$ and $m \notin \mathcal{M}_{M'}$. We have two cases:

1. $m = \langle t_1 - \surd(s) \rightarrow t_2 \rangle$: this case is trivial because no rule is able to revert this kind of memory (in fact, the forward rule [Commit] is not paired with a corresponding backward rule). Thus, no process M' such that $m \notin \mathcal{M}_{M'}$ can be derived from M, which contradicts the hypothesis.
2. $m \neq \langle t_1 - \surd(s) \rightarrow t_2 \rangle$: since $m \in \mathcal{L}_M$, by Definition 8, there exists a memory $m' \in \mathcal{L}_M$ and tag t such that $t \in \text{tail}(m)$ and $t \in \text{head}(m')$. To revert m, according to rules [bwcon], [bwcom], [bwlab], and [bwif], for each tag $t' \in \text{tail}(m)$ a thread tagged by t' must be in parallel with the memory. However, the tag t in $\text{tail}(m)$ also belongs to $\text{head}(m')$, meaning that the thread tagged by t (which, we recall, is unique) has been already executed (in fact, data concerning such execution is stored in m'). Thus, no backward rule can be applied to revert m in one step. The only possibility is to revert m' before. Now, if m' is a commit memory, then we proceed as in case 1, i.e. m' cannot be reverted and, hence, m is not reversible, which is a contradiction. Otherwise, we repeat the same reasoning of case 2 for m' and proceed in this way until a commit memory is found. Indeed, this commit memory must exist by construction of $\mathcal{L}_M$ (Definition 8, first point). As in case 1, this memory cannot be reverted and, hence, all involved memories, included m, are not reversible, which is a contradiction. □

Theorem 6. *Let M be a ReSπC process and s a session committed in M. Then, all interactions performed along s cannot be reverted.*

Proof. Since s is committed in M, then there exists a memory m of the form $\langle t_1 - \surd(s) \rightarrow t_2 \rangle$ such that $m \in \mathcal{M}_M$. By Definition 8, $m \in \mathcal{L}_M$. Now, we have to prove that all other interactions performed along s corresponds to memories in $\mathcal{L}_M$. By linearity of sessions (ensured by the type discipline in Section 6), each interaction in s causally depends on one of the threads produced by the previous interaction along s. Since m corresponds to the last interaction along s (as ensured by rule [Commit] of the type system), m causally depends on all memories corresponding to the interactions along s. Since $m \in \mathcal{L}_M$, by Definition 8 (second point), all such memories are included in $\mathcal{L}_M$. Thus, by applying Lemma 7, we obtain the thesis. □

References

[1] F. Tiezzi, N. Yoshida, Towards reversible sessions, in: PLACES, in: Electron. Proc. Theoret. Comput. Sci., vol. 155, 2014, pp. 17–24.
[2] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.
[3] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes. I, Inf. Comput. 100 (1992) 1–40;
R. Milner, J. Parrow, D. Walker, A calculus of mobile processes. II, Inf. Comput. 100 (1992) 41–77.
[4] G. Berry, J. Lévy, Minimal and optimal computations of recursive programs, J. ACM 26 (1979) 148–175.
[5] V. Danos, J. Krivine, Reversible communicating systems, in: CONCUR, in: Lect. Notes Comput. Sci., vol. 3170, Springer, 2004, pp. 292–307.
[6] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, Stud. Logic Found. Math., Elsevier Science, 1985.
[7] R. De Nicola, U. Montanari, F.W. Vaandrager, Back and forth bisimulations, in: CONCUR, in: Lect. Notes Comput. Sci., vol. 458, Springer, 1990, pp. 152–165.
[8] W. Reisig, Understanding Petri Nets – Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013.
[9] I. Lanese, C.A. Mezzina, F. Tiezzi, Causal-consistent reversibility, Bull. Eur. Assoc. Theor. Comput. Sci. 114 (2014).
[10] V. Danos, J. Krivine, Transactions in RCCS, in: CONCUR, in: Lect. Notes Comput. Sci., vol. 3653, Springer, 2005, pp. 398–412.
[11] V. Danos, J. Krivine, Formal molecular biology done in CCS-R, Electron. Notes Theor. Comput. Sci. 180 (2007) 31–49.
[12] I.C.C. Phillips, I. Ulidowski, Reversing algebraic process calculi, J. Log. Algebr. Program. 73 (2007) 70–96.
[13] J.A. Bergstra, J.W. Klop, Process algebra for synchronous communication, Inf. Control 60 (1984) 109–137.
[14] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, J. ACM 31 (1984) 560–599.
[15] I. Lanese, C.A. Mezzina, J.-B. Stefani, Reversing higher-order pi, in: CONCUR, in: Lect. Notes Comput. Sci., vol. 6269, Springer, 2010, pp. 478–493.
[16] D. Sangiorgi, Bisimulation for higher-order process calculi, Inf. Comput. 131 (1996) 141–178.
[17] I. Lanese, C.A. Mezzina, A. Schmitt, J.-B. Stefani, Controlling reversibility in higher-order pi, in: CONCUR, in: Lect. Notes Comput. Sci., vol. 6901, Springer, 2011, pp. 297–311.
[18] E. Giachino, I. Lanese, C.A. Mezzina, F. Tiezzi, Causal-consistent reversibility in a tuple-based language, in: PDP, IEEE, 2015, in press.
[19] R. De Nicola, G.L. Ferrari, R. Pugliese, Klaim: a kernel language for agents interaction and mobility, IEEE Trans. Softw. Eng. 24 (1998) 315–330.
[20] I. Cristescu, J. Krivine, D. Varacca, A compositional semantics for the reversible p-calculus, in: LICS, IEEE, 2013, pp. 388–397.
[21] L. Cardelli, C. Laneve, Reversible structures, in: CMSB, ACM, 2011, pp. 131–140.
[22] A. Phillips, L. Cardelli, A programming language for composable DNA circuits, J. R. Soc. Interface 6 (2009).
[23] N. Yoshida, V.T. Vasconcelos, Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication, Electron. Notes Theor. Comput. Sci. 171 (2007) 73–93.
[24] D. Mostrous, N. Yoshida, Session-based communication optimisation for higher-order mobile processes, in: TLCA, in: Lect. Notes Comput. Sci., vol. 5608, Springer, 2009, pp. 203–218.

[25] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, Asynchronous session types and progress for object oriented languages, in: FMOODS, in: Lect. Notes Comput. Sci., vol. 4468, Springer, 2007, pp. 1–31.
[26] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: POPL, ACM, 2008, pp. 273–284.
[27] A. Bejleri, N. Yoshida, Synchronous multiparty session types, Electron. Notes Theor. Comput. Sci. 241 (2009) 3–33.
[28] M. Boreale, R. Bruni, R. De Nicola, M. Loreti, Sessions and pipelines for structured service programming, in: FMOODS, in: Lect. Notes Comput. Sci., vol. 5051, Springer, 2008, pp. 19–38.
[29] R. Bruni, I. Lanese, H.C. Melgratti, E. Tuosto, Multiparty sessions in SOC, in: COORDINATION, in: Lect. Notes Comput. Sci., vol. 5052, Springer, 2008, pp. 67–82.
[30] L. Caires, H.T. Vieira, Conversation types, Theor. Comput. Sci. 411 (2010) 4399–4440.
[31] F. Barbanera, M. Dezani-Ciancaglini, U. de'Liguoro, Compliance for reversible client/server interactions, in: BEAT, in: Electron. Proc. Theoret. Comput. Sci., vol. 162, 2014, pp. 35–42.
[32] F. Barbanera, U. de'Liguoro, Sub-behaviour relations for session-based client/server systems, Math. Struct. Comput. Sci. (2014), in press, http://dx.doi.org/10.1017/S096012951400005X.
[33] S.J. Gay, M. Hole, Subtyping for session types in the pi calculus, Acta Inform. 42 (2005) 191–225.
[34] I. Lanese, C.A. Mezzina, J.-B. Stefani, Controlled reversibility and compensations, in: RC, in: Lect. Notes Comput. Sci., vol. 7581, Springer, 2013, pp. 233–240.